Rafał Kubas

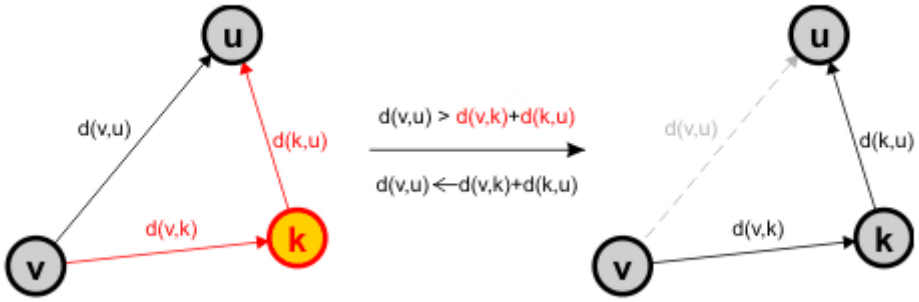
:

1.Task topic

Route planner. There are a lot of bus stops and bus lines in a city. Each bus line has its own time schedule. Write a program that finds the fastest route between two given bus stops. Details to be discussed.

2.Project analysis

Main assumption was using Floyd Warshall algorithm in our problem. It's used for distances but when we take hours into account instead of distances, it works similarly. Of course possible route won't be such great as it would be decided by program with artificial inteligence or at least more advanced algorithms for distances. Sometimes it can show the way which can shorter (sum of hours is less), but there can occur situation where passenger has to wait long time (it depends on complexity of input file)
Whole bus stop plans are included in 1 txt file, from which I had to take data, put them into arrays (2d/1d/double/string) correctly. When we've got fullfilled arrays with possible connections and values of it next step was using Floyd Warshall algorithm. In shortest way, it is based on this easy statement. In our case, distances are replaced with hours (double).



If distance from A->C than A->B+ B->C then we assume that A->C=A->B+B->C. And we will perform it on whole matrix.

$$D[i,j]^{(k)} = \begin{cases} T[i,j], & \textbf{gdy } \mathbf{k = 0} \\ \min(D[i,j]^{(k-1)}, D[i,k]^{(k-1)} + D[k,j]^{(k-1)}), & \textbf{gdy } \mathbf{k \geq 1} \end{cases}$$

That's the solution ('gdy' means 'when') of Floyd Warshall algorithm. Now time for more specified description of this algorithm skip to next point if you're not interested)

In computer science, the Floyd–Warshall algorithm (also known as Floyd's algorithm, Roy–Warshall algorithm, Roy–Floyd algorithm, or the WFI algorithm) is a graph analysis algorithm for finding shortest paths in a weighted graph with positive or negative edge weights (but with no negative cycles, see below) and also for finding transitive closure of a relation R. A single execution of the algorithm will find the lengths (summed weights) of the shortest paths between all pairs of vertices, though it does not return details of the paths themselves.

The Floyd–Warshall algorithm was published in its currently recognized form by Robert Floyd in 1962. However, it is essentially the same as algorithms previously published by Bernard Roy in 1959 and also by Stephen Warshall in 1962 for finding the transitive closure of a graph.[1] The modern formulation of Warshall's algorithm as three nested for-loops was first described by Peter Ingerman, also in 1962. The Floyd–Warshall algorithm compares all possible paths through the graph between each pair of vertices. It is able to do this with Θ(|V |3) comparisons in a graph. This is remarkable considering that there may be up to Ω(|V |2) edges in the graph, and every combination of edges is tested. It does so by incrementally improving an estimate on the shortest path between two vertices, until the estimate is optimal.

Consider a graph G with vertices V numbered 1 through N. Further consider a function shortestPath(i, j, k) that returns the shortest possible path from i to j using vertices only from the set {1,2,...,k} as intermediate points along the way. Now, given this function, our goal is to find the shortest path from each i to each j using only vertices 1 to k + 1.

