INSTRUCTIONS OF METRO SYSTEM PROGRAM

December 7, 2016

Student Name: He Yan
Student ID: 1400015464
Peking University

Contents

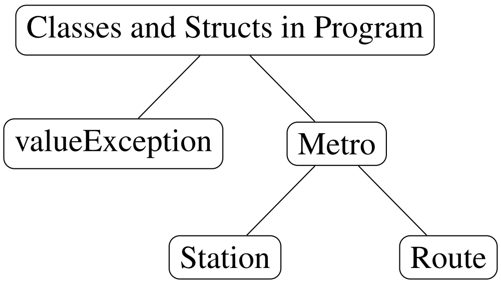
Introduction

This is a metro system which can help you find the shortest route by the names of source station and destination station. Loop and branch routes are supported.

Attention that the compiled executable file (.exe) should be in the same directory as metro data txt files such as "Beijing.txt" before running it. If you want to switch to another city or edit the directory, read the "Input" section in this instruction.

1 Program Structure

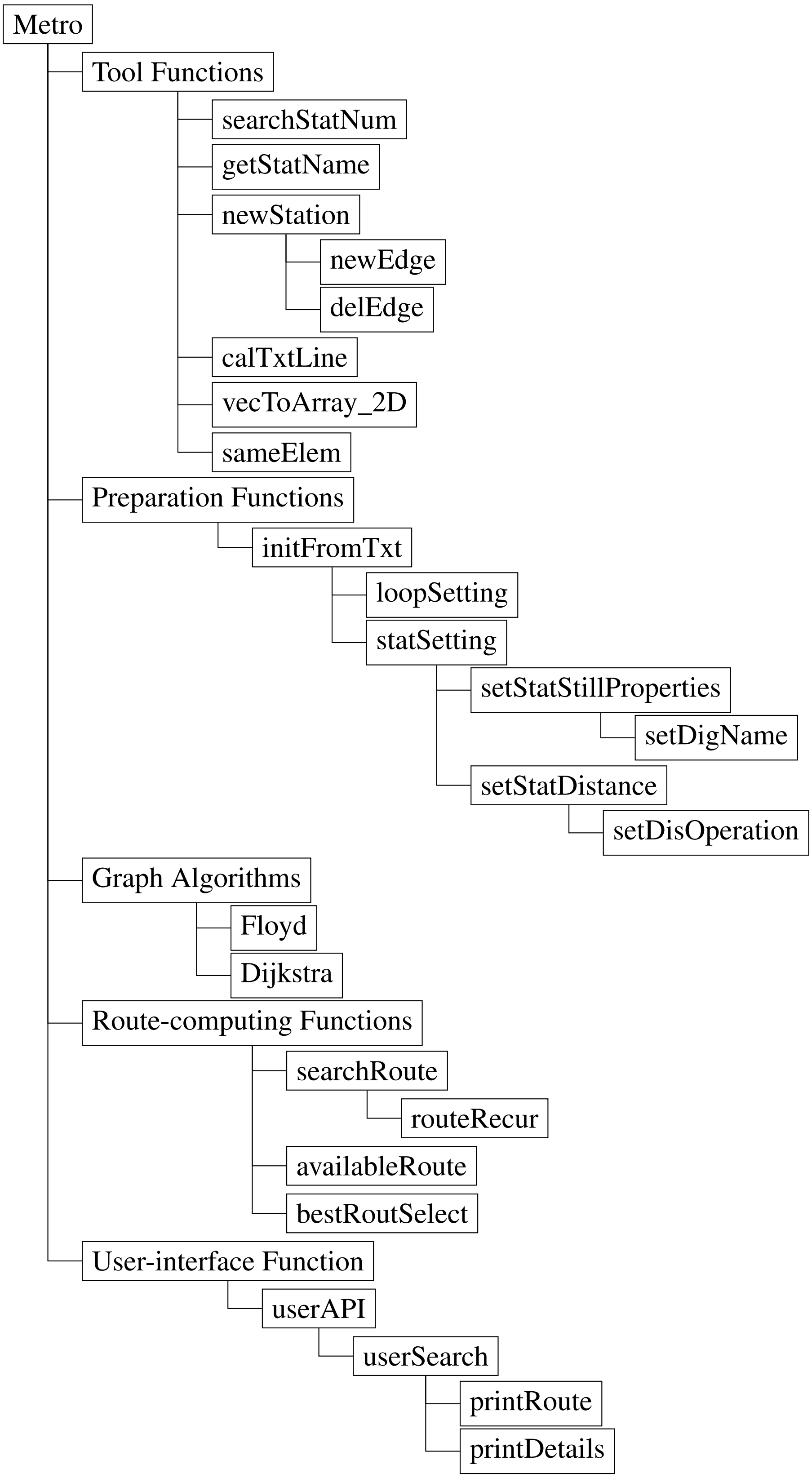
First let me introduce the design skeleton of this program. It's mainly formed by class *Metro*. Besides there's also an exception handling class *valueException*. And in *Metro*, there's two extra structs to describe a station and a route. See the tree below.



In brief, *valueException* inherits the class *logic_error*, and implements a function *printValue* to print the value of error variable. Struct *Metro::Station* and *Metro::Route* are used to describe a station or a route's properties. It should be noted that in *Metro::Station*, there're two more functions, *addNode* and *clearNodePath*. The two small functions will be useful to configure "for a specified station, which is the next station and how to reach it".

Now let's concentrate on the main class *Metro*. The main target is to know "how to get from source to destination in shortest path". When there's more than one routes, I'll choose the one with the least transfer times. Therefore I divide the program into five parts.

1. Tool Functions: used as small tools to make programming easier.
2. Preparation Functions: invoked before graph algorithms, which get data from txt file.
3. Graph Algorithms: two famous graph algorithms are provided, namely, *Floyd* and *Dijkstra*.
4. Route-computing Functions: invoked after graph algorithms to compute best route.
5. User-interface Function: a public API provided to user and several private sub-functions.


Metro
Tool Functions
searchStatNum
getStatName
newStation
newEdge
delEdge
calTxtLine
vecToArray_2D
sameElem
Preparation Functions
initFromTxt
loopSetting
statSetting
setStatStillProperties
setDigName
setStatDistance
setDisOperation
Graph Algorithms
Floyd
Dijkstra
Route-computing Functions
searchRoute
routeRecur
availableRoute
bestRoutSelect
User-interface Function
userAPI
userSearch
printRoute
printDetails

The function relationships in class *Metro* are shown in the above tree. Of course this is just an outline, and more detailed information can be found in comments in source code.

2 Highlights

2.1 Algorithms

The program selectes two famous graph algorithms to compute the shortest path: *Floyd* and *Dijkstra*. Here *Floyd* and *Dijkstra* are both provided for users to choose.

These algorithms need high efficiency when visiting data, so when computing the shortest path, vector is out of consideration. On my laptop, it'll take up to several minutes to compute shortest path if I use vector. Hence for the sake of efficiency improvement, instead I choose the basic 2-dimension arrays attached to double pointers, which is the most primitive but efficient.

In pratical application, *Dijkstra* performs much better in time than *Floyd* in single-source shortest path query.

Specially in *Dijkstra* algorithm, I use an unordered set by including *<unordered_set>*.

2.2 Exception Handling

This program applies many exception handlings for robustness. Class *valueException* is specialized for processing exceptions. Just try, throw and catch!

2.3 Data Storage

The metro system program stores data in both vectors and arrays. Detailedly, vectors are used to store data about stations and routes, namely *stat* and *rout*. But when it comes to storing the *distance* and *path* (which are necessary for graph algorithms to compute shortest path) between two stations, both vector and array are used.

In program, *origDis* and *origPath* are vectors to dynamically store the original data, and *leastDis* and *path* are double pointers indicating 2D arrays, which are intended for computing shortest path. The former two vectors will get data from the txt file, then the data will be passed to the latter two pointers by *vecToArray_2D* (transforming a 2-dimension vector to a 2-dimension array), and finally graph algorithms will compute by the latter two 2D arrays.

3 Input

3.1 Input Source of Data

Since the data of metro in a certain city is extremely complicated, I've prepared it in advance as txt files. And this program read data from the txt by <*fstream*>

Switch to another city or configure the directory of the txt file by editing *DEFAULT_SRC* in source code if you want.

3.2 Input Format

The program reads data from a formatted txt file. Pay attention to the following items.

1. Only need to fill routes in a txt file. Stations will be automatically created in data-reading process.

2. Input one route in each line of txt file.
   Format:[name of route] [if loop] [1st station name] [distance] [direction] [2nd station name] ...

   (i) [name of route] can be any string without space.

   (ii) [if loop] is a char, 'y'(yes) or 'n'(no), indicating whether or not the route is loop. When you mark a route as loop, its last station should be the same as first station.

   (iii) [distance] is a double from the former station to latter station, calculated by m(meter).

   (iv) [direction] is a char indicating whether the route is two-way or one-way.

   - 'b'(both) means two-way.
   - 'u'(up) means one-way, from the former station to latter station.
   - 'd'(down) means one-way, from the latter station to former station.

   (v) Don't forget a '\n' at the end of the whole file.

   (vi) This program supports branch route. To begin with a branch line, you just need to input the same route name and the station information from the branch point station. For example, a route A -> B -> C with branch route B -> D, then input A -> B -> C and then regard B -> D as another route which has the same name as A -> B -> C. Of course it's OK to regard it as A -> B -> D with branch route B -> C.

3. If you really want to edit routes or stations, we recommend to edit the txt file directly.

4. For more details, you can refer to the example txt data files.

4 Output

After running the executable file and following the instructions given, the program will print out the "best route".

Detailedly, the output will tell you take which route of metro and transfer at which station. For example, "Station: A -> Route: 1 -> Station: B -> Route: 3 -> Station: C" means from A you need to take Route 1 to B and transfer to Route 3 then finally reach C. And in program more detailed information is provided about the route. Try it by running!

5 Best Route

5.1 Conception

After reading the above part, you may ask what is a "best route"?

We know that between two stations there are sometimes more than 2 available metro lines/routes. Of course, after we find which stations we'll pass in a shortest path, further we'd like to know which route to take.

For instance, we know from A to D the shortest path is A -> B -> C -> D, but what we need is which route to take between A and B, B and C, C and D. There may be several routes to choose. Our mission is to determine which to take. And when we have to change route, it's called "transfer".

Let's take an example. There're 4 stations (A, B, C, D) and 5 routes(1, 2, 3, 4, 5). We have to travel from A to D, and we can only choose routes as the following.

- A to B: 1, 2
- B to C: 2, 4, 5
- C to D: 1, 2, 3, 5

Of course it's already a shortest path. No matter which route you choose, the distance is optimal, and the routes are all available. But as a passenger, we want to transfer to another route as little as possible. That is, we do NOT want to transfer too many times.