For each of these pairs of vertices, the true shortest path could be either (1) a path that only uses vertices in the set {1, ..., k} or (2) a path that goes from i to k + 1 and then from k + 1 to j. We know that the best path from i to j that only uses vertices 1 through k is defined by shortestPath(i, j, k), and it is clear that if there were a better path from i to k + 1 to j, then the length of this path would be the concatenation of the shortest path from i to k + 1 (using vertices in {1, ..., k}) and the shortest path from k + 1 to j (also using vertices in {1, ..., k}).

If w(i, j) is the weight of the edge between vertices i and j, we can define shortestPath(i, j, k + 1) in terms of the following recursive formula: the base case is

 {shortestPath}(i, j, 0) = w(i, j)
and the recursive case is

shortestPath}(i,j,k+1) = min{shortestPath}(i,j,k),\, {shortestPath}(i,k+1,k) + {shortestPath}(k+1,j,k))
This formula is the heart of the Floyd–Warshall algorithm. The algorithm works by first computing shortestPath(i, j, k) for all (i, j) pairs for k = 1, then k = 2, etc. This process continues until k = n, and we have found the shortest path for all (i, j) pairs using any intermediate vertices.

The algorithm is an example of dynamic programming.
Source: https://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm
Picture source: http://eduinf.waw.pl/inf/alg/001_search/0138b.php

3.External specification

To run the program there is required file timetable.txt (note). It contains bus stop plans seperated from each other with '*'. First line is name of bus stop, and then there are listed possible connections (other bus stops) and time of arrival. Examplary bus plan (Names don't have to be 1 character, there are string arrays for them):

```
Market
Blacksmith 11.20
Church 10.40
Post 11.30
*
Post
Church 13.11
Blacksmith 15.2
```

*
Don't forget about '*' at the end of file.

4.Internal specification

First of all, almost everything is commented in code, but let's remind important ones.

Class:

```
class full_plan //class containing all basic informations
about bus stops, (2 tables for path and for value(Floyd
algotrithm)
1.{
2.    private:
3.    int fconnections;
4.    public:
5.    double **ftab_value;
6.    int **ftab_destination;
7.    string *fncities; //names of cities
8.        full_plan(int fsize) //constructor
9.        {
10.        fncities= new string[fsize];
11.        string *ncities; //names of cities
12.        ftab_value=new double *[fsize];
13.        ftab_destination=new int *[fsize];
14.        for(int i=0; i<fsize; i++) //creating dynamic
arrays with constructor
15.            {ftab_value[i] = new double[fsize];
16.            ftab_destination[i] = new int[fsize];
17.            }
18.        fconnections=fsize;
19.        }
20.      ~full_plan() //destructor
21.        {
22.            delete[]fncities;
23.            for(int i=0; i<fconnections; i++)
24.                {delete[] ftab_value[i];
25.                delete[] ftab_destination[i];
26.                }
27.                delete[]ftab_destination;
28.                delete[]ftab_value;
29.}
30.void filling(int tsize, string *basic, string