For example, if you take Route 1, Route 4, Route 1 in order, then you should transfer from Route 1 to Route 4 at Station B, from Route 4 to Route 1 at Station C. In this way, you have to transfer 2 times.

However, you can choose to take Route 2 from A straight to D without transferring! And this is so-called "best route".

5.2 How to Compute

We just need to find the route which crosses the most stations so that we transfer the least times. In other words, choose the "longest common route", like Route 2 in the above example. Of course we have to transfer sometimes, when we have to transfer, then we do.

However, sometimes even 2 or more "best routes" exist. On that ocassion, all the best routes will be computed and printed out for user.

6 Compiling & Running Environment

It's recommended to compile this program in Visual Studio 2015 and run on Windows 10. The program has been successfully compiled and run in the above environments.

7 Data Source of Subways

Data collected from:

- Beijing: Beijing Subway
- Shanghai: Shanghai Metro
- Tokyo[1]: Tokyo Metro & Toei Subway

Though I'm very eager to try metro data of other cities, it's a pity that I can't find the data of station distances on their official website. I've tried many other cities like Shanghai, Guangzhou, Hongkong, London, New York, Los Angeles, Tokyo and so on but none provides distance data. Fortunately, Wikipedia provides some of the data (but not completed). Thus I have to "make up" the other distance data. No doubt that the topological structure of metros won't be changed.

Three cities have been selected: Beijing, Shanghai and Tokyo.

As for Beijing, the loop line is worth considering. When it comes to Shanghai, it becomes more complicated. Besides loop lines, there're also branch lines in metro system of Shanghai. But Tokyo is even more peculiar. Oedo line in Tokyo metro system is in the shape of a balloon. That is, there's an extra branch line extending from a loop line.

Though the shapes of different lines are quite strange, all of them can be handled by the program. Just regard the branch line as an independent line that has the same name as the main line. In this program duplicate line names are compatible.

[1] Tokyo has two metro systems respectively managed by corporation and government.

Appendix

Finally, I'll post the source code below.

```
/*
* Author: He Yan
*/

#include<iostream>
#include<fstream>
#include<vector>
#include<string>
#include<algorithm>
#include<sstream>
#include<unordered_set>

#define INF INFINITY                    // INF means infinity
#define MAX_ROUTE_LEN 1000              // maximum number of stations in a shortest path
#define DEFAULT_SRC "Beijing.txt"
/*
* DEFAULT_SRC is the file position of metro data, by default it's configured in the same directory as executable file (.exe).
* Edit it if you want to switch to another city or configure directory.
*/

using namespace std;

static string fileSrc;      // string variable for file position of metro data
static int init_len;        // length of the array to record shortest path, used in the function Metro::routeRecur
static char alg;            // algorithm chosen, d - Dijkstra, f - Floyd, used in user API functions

// exception class
class valueException : public logic_error
{
private:
        // 4 different error data type
        char outlier_char;
        int outlier_int;
        string outlier_string;
        double outlier_double;
        // type of error data
        string type;
public:
        valueException(int value) : logic_error("This int value is out of available range! Error Value: "), outlier_int(value) { type = typeid(value).name(); }
        valueException(char value) : logic_error("This char value is out of available range! Error Value: "), outlier_char(value) { type = typeid(value).name(); }
        valueException(string value) : logic_error("This string value is out of available range! Error Value: "), outlier_string(value) { type = typeid(value).name(); }
        valueException(double value) : logic_error("This string value is out of available range! Error Value: "), outlier_double(value) { type = typeid(value).name(); }

        // print the value of error data
        void printValue()
        {
                if (type == "char") cout << outlier_char;
                else if (type == "int")cout << outlier_int;
                else if (type == "string")cout << outlier_string;
                else if (type == "double")cout << outlier_double;
        }
};

// main class
class Metro
{
private:
        // declare 2 basic structures to store data about station/route
        struct Station;
        struct Route;
        // 2 vectors to store station and route data
        vector<Station> stat;
        vector<Route> rout;

        // The following two vectors store original data about
        distance and path between two stations.
    // They are square 2D vectors!

    // origDis[i][j] means distance from i to j
    // default value: 0 when i=j, or else infinity
    vector<vector<double>> origDis;
    // origPath[i][j] means the number of the station you need to pass when moving from i to j
    // default value: i when i=j, or else -1
    vector<vector<int>> origPath;

    /*
    * Two double pointers to store data in the process of Dijkstra or Floyd algorithm and the result.
    *
    * - Why I don't choose vector or unique_ptr?
    * - Algorithms above is complicated. Therefore, when using vector to visit elements, it has a very low efficiency.
    * - Therefore double pointer is the best choice for efficiency.
    * - On my own computer, when using vector for Floyd algorithm, program crashed down after computing for several minutes.
    */
    double **leastDis;
    int **path;

    // The following are small useful tool functions.

    // search by the name of station and return the sequence number
    int searchStatNum(const string&);
    // get the name of a station by its sequence number
    string getStatName(int);
    // search by name of a station, when it exists then return sequence number, or else create a new station named this and return its number
    int newStation(const string&);
    // add a new "edge" for a square 2D vector, which means the side length of the 2D vector adds 1
    template<typename T>
    void newEdge(vector<vector<T>>&, T);
```

```
        // delete an "edge" in a square 2D vector, which means delete a certain row and col whose sequence number is the same
        template<typename T>
        void delEdge(vector<vector<T>>&, int);
        // calculate how many lines are there in a txt file
        int calTxtLine(ifstream&)const;
        // copy data from a vector to a 2-dimension array
        template<typename T>
        void vecToArray_2D(T**&, const vector<vector<T>>&)const;
        // return all the same elements between two vectors
        template<typename T>
        vector<T> sameElem(vector<T>&, vector<T>&)const;

        /*
        * The following functions read data from txt file and prepare for Floyd or Dijkstra algorithm.
        * The functions are before Floyd or Dijkstra algorithm.
        */

        /*
        * Main function: initFromTxt()
        */
        // initialize data by the txt file
        void initFromTxt()throw(valueException);
        // set a route whether it's loop
        void loopSetting(ifstream&, Route&)throw(valueException);
        // set (or create) station properties, including setting which stations are in a certain route
        void statSetting(ifstream&, Route&);
        // set still properties like name in a station (including adding the station to a certain route)
        int setStatStillProperties(const string&, Route&, int);
        // set digital name in a station, in the format of "0101", "0215" just to meet the homework's requirements
        void setDigName(Station&, string, int);
        // set distance data between two stations, including adding the data to involved stations
        void setStatDistance(char, double, Route&, int, int)throw(valueException);
        // sub-function of setStatDistance, involving origDis, origPath and stat operations
        void setDisOperation(double, Route&, int, int);

        // Algorithms for computing the shortest path, including Floyd and Dijkstra.
        void Floyd();
        void Dijkstra(int); // the int argument represents the sequence number of source place (for Dijkstra is a single-source algorithm)

        /*
        * The following functions are after Floyd or Dijkstra algorithm.
        */

        /*
        * After Dijkstra or Floyd algorithm, we get the shortest distance.
        * Now we should compute the whole path, that is, all the stations we'll pass along the shortest path.
        */
        // get the shortest path between two stations by their names, including a int variable recording the length of path
        int* searchRoute(const string&, const string&)throw(valueException);
        // sub-function of searchRoute, execute recursion process to get the whole path
        void routeRecur(int*, int);
        /*
        * After getting the whole path, now we want to compute which route to choose between two stations.
        * Compute all available routes along the shortest path.
        */
        vector<vector<string>> availableRoute(int*);
        /*
        * After getting all available routes, we want the "best" ones.
        * Now I'll get the best routes from all available routes in function bestRoutSelect.
        */
        vector<vector<string>> bestRoutSelect(vector<vector<string>>&);

```

```cpp
        /*
        * Now almost everything has been done. We've got the "best routes".
        * We just have to create an I/O interface, or we can call it an API.
        * The following are sub-functions invoked by function userAPI().
        */
        // input source place and destination place, then search and print out the best route and some more details
        void userSearch();
        // sub-function of userSearch(), which print out the best route
        void printRoute(int*, int, const vector<vector<string>>&);
        // sub-function of userSearch(), which print out details about the route
        void printDetails(int*, int);

public:
        // constructors
        Metro();
        Metro(const Metro&);
        Metro(Metro&&);
        // destructors
        ~Metro();

        // The main API function to implement user interface.
        void userAPI();
};

// a struct to hold information about a station
struct Metro::Station
{
        struct Node
        {
                string nextStat;      // from this station, which station we can directly go to (that is, adjacent stations)
                vector<string> path;  // to go to the station mentioned above, which routes we can choose
        };

        vector<Node> next;        // as described above, it tells us information about adjacent stations
        string name;              // name of this station
        vector<string> digName;   // digital names of this station, like "0101"(1st station in route 1) "0213"(13th station in route 2), just to meet the requirements of homework
        bool isTrans;             // whether it's a transfer station

        // useful function tools

        // add a new node to vector "next" if the added adjacent station doesn't exist, or else directly add a new path
        void addNode(string myPath, int num, vector<Station> &stat)
        {
                bool isExisting = false;
                string nextStat = stat[num].name;
                for (vector<Node>::iterator iter = next.begin(); iter != next.end(); ++iter)
                        if ((*iter).nextStat == nextStat)
                        {
                                isExisting = true;
                                (*iter).path.push_back(myPath);
                                break;
                        }
                if (!isExisting)
                {
                        Node tempNode;
                        tempNode.nextStat = nextStat;
                        tempNode.path.push_back(myPath);
                        next.push_back(tempNode);
                }
        }

        // clear the paths in a node
        void clearNodePath(int num, vector<Station> &stat)
        {
                string nextStat = stat[num].name;
                for (vector<Node>::iterator iter = next.begin(); iter != next.end(); ++iter)
                        if ((*iter).nextStat == nextStat)
                                (*iter).path.clear();
        }
```

```
};

// a struct to hold information about a route
struct Metro::Route
{
        string name;            // name of this route, either number or Chinese characters or something else
        vector<string> myStat;  // which stations this route passes
        bool isLoop;            // whether the route is loop or not
};

// search for a station whose name is matched with the passed-in argument
int Metro::searchStatNum(const string &name)
{
        for (vector<Station>::iterator iter = stat.begin(); iter != stat.end(); ++iter)
                if ((*iter).name == name)         // if the passed-in name argument is matched with a station's name
                        return iter - stat.begin();   // return its sequence number
        return -1;                            // or else return -1 to show "not found"
}

// get a station's name by its sequence number
string Metro::getStatName(int num)
{
        return stat[num].name;
}

// set up a new station if the name doesn't exist
// or else return the sequence number of the station whose name is matched with passed-in argument
int Metro::newStation(const string &name)
{
        int whichStat = searchStatNum(name); // here function searchStatNum is invoked

        // the following set up a new station if the name not found
        if (whichStat < 0)
        {
                Station temp;
                temp.name = name;

                // when first set up while reading data from routes in txt, it has been passed for only once
                // therefore it's not a transfer station
                temp.isTrans = false;

                stat.push_back(temp);

                // adjust origDis and origPath for the new station
                // include adding 1 to the side length of the two square 2D vectors and setting default values
                newEdge<double>(origDis, INF);
                origDis[stat.size() - 1][stat.size() - 1] = 0;
                newEdge(origPath, -1);
                origPath[stat.size() - 1][stat.size() - 1] = stat.size() - 1;

                // the sequence number of new station
                whichStat = stat.size() - 1;
        }
        // if the station already exists, it's the second time visited, thus it's a transfer station
        else stat[whichStat].isTrans = true;
        return whichStat;
}

// add 1 to the side length of a square 2D vector
template<typename T>
void Metro::newEdge(vector<vector<T>> &data, T value)
{
        for (int i = 0; i < (int)data.size(); ++i)
                data[i].push_back(value);       // the added elements have default "value"
        vector<T> temp(data.size() + 1, value);
        data.push_back(temp);
}

// delete the k row and k col of a square 2D vector
template<typename T>
void Metro::delEdge(vector<vector<T>> &data, int k)throw(
    valueException)
{
        try
        {
                int n = data.size();
                if (k < 0 || k >= n) throw valueException(k);
                // delete single elements of k col
                for (int i = n - 1; i > k; --i)
                        data[i].erase(data[i].begin() + k);
                // delete k row
                data.erase(data.begin() + k);
                // delete single elements of k col
                for (int i = k - 1; i >= 0; --i)
                        data[i].erase(data[i].begin() + k);
        }
        catch (valueException &ex) { cout << ex.what(); ex.printValue(); cout << endl; }
}

int Metro::calTxtLine(ifstream &file)const
{
        // renew file state and put file pointer to the beginning of txt
        file.clear();
        file.seekg(0, ios::beg);

        int lineNum = 0;

        while (!file.eof())
        {
                char temp[1024];          // a temp char array, just to contain the data, no significance
                file.getline(temp, 1024); // getline stop at '\n', thus we used it to calculate lines
                if (temp[0] != 0)         // ignore empty lines
                        ++lineNum;
        }

        // renew file state and put file pointer to the beginning of txt
        file.clear();
        file.seekg(0, ios::beg);

        return lineNum;
}

// here the ptr is passed in by reference
template<typename T>
void Metro::vecToArray_2D(T**& ptr, const vector<vector<T>>& vec) const
{
        int n = vec.size();
        ptr = new T*[n];
        for (int i = 0; i < n; ++i)
        {
                ptr[i] = new T[n];
                for (int j = 0; j < n; ++j)
                        ptr[i][j] = vec[i][j];  // copy elements from vector to array
        }
}

// return same elements between vector a and vector b
template<typename T>
vector<T> Metro::sameElem(vector<T> &vec1, vector<T> &vec2)const
{
        vector<T> res;
        for (vector<T>::iterator iter1 = vec1.begin(); iter1 != vec1.end(); ++iter1)
                for (vector<T>::iterator iter2 = vec2.begin(); iter2 != vec2.end(); ++iter2)
                        if (*iter1 == *iter2)
                        {
                                res.push_back(*iter1);
                                break;
                        }
        return res;
}

void Metro::initFromTxt()throw(valueException)
{
        ifstream file(fileSrc);
        if (!file)
        {
                cout << "Invalid file directory!" << endl;
                throw valueException("File Open Failed");
        }
        int lineNum = calTxtLine(file);

        // read data in each line in txt
        for (int i = 0; i < lineNum; ++i)
        {
                // record route number
                Route routTemp;
                file >> routTemp.name;
                // set whether loop
                loopSetting(file, routTemp);
                // begin to read stations, distances and directions
                statSetting(file, routTemp);
        }
        file.close();
        // copy the original vector to array to prepare for Floyd or Dijkstra algorithm
        vecToArray_2D(leastDis, origDis);
        vecToArray_2D(path, origPath);
}

// set whether a route is loop
void Metro::loopSetting(ifstream &file, Route &temp)throw(valueException)
{
        char isLoop;
        try
        {
                file >> isLoop;
                if (isLoop == 'y')
                        temp.isLoop = true;
                else if (isLoop == 'n')
                        temp.isLoop = false;
                else
                        throw valueException(isLoop);
        }
        catch (valueException &ex) { cout << ex.what(); ex.printValue(); cout << endl; }
}