*destination); //class methods
31.void tabp_fill(int tsize, int **tab); //initial path
filling
32.void tabv_fill(int tsize, double **tab); //initial value
filling
```

```
33.void drawptab(int tsize, int **tab); //methods used in
testing- shows 2d arrays
34.void drawvtab(int tsize, double **tab); //similar but for
values
35.};
```
Functions, procedures, methods (whole in source code)

```
int counter(int tsize, string *names) //function counting
duplicates

void cleaners(int tsize, string *names) //procedure cleaning
array of string

void cleanerd(int tsize, double *tab) //procedure cleaning
double array

void repeat(int tsize, string *names, string *fncities)
//procedure detecting and deleting duplicates in bus stop
connections

void pathfill(int tsize, string *dictionary, string first,
string input, int **tab) //filling path array with possible
connections


void valuefill(int tsize, string *dictionary, string first,
string input, double source, double **tab) //procedure fillin'
array with values (in our case hours)


void full_plan::filling(int tsize, string *basic, string
*destination)// putting names of bus stops into object

void full_plan::tabv_fill(int tsize, double **tab) //filling
value 2d array with initial data

void full_plan::tabp_fill(int tsize, int **tab)

void full_plan::drawptab(int tsize, int **tab) //drawing path
table

void full_plan::drawvtab(int tsize, double **tab) //drawing
value table, both used by me to see if it works correctly
```

```
void previouspath(int i, int j, int **tab, string *base,
double **tabv) // procedure in floyd warshall algoritm to
investigate whole needed
//combination of bus stops using path array 2d
void copying(int tsize, double **tab1, double **tab2)
//procedure of copying from 1 double array to the different
one
Variable & dynamic arrays
//string used in program
        string buff_line;
        string tkn;
        string star="*"; //sign of ending bus stop list
        string buff;
        // variables used in program, most of them used
operation on file, to obtain data properly
        double buffor=0;
        int linenumber=0; //number of lines
        int busnumb=0; //number of bus stops
        int divider=0; //first kind of variable which
segregates if it's string or double
        int add_divider=0; //used to put values into array
properly (without empty cells)
        int connections=0; //number of connections for 1 bus
stop
        int cnt_tabd=0; //used to put bus stop names into
array properly (without empty cells)
        int indp_bustp=0; //number of all possible bus stops
(without duplicates)
        int q=0; //occurs in loop only

string *namestab= new string[linenumber*10]; //array with city
names
        string *namestab_rep= new string[linenumber*10];
//array with city names after deleting duplicates
        double *hourtab= new double[linenumber*10]; //array of
hours per connections
        double *fhourtab= new double[linenumber*10]; //full
array of hours

double **copy_vtab=new double*[indp_bustp]; //copying value to
other array to save initial positions
        for(int i=0; i<indp_bustp; i++)
            copy_vtab[i]= new double[indp_bustp];
```