```

```
void Metro::statSetting(ifstream &file, Route &temp)
{
        // isLineEnd is used to record '\n', the end of a line in txt
        // direc means direction
        char isLineEnd, direc;
        // counter records the sequence number of each station in a route, like 1 means the 1st station (departure) in a route
        // preNum and sufNum records sequence number of two adjacent stations, preNum is the former one, and sufNum is the latter
        int counter = 1, preNum, sufNum;
        // distance between two adjacent stations
        double distance;
        // like preNum and sufNum, preName and sufName mean the names of two adjacent stations
        string preName, sufName;

        // initialization of the first station in a route
        file >> preName;
        preNum = setStatStillProperties(preName, temp, counter);

        file.get(isLineEnd);
        // when the file and the line doesn't end
        while (!file.eof() && isLineEnd != '\n')
        {
                file >> distance >> direc >> sufName; // read data
                ++counter; // since it's the next station, counter adds 1

                // set station and route information
                sufNum = setStatStillProperties(sufName, temp, counter);
                setStatDistance(direc, distance, temp, preNum, sufNum);

                // the latter station will become the former one
                preNum = sufNum;
                file.get(isLineEnd);
        }
}
```

```

// mainly set station information, including adding it to the route
int Metro::setStatStillProperties(const string &name, Route &temp, int counter)
{
        // if existing, then return the sequence number, or else create a new one then return its number
        int num = newStation(name);
        // add it to the route
        temp.myStat.push_back(name);
        // set digital names in the station, like "0101" "0213", as required in the homework
        setDigName(stat[num], temp.name, counter);

        return num;
}

void Metro::setDigName(Station &myStat, string routName, int counter)
{
        // use the counter and the route name
        char digName[3] = { 0 };
        if (routName.size() == 1)
                routName.insert(0, "0");
        digName[0] = counter / 10 + '0';
        digName[1] = counter % 10 + '0';
        routName += digName;
        myStat.digName.push_back(routName);
}

// mainly set distance data between 2 stations
void Metro::setStatDistance(char direc, double distance, Route &temp, int preNum, int sufNum)throw(valueException)
{
        try
        {
                // decide according to the route diretion between 2 stations
                switch (direc)
                {
                // 'b'(both) means both directions are ok, 'u'(up) means from former to latter, 'd'(down) means from
```

```
                    latter to former
                case'b': setDisOperation(distance, temp, preNum, sufNum); setDisOperation(distance, temp, sufNum, preNum); break;
                case'u': setDisOperation(distance, temp, preNum, sufNum); break;
                case'd': setDisOperation(distance, temp, sufNum, preNum); break;
                default: throw valueException(direc);
                }
        }
        catch (valueException &ex) { cout << ex.what(); ex.printValue(); cout << endl; }
}

// detailed operation process
void Metro::setDisOperation(double distance, Route &temp, int preNum, int sufNum)
{
        // if never set distance data between the 2 stations
        if (origPath[preNum][sufNum] < 0)
        {
                origDis[preNum][sufNum] = distance;
                origPath[preNum][sufNum] = preNum;
                stat[preNum].addNode(temp.name, sufNum, stat); // add data to relevant station
        }

        // if have already set the data once, twice or more times
        // when distance is shorter than former ones
        else if (distance < origDis[preNum][sufNum])
        {
                stat[preNum].clearNodePath(sufNum, stat); // clear path data of the station
                origDis[preNum][sufNum] = distance;       // reset distance
                stat[preNum].addNode(temp.name, sufNum, stat);
        }
        else if (distance == origDis[preNum][sufNum])
                stat[preNum].addNode(temp.name, sufNum, stat); // just add data to the station
}
```

```

// Floyd algorithm to calculate the shortest path from all stations to all stations
// time cost is O(n^3)
void Metro::Floyd()
{
    int size = stat.size();
    for (int k = 0; k < size; ++k)
        for (int i = 0; i < size; ++i)
            for (int j = 0; j < size; ++j)
                if (leastDis[i][j] > leastDis[i][k] + leastDis[k][j])
                {
                    leastDis[i][j] = leastDis[i][k] + leastDis[k][j];
                    path[i][j] = k;
                }
}

// Dijkstra algorithm to calculate the shortest path from one station to all stations
// n means the sequence number of the departure station
void Metro::Dijkstra(int n)
{
    // here an unordered set is used
    unordered_set<int> des;
    // insert destinations to the unordered set
    for (int i = 0; i < (int)stat.size(); ++i)
        if (i != n)des.insert(i);

    while (des.size() > 0)
    {
        // find the minimum distance by traversing the unordered set
        double min = leastDis[n][*des.begin()];        // minimum distance
        unordered_set<int>::iterator min_iter = des.begin();    // iterator of the element which has the minimum distance
        for (unordered_set<int>::iterator iter = des.begin(); iter != des.end(); ++iter)
        {
```

```
                        if (leastDis[n][*iter] < min)
                        {
                                min = leastDis[n][*iter];
                                min_iter = iter;
                        }
                }

                int min_sub = *min_iter; // min_sub records the subscript of the element which has the minimum distance

                // remove the element from unordered set
                des.erase(min_iter);

                // update distance data of remaining elements in unordered set
                for (unordered_set<int>::iterator iter = des.begin(); iter != des.end(); ++iter)
                {
                        if (min + leastDis[min_sub][*iter] < leastDis[n][*iter])
                        {
                                leastDis[n][*iter] = min + leastDis[min_sub][*iter];
                                path[n][*iter] = min_sub;
                        }
                }
        }
}

// get the complete route (all the passed stations) along the shortest path
int* Metro::searchRoute(const string &src, const string &des)throw(valueException)
{
        int srcNum, desNum; // sequence number of the source station and destination station
        try
        {
                // get number by station's name
                srcNum = searchStatNum(src);
                desNum = searchStatNum(des);
```

```cpp

		// exception processing
		if (srcNum < 0)
			throw valueException(srcNum);
		else if (desNum < 0)
			throw valueException(desNum);
		else if (leastDis[srcNum][desNum] == INF)
		{
			cout << "Can't arrive!" << endl;
			throw valueException(INF);
		}

		// init is to store all passed stations
		int *init = new int[MAX_ROUTE_LEN];
		// at beginning we only have source station and destination station
		init[0] = srcNum, init[1] = desNum, init_len = 2;
		// compute by recursion
		routeRecur(init, 0);

		return init;
	}
	catch (valueException &ex) { cout << ex.what(); ex.printValue(); cout << endl; }
	return nullptr;
}

void Metro::routeRecur(int *init, int k) // init is to store the result, and we insert after init[k]
{
	/*
	* We choose two adjacent stations and insert a new one between them.
	* But after insertion, though the subscript of the former station won't change, the latter one will.
	* To solve this, we find that the distance from the latter one to the end of the array won't change.
	* Thus we use variable "disFromEnd" to record the position of the latter station.
	*/
	int pathNum = path[init[k]][init[k + 1]], disFromEnd = init_len - k - 1;
```

```

    // recursion stops only when the sequence number of the relay station equals the former station
    if (pathNum == init[k])
        return;

    // before insertion, move forward all the elements from the inserted position to the end of the array
    // by the way, init_len has been defined as a static global variable
    for (int i = init_len; i > k + 1; --i)
        init[i] = init[i - 1];
    // insert to init[k + 1]
    init[k + 1] = pathNum;
    // length of the array adds 1
    ++init_len;

    // continue recursion
    routeRecur(init, k);
    routeRecur(init, init_len - disFromEnd - 1);
}

// get availbale routes along the shortest path (maybe more than 1)
// shortPath is the whole shortest route (including all passed stations) that we get by function searchRoute
vector<vector<string>> Metro::availableRoute(int *shortPath)
{
    // when bestRout is null then stop
    if (shortPath == nullptr)
        return vector<vector<string>>(0);

    // posRout records the result
    vector<vector<string>> posRout;

    // preStat means the sequence number of the former station, and sufStat is the latter one
    // the total of stations is always one more than routes, thus we can initialize preStat
    // for example, from A to B we take Route 2, there're 2 stations and 1 (1 = 2 - 1) route
    int preStat = shortPath[0], sufStat;