Mainly used in implementation of data from txt file into array of class, or just buffor arrays

5.Source Code

```
#include <iostream>
1.#include <string.h>
2.#include <string>
3.#include <stdio.h>
4.#include <sstream>
5.#include <stdlib.h>
6.#include <fstream>
7.#include <cstdlib>
8.#include <cstdio>
9.#include <ctime>
10.
11.
12.using namespace std;
13.
14.class full_plan //class containing all basic informations about bus stops, (2 tables for path and for value(Floyd algotrithm)
15.{
16.    private:
17.    int fconnections;
18.    public:
19.    double **ftab_value;
20.    int **ftab_destination;
21.    string *fncities; //names of cities
22.        full_plan(int fsize) //constructor
23.        {
24.        fncities= new string[fsize];
25.        string *ncities; //names of cities
26.        ftab_value=new double *[fsize];
27.        ftab_destination=new int *[fsize];
28.        for(int i=0; i<fsize; i++) //creating dynamic arrays with constructor
29.            {ftab_value[i] = new double[fsize];
30.            ftab_destination[i] = new int[fsize];
31.            }
32.        fconnections=fsize;
33.        }
34.         ~full_plan() //destructor
35.        {
36.            delete[]fncities;
37.            for(int i=0; i<fconnections; i++)
38.                {delete[] ftab_value[i];
39.                delete[] ftab_destination[i];
```

```
40.             }
41.             delete[]ftab_destination;
42.             delete[]ftab_value;
43.}
44.void filling(int tsize, string *basic, string *destination); //class methods
45.void tabp_fill(int tsize, int **tab); //initial path filling
46.void tabv_fill(int tsize, double **tab); //initial value filling
47.void drawptab(int tsize, int **tab); //methods used in testing- shows 2d arrays
48.void drawvtab(int tsize, double **tab); //similar but for values
49.};
50.
51.int counter(int tsize, string *names) //function counting duplicates
52.{
53.       int j=0;
54.       for(int i=0; i<tsize; i++)
55.       {
56.          if(names[i]==names[i+1])
57.          {
58.             j++;
59.          }
60.
61.       }
62.    return j;
63.    }
64.void cleaners(int tsize, string *names) //procedure cleaning array of string
65.{
66.    for(int i=0; i<tsize; i++)
67.       names[i]="";
68.}
69.void cleanerd(int tsize, double *tab) //procedure cleaning double array
70.{
71.    for(int i=0; i<tsize; i++)
72.       tab[i]=0;
73.}
74.
75.void repeat(int tsize, string *names, string *fncities) //procedure detecting
and deleting duplicates in bus stop connections
76.    {
77.       int j=0;
78.       for(int i=0; i<tsize; i++)
79.       {
80.          if(names[i]!=names[i+1])
81.          {
82.             fncities[i-j]=names[i];
83.          }
```

```
84.           else
85.           {
86.           j++;
87.           }
88.
89.       }
90.
91.   }
92.void pathfill(int tsize, string *dictionary, string first, string input, int
**tab) //filling path array with possible connections
93.{
94.    int k=0;
95.    for(int i=0; i<tsize; i++)
96.    {
97.     if(first==dictionary[i])
98.     {
99.        k=i; //saving index of position
100.     }
101.    }
102.    for(int j=0; j<tsize; j++)
103.    {
104.           if(input==dictionary[j])
105.           {
106.              tab[k][j]=k; //comes from floyd warshall algorithm (path table
section)
107.
108.           }
109.    }
110.
111.
112.}
113.
114.void valuefill(int tsize, string *dictionary, string first, string input, double
source, double **tab) //procedure fillin' array with values (in our case hours)
115.{
116.     int k=0;
117.     int x=0;
118.    for(int i=0; i<tsize; i++)
119.    {
120.     if(first==dictionary[i])
121.     {
122.        k=i;
123.     }
124.    }
125.
```

```
126.    for(int j=0; j<tsize; j++)
127.    {
128.            if(input==dictionary[j])
129.            {
130.               tab[k][j]=source;
131.               x++;
132.
133.            }
134.    }
135.}
136.
137.void full_plan::filling(int tsize, string *basic, string *destination)// putting
names of bus stops into object
138.    {
139.        for(int i=0; i<tsize;i++)
140.            {
141.               destination[i]=basic[i];
142.
143.            }
144.    }
145.void full_plan::tabv_fill(int tsize, double **tab) //filling value 2d array with
initial data
146.{
147.
148.    for(int i=0; i<tsize; i++)
149.    {
150.        for(int j=0; j<tsize; j++)
151.            {
152.               if(i==j)
153.               {
154.                  tab[i][j]=0;
155.               }
156.               else
157.               {
158.                  tab[i][j]=UINT_MAX; //infinity representation
159.
160.               }
161.            }
162.    }
163.}
164.void full_plan::tabp_fill(int tsize, int **tab)
165.{
166.for(int i=0; i<tsize; i++)
167.    {
168.        for(int j=0; j<tsize; j++)
```

```
169.          {
170.
171.             tab[i][j]=-1;
172.          }
173.    }
174.}
175.void full_plan::drawptab(int tsize, int **tab) //drawing path table
176.{
177.    for(int i=0; i<tsize; i++)
178.    {
179.       for(int j=0; j<tsize; j++)
180.          {
181.
182.             cout<<tab[i][j]<<"  ";
183.          }
184.       cout<<'\n';
185.    }
186.}
187.void full_plan::drawvtab(int tsize, double **tab) //drawing value table, both
used by me to see if it works correctly
188.{
189.    for(int i=0; i<tsize; i++)
190.    {
191.       for(int j=0; j<tsize; j++)
192.          {
193.             if(tab[i][j]==UINT_MAX)
194.                cout<<"#"<<"  ";
195.             else
196.                cout<<tab[i][j]<<"  ";
197.          }
198.       cout<<'\n';
199.    }
200.}
201.void previouspath(int i, int j, int **tab, string *base, double **tabv) //
procedure in floyd warshall algoritm to investigate whole needed
202.//combination of bus stops using path array 2d
203.{
204.  if(i == j) cout <<base[i]<<" ";
205.  else if(tab[i][j] == -1) cout << "NO PATH";
206.  else
207.  {
208.    previouspath(i,tab[i][j], tab, base, tabv); //recursion, calling same
procedure with different parameters by itself
209.cout << " -("<<tabv[tab[i][j]][j]<<")-> ";
210.cout << base[j];
```

```
211.  }
212.}
213.
214.void copying(int tsize, double **tab1, double **tab2) //procedure of copying from 1 double array to the different one
215.{
216.    for(int i=0; i<tsize; i++)
217.        for(int j=0; j<tsize; j++)
218.    {
219.        tab1[i][j]=tab2[i][j];
220.    }
221.}
222.
223.
224.int main()
225.{       //FILE OPERATORS
226.        fstream timetab; //creating steam
227.        timetab.open("timetable.txt"); //opening source file
228.
229.        //string used in program
230.        string buff_line;
231.        string tkn;
232.        string star="*"; //sign of ending bus stop list
233.        string buff;
234.        // variables used in program, most of them used operation on file, to obtain data properly
235.        double buffor=0;
236.        int linenumber=0; //number of lines
237.        int busnumb=0; //number of bus stops
238.        int divider=0; //first kind of variable which segregates if it's string or double
239.        int add_divider=0; //used to put values into array properly (without empty cells)
240.        int connections=0; //number of connections for 1 bus stop
241.        int cnt_tabd=0; //used to put bus stop names into array properly (without empty cells)
242.        int indp_bustp=0; //number of all possible bus stops (without duplicates)
243.        int q=0; //occurs in loop only
244.
245.
246.        bool guard=true; //guard used for recognising main bus stop (on which connections are based)
247.
248.
```

```
249.
250.
251.
252.        if(timetab.good()) //if source file was opened correctly
253.    {
254.
255.
256.while(!timetab.eof()) //till it's not end of file
257.{
258.            getline(timetab, buff_line); //take line from a file and put it into string
buffor
259.            linenumber++; //increase number of lines
260.            istringstream iss(buff_line); //putting into buffor
261.             if (!buff_line.compare(star)) //if there is star
262.                  {
263.                     busnumb++; //number of bus stops
264.                  }
265.}
266.
267.
268.
269.            timetab.seekg(0); //refreshing stream
270.
271.
272.        string *namestab= new string[linenumber*10]; //array with city names
273.        string *namestab_rep= new string[linenumber*10]; //array with city
names after deleting duplicates
274.        double *hourtab= new double[linenumber*10]; //array of hours per
connections
275.        double *fhourtab= new double[linenumber*10]; //full array of hours
276.        while(!timetab.eof())
277.        { //whole procedure is based on dividing strings from double variables
using simple properties of odd/even variable divider
278.                getline(timetab, buff_line);
279.                  if (buff_line.compare(star))
280.                  {
281.                     connections++; //increase connections if there is star in txt file
282.                     istringstream iss(buff_line);