```

```
        for (int i = 1; i < init_len; ++i)
        {
                sufStat = shortPath[i];

                // search the information of station, whose value has been assigned while reading data from txt
                // now we aim to match the next station's name
                vector<Station::Node>::iterator nextStatIter = stat[preStat].next.begin();
                while ((*nextStatIter).nextStat != stat[sufStat].name)
                        ++nextStatIter;
                // after found, then push the route information into the result vector "posRout"
                posRout.push_back((*nextStatIter).path);

                // then the latter station will become the former one
                preStat = sufStat;
        }
        return posRout;
}

// compute the "best route" by the available routes that we get from the above function
// there may be 2 or more "best routes", and they'll be completely included in result
vector<vector<string>> Metro::bestRoutSelect(vector<vector<string>> &myRout)
{
        // when empty, then stop
        if (myRout.size() == 0)
                return vector<vector<string>>(0);

        // resRout stores the result
        vector<vector<string>> resRout;
        // preRout means the former route, and sufRout means the latter one
        vector<string> preRout = *myRout.begin(), sufRout;

        resRout.push_back(preRout);

        // screen out the best routes
        for (vector<vector<string>>::iterator iter = myRout.begin() + 1; iter != myRout.end(); ++iter)
        {
                sufRout = *iter;

                // "temp" and "next" are used to record which available routes exist both in the former available routes and the latter available routes
                // here we use "temp" to store the current ones, and use "next" to store the next ones
                vector<string> temp = sameElem(preRout, sufRout), next = temp;

                // when temp is empty (no same route), then transfer, just push it in resRout
                if (temp.size() == 0)
                        resRout.push_back(sufRout);
                // when there're same routes
                else
                {
                        // resIter is a iterator of resRout
                        // we insert temp into resRout, and resIter is the inserted position
                        vector<vector<string>>::iterator resIter = resRout.insert(resRout.end(), temp) - 1;

                        // search the longest common route
                        while (next.size() != 0 && resIter >= resRout.begin() + 1)
                        {
                                temp = next;
                                next = sameElem(temp, *resIter);
                                --resIter;
                        }

                        // adjustment details
                        if (next.size() == 0)
                                resIter += 2;
                        else
                        {
                                temp = next;
                                next = sameElem(temp, *resIter);
                                if (next.size() > 0)
                                        temp = next;
                                else
                                        ++resIter;
                        }

                        // assign the longest common route to relevant members of resRout
                        for (; resIter != resRout.end(); ++resIter)
                                *resIter = temp;
                }
                // the latter member will become the former one
                preRout = sufRout;
        }
        return resRout;
}

// sub-function of userAPI, mainly for searching and printing out best route
void Metro::userSearch()throw(valueException)
{
        string src, des; // source station name and destination station name
        cout << endl << "Now input your source:" << endl; cin >> src;
        cout << endl << "Next input your destination:" << endl; cin >> des;

        // if you've chosen Dijkstra algorithm
        if (alg == 'd' && searchStatNum(src) >= 0)
                Dijkstra(searchStatNum(src));
        // search for the whole shortest path
        int *route = searchRoute(src, des);
        // if result is empty, then throw exception
        if (route == nullptr)
        {
                cout << "Illegal location!" << endl;
                throw valueException(src + " " + des);
        }

        // compute available and then best routes
        vector<vector<string>> available = availableRoute(route);
        vector<vector<string>> best = bestRoutSelect(available);
        // print result out
        printRoute(route, init_len, best);

        // some more details, including total distance, names of passed stations and their distance
        cout << endl << endl << "Would like to see all details about passing stations? (y/n)" << endl;
        char flag; cin >> flag;
        try
        {
                if (flag == 'y')
                        // print out details
                        printDetails(route, init_len);
                else if (flag != 'n')