283.                        while(getline(iss, tkn, ' ')) //dividing into tokens
284.                        {
285.                           if(guard)
286.                           {
287.                           namestab[cnt_tabd]=tkn;
288.                           cnt_tabd++;
289.                           guard=false;
```

```
290.                     }
291.                     else
292.                     {
293.                        if(divider%2)
294.                        {
295.                           buffor=atof(tkn.c_str());
296.                           fhourtab[add_divider]=buffor;
297.                           add_divider++;
298.                           divider++;
299.                        }
300.                        else
301.                        {
302.                           divider++;
303.                           namestab[cnt_tabd]=tkn;
304.                           cnt_tabd++;
305.                        }
306.                     }
307.
308.                  }
309.
310.               }
311.               else
312.               {
313.                  guard=true;
314.                 // divider=0;
315.                  //add_divider=0;
316.                  //connections=0;
317.               }
318.         };
319.
320.full_plan base(connections); //creating object of class
321.//SORTING INPUT
322.connections=0;
323.divider=0;
324.add_divider=0;
325.cnt_tabd=0;
326.//sorting bus stops names
327.for(int i=0; i<linenumber-busnumb; i++)
328.{
329. for(int j=0; j<linenumber-busnumb; j++)
330.         {
331.            if(namestab[i]<namestab[j])
332.               {
333.                  buff_line= namestab[j];
334.                  namestab[j]= namestab[i];
```

```
335.                    namestab[i]= buff_line;
336.
337.                }
338.          }
339.          }
340.          indp_bustp=linenumber-(busnumb+counter(linenumber-busnumb,
namestab)); //number of whole bus stops
341.//works
342.          repeat(linenumber-busnumb, namestab, namestab_rep); //deleting
duplicates
343.          base.filling(linenumber-busnumb, namestab_rep,
base.fncities);//works- putting names of bus stops into object
344.          base.tabp_fill(indp_bustp, base.ftab_destination);//filling path array
345.          base.tabv_fill(indp_bustp, base.ftab_value); //filling value array
346.         // base.drawptab(indp_bustp, base.ftab_destination);
347.        // base.drawvtab(indp_bustp, base.ftab_value);
348.
349.
350.
351.
352.
353.
354.timetab.seekg(0);
355.while(!timetab.eof())
356.      {
357.            getline(timetab, buff_line);
358.              if (buff_line.compare(star))
359.              {
360.                 connections++;
361.                 istringstream iss(buff_line);
362.                    while(getline(iss, tkn, ' '))
363.                    {
364.                       if(guard)
365.                       {
366.                       namestab[cnt_tabd]=tkn;
367.                       cnt_tabd++;
368.                       guard=false;
369.                       }
370.                       else
371.                       {
372.                          if(divider%2)
373.                          {
374.                             buffor=atof(tkn.c_str());
375.                             hourtab[add_divider]=buffor;
376.                             add_divider++;
```

```
                        divider++;

                    }
                    else
                    {
                        divider++;
                        namestab[cnt_tabd]=tkn;
                        cnt_tabd++;

                    }



                }

            }

        }
        else
        {
            guard=true;
            for(int i=1; i<connections; i++)
            {
                pathfill(indp_bustp, base.fncities, namestab[0], namestab[i],
base.ftab_destination); //filling path array
                valuefill(indp_bustp, base.fncities, namestab[0],
namestab[i], hourtab[q], base.ftab_value); //filling value array
                q++;
            }
            divider=0;
            cnt_tabd=0;
            add_divider=0;
            connections=0;
            q=0;
        }

    };
//base.drawptab(indp_bustp, base.ftab_destination);
//base.drawvtab(indp_bustp, base.ftab_value);
    double **copy_vtab=new double*[indp_bustp]; //copying value to other
array to save initial positions
    for(int i=0; i<indp_bustp; i++)
        copy_vtab[i]= new double[indp_bustp];
    copying(indp_bustp, copy_vtab, base.ftab_value);//copying from value
```

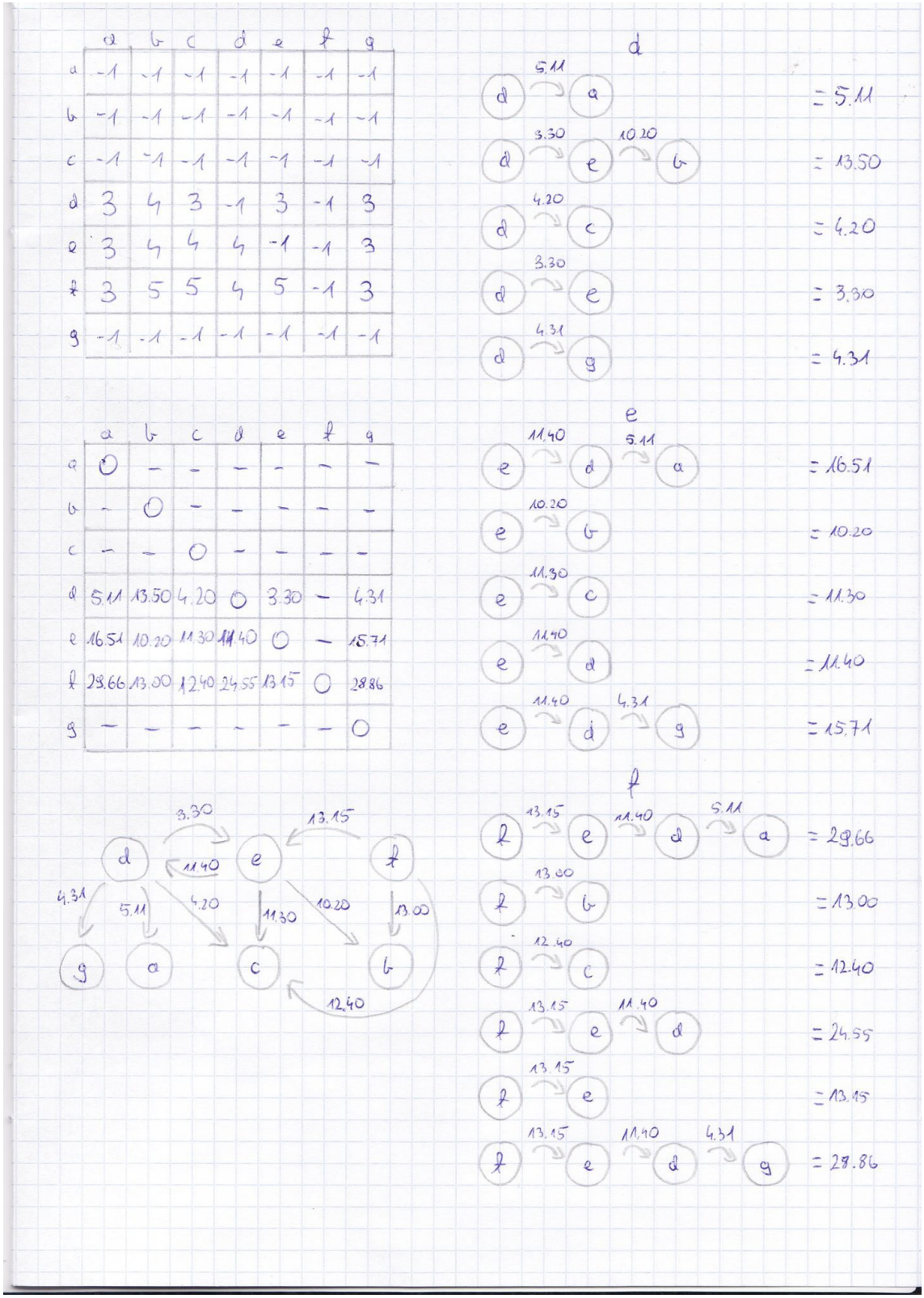
```
array to our new made one
//Floyd_warshall procedure
for(int k=0; k<indp_bustp; k++)
{
   for(int i=0; i<indp_bustp; i++)
   {
      for(int j=0; j<indp_bustp; j++)
      {
         if(base.ftab_value[i][k]==UINT_MAX || base.ftab_value[k]
[j]==UINT_MAX)
            continue;
         if(base.ftab_value[i][j]>base.ftab_value[i][k]+base.ftab_value[k][j])
         {
            base.ftab_value[i][j]=base.ftab_value[i][k]+base.ftab_value[k][j];
            base.ftab_destination[i][j]=base.ftab_destination[k][j];
         }
      }
   }
}


//If someones wants to see how it works
//base.drawptab(indp_bustp, base.ftab_destination);

//base.drawvtab(indp_bustp, base.ftab_value);

   //User initial interface
   string user_bus_start;
   string user_bus_finish;
   cout<<"Enter your bus stop"<<'\n';
   cin>>user_bus_start;
   cout<<"Enter your finish bus stop"<<'\n';
   cin>>user_bus_finish;
   int xstart=0;
   int xfinish=0;

   for(int i=0; i<indp_bustp; i++) //getting bus stops names
   {
      if(base.fncities[i]==user_bus_start)
      {
         xstart=i;
      }
      if(base.fncities[i]==user_bus_finish)
      {
         xfinish=i;
```

```
462.      }
463.   }
464.      previouspath(xstart, xfinish, base.ftab_destination, base.fncities,
copy_vtab); //recursion algorithm for path directions
465.   }
466.}
467.
```