                        throw valueException(flag);
        }
        catch (valueException &ex) { cout << ex.what(); ex.printValue(); cout << endl; }
}

// sub-function of userSearch, used to print routes
void Metro::printRoute(int *route, int routeLen, const vector<vector<string>> &best)
{
        int cursorStat = 1; // a cursor used to print station and route (cursorStat - 1)
        cout << endl << "The best route is:" << endl;
        cout << "Station: " << stat[route[0]].name;

        while (cursorStat < routeLen)
        {
                // search for transfer station
                while (cursorStat < routeLen - 1 && best[cursorStat - 1] == best[cursorStat])
                        ++cursorStat;

                // then print out name of route and transfer station
                cout << " -> Route: " << best[cursorStat - 1][0];
                for (int temp = 1; temp < (int)best[cursorStat - 1].size(); ++temp)
```

```
                        cout << " or " << best[cursorStat - 1][temp]; // there may be more than one best route
                cout << " -> Station: " << stat[route[cursorStat]].name;

                ++cursorStat;
        }
}

// sub-function of userSearch, used to print details
void Metro::printDetails(int *route, int routeLen)
{
        // total distance from source to destination
        double dis = leastDis[route[0]][route[routeLen - 1]];

        // print out total distance
        cout << endl << "Total distance: (calculated by m)" << endl;
        cout << dis << endl;

        // print out names of passed stations
        cout << endl << "Names of passing stations:" << endl;
        for_each(route, route + routeLen - 1, [&](int x) {cout << getStatName(x) << " -> "; });
        cout << getStatName(route[routeLen - 1]) << endl;

        // print out distance between two adjacent stations in the above passed stations
        cout << endl << "Distance of passing routes: (calculated by m)" << endl;
        for (int i = 0; i < routeLen - 2; ++i)cout << leastDis[route[i]][route[i + 1]] << " -> ";
        cout << leastDis[route[routeLen - 2]][route[routeLen - 1]] << endl;
}

// constructors
// vectors have built-in copy/move constructors and destructors
Metro::Metro()
{
        leastDis = nullptr;
        path = nullptr;
```

```
}

Metro::Metro(const Metro &a) : stat(a.stat), rout(a.rout), origDis(a
    .origDis), origPath(a.origPath)
{
        int size = origDis.size();
        leastDis = new double*[size];
        path = new int*[size];
        for (int i = 0; i < size; ++i)
        {
                leastDis[i] = new double[size];
                path[i] = new int[size];
                for (int j = 0; j < size; ++j)
                {
                        leastDis[i][j] = a.leastDis[i][j];
                        path[i][j] = a.path[i][j];
                }
        }
}

Metro::Metro(Metro &&a) : stat(a.stat), rout(a.rout), origDis(a.
    origDis), origPath(a.origPath)
{
        leastDis = a.leastDis;
        path = a.path;
        a.leastDis = nullptr;
        a.path = nullptr;
}

Metro::~Metro()
{
        int size = origDis.size();
        for (int i = 0; i < size; ++i)
        {
                delete[] leastDis[i];
                delete[] path[i];
        }
        delete[] leastDis;
        delete[] path;
        leastDis = nullptr;
        path = nullptr;
}
```

```

// user API function
void Metro::userAPI()
{
	// instructions
	cout << "Welcome to Metro Route System!" << endl << "Our system helps to calculate the best route from source to destination." << endl;
	cout << endl << "Commands list:" << endl;
	cout << "search - Search for best route between two stations." << endl;
	cout << "exit - Leave the Metro Route System." << endl;

	// read data from txt
	initFromTxt();

	// choose your algorithm
	cout << endl << "Choose algorithm: Dijkstra or Floyd? (d/f)" << endl; cin >> alg;
	try
	{
		if (alg == 'f')
		{
			// Floyd algorithm takes more time to initialize, thus you may have to wait for a while
			cout << endl << "Loading... Please wait for a few seconds." << endl;
			Floyd();
			cout << endl << "Loading Floyd algorithm complete." << endl;
		}
		else if (alg != 'd')
		{
			alg = 'd'; // default: dijkstra
			throw valueException(alg);
		}
	}
	catch (valueException &ex) { cout << ex.what(); ex.printValue(); cout << endl; }

	// input your command
```

```cpp
        string command;
        cout << endl << "Your command:" << endl; cin >> command;

        // command processing
        while (command != "exit")
        {
                if (command == "search")
                {
                        try { userSearch(); }
                        catch (valueException &ex) { cout << ex.what(); ex.printValue(); cout << endl; }
                }
                else
                        cout << endl << "Invalid command." << endl;

                // input command again until you input "exit"
                cout << endl << "Your command:" << endl; cin >> command;
        }
}

int main()
{
        fileSrc = DEFAULT_SRC; // set txt file source
        Metro sample;
        sample.userAPI();
        return 0;
}
```