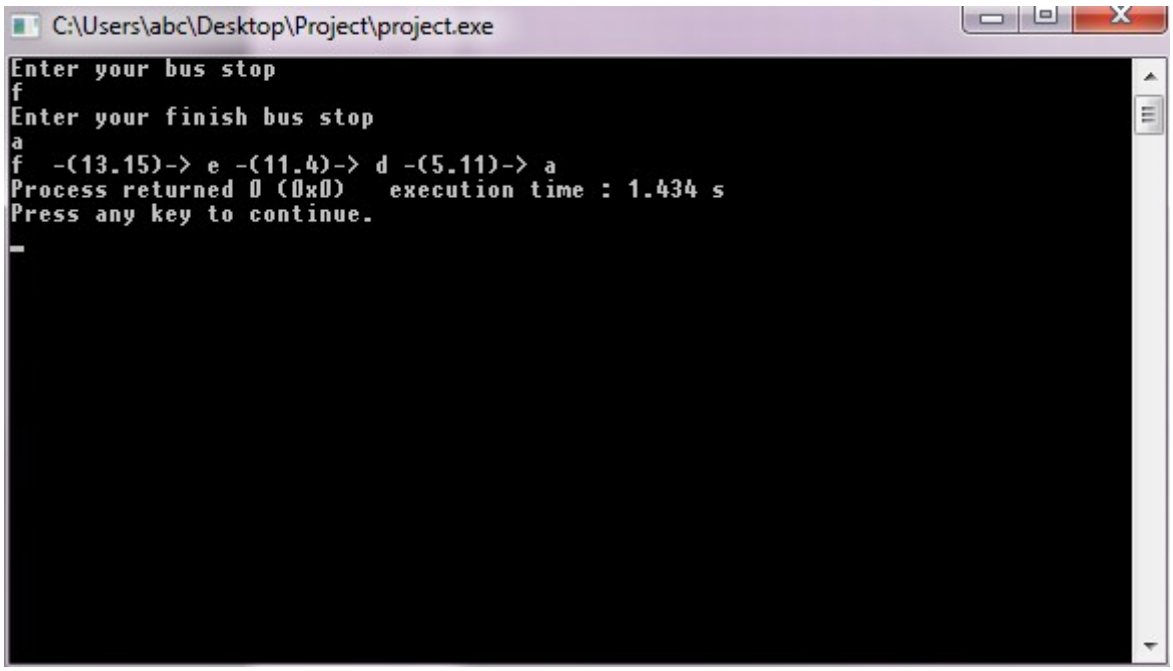
6.Testing

Main testing example is this one included already in timetable. There are used only letters, because I wanted to save some time during changes. But it's possible to contain whole words.

I will include here written graph for our example, and then shows output in console window.



And our examplary output from f bus stop to a bus stop



As we can see it works corectly.

7.Conclusions

Program is completed, problem is, that there is only one possible connection from bus stop per day in direction. Also interface could be more friendly, with shown possibilites, or just writing all posibilites, it would slow down program that's why I didnt include it.