Rafał Kubas

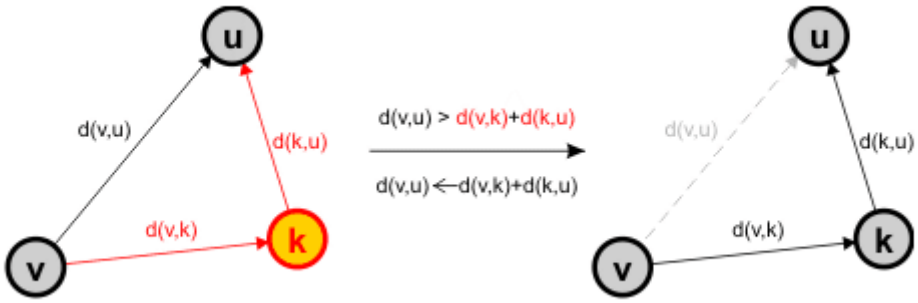
1. Task topic

Shortest path Write a program that calculates the shortest route between two cities. Cities are connected by one-way roads with a length. Road map file is as follows: [origin city] [final city] [distance] The program should ask the user for origin and destination cities, and then output the shortest possible route, which is described by its total length and the lengths of the individual segments (together with their origins and destinations).

2. Project analysis

First I had to move data from file into arrays or something to be able to operate on this. So I put City names into array, whole combinations into array. After fullfiling 2 matrices (matrices were required because it is looking distance between two elements) for with 0 or infinity (MAX_int) I had to put some data into it. So with sscanf I was dividing combination string and filling properly place in matrix. There was nothing special about getting data and putting it into arrays, operations on strings and on arrays were needed.

Then I used Floyd Warshall algorithm. In shortest way, it is based on this easy statement.



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.

3. External specification

To run the program there is required file city.txt (note), in this file first row should be just number of cities declared(let's call it x). Then in next x rows, there should be name of cities (Something like list, but instead of space press enter). Rest of files should be (row by row) combinations of city distances. For example:
City1 City2 30
It means that distance between City1 and City2 equals 30. That's all about required file. Now that's time for using console. Just follow tips, and answer questions correctly. (Give exactly same name of city1, city2, and that's all. You can also use other cities if you answer „Yes” after question if you want to try other cities) Program returns distance between them, name of next city on the move and distance to it (something like really poorly made GPS).

4. Internal specification

First of all, I had to use dynamic arrays and I needed more than one:

*FloydTable – 2D int array where I performed whole Floyd Warshall algorithm

*PathTable- 2D int array where I performed path between cities (next move)

* dynCNames – 2D char array where I was storing city names
*dynCDist – 2D char array where I was storing distances between A and B, where A, B are cities names.
I used one function (technically two, but the second one was self-calling function to imitate whole path and it doesn't work)
*LineCounter – function that returns number of lines in file (very important to properly put distances into arrays) This function was based on fgetc to get character by character till it's \n
Important parts of code

```
*while (fgets(line, line_size, fp) != NULL) – loop condition(put line from file into buffer (line is the buffer)and it is not equal NULL);
*strcpy(dynCNames[counter - 2], line) – strcpy is used to copy whole string, from line (buffer) to dynCNames with specified number (lowered counter to fit perfectly)
*sscanf(dynCDist[k], "%s %s %d", FirstCChecker, SecondCChecker, &DistanceValue); - Reads formatted input from the string
*strncmp(FirstCChecker, dynCNames[i], strlen(FirstCChecker))- strncmp is comparison of two string, in this case buffer FirstCChecker and dynCNames up to x signs, where x is lenght of FirstCChecker (strlen)
*memset(FirstCChecker, 0, strlen(FirstCChecker)) – memset sets the first num bytes of the block of memory pointed by ptr to the specified value (interpreted as an unsigned char).
*char *statement = (char *)malloc(sizeof(char)) – allocating memory for pointer;
*Also I used variable counter, which can be weird but actually works to divide strings from line into two groups (those which are cities names, and combination with distances)
That's all 'difficult' functions in code, rest will be in source code part with comments
```

5. Source Code

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#pragma warning(disable:4996)  //to avoid visual studio problems about unsafe functions

int **PathTable; //Global declared because is needed in FWPath procedure

int LineCounter(FILE *Filename, int NLines) //As name says, this is
```

```
    function for counting lines in file
{
    int Checker1;   //Needed variable which could  compare signs in
    ASCII
    while ((Checker1 = fgetc(Filename)) != EOF) //Checking Till End
    {
      if (Checker1 == '\n') // New line found
      {
          NLines = NLines + 1; //increasing variable everytime it finds
    new line
      }
    }

    return NLines; //function returns number of lines
}

void FWPath(int i, int j) //It should have found order of path (using
    recurection) but it has memory crash but i wont hide it as a comment
{
    if (i == j) printf("%d  ", i);
    else if (PathTable[i][j] == 0) printf("No Path");    //No path while
    there is 0 in PathTable
    else
    {
      FWPath(i, PathTable[i][j]);
      printf("%d  ", j);
    }
}

int main()
{

    FILE *fp;
    fp = fopen("city.txt", "r");  //creating pointer and assigning it to
    file city in reading mode
    if (fp == NULL)              //condition checking if stream isn't empty
    {
      printf("Something is wrong with file. Please correct data."); //if
    stream is empty program won't start compiling code
      return -1;
    }
    else
    {

      // GETTING NUMBER OF LINES (LINE COUNTER)
```

```
47.     int NumberOfLines = 1;
48.     NumberOfLines = LineCounter(fp, NumberOfLines);//Using
    created function to count number of lines
49.     printf("Number Of Lines in file :%d\n", NumberOfLines);
50.     //Refreshing fp after last initation
51.     rewind(fp);
52.     //Another part of variables
53.     int infinity = INT_MAX; //Will be used in filling matrices
54.     int MAX_NAME_LEN = 30; //Used to limit memory needed in
    allocation
55.     //Taking one line (First Line)
56.     int NumbCities;
57.     fscanf(fp, "%d", &NumbCities);
58.     printf("Number of city declared :%d\n\n", NumbCities);
    //showing if number of city is correct by printing it
59.
60.     int **FloydTable; //Creating dynamic two dimensial array used to
        make floyd warshall algorithm on it
61.     FloydTable = (int **)malloc(NumbCities * sizeof(int*));
62.     for (int i = 0; i < NumbCities; i++)    //allocating memory for every
    array
63.     {
64.         FloydTable[i] = (int *)malloc(NumbCities * sizeof(int));
65.     }
66.
67.     PathTable = (int **)malloc(NumbCities * sizeof(int*));//Creating
    dynamic two dimensial array used to carry out path ('next move)
68.     for (int i = 0; i < NumbCities; i++)
69.     {
70.         PathTable[i] = (int *)malloc(NumbCities * sizeof(int));
    //allocating memory for it
71.     }
72.
73.
74.     //Filling FloydTable
75.     for (int k = 0; k < NumbCities; k++)
76.     {
77.
78.         for (int j = 0; j < NumbCities; j++)
79.         {
80.             if (k == j)
81.             {
82.                         FloydTable[k][j] = 0;// We fill diagonal with
    zeros (because distance between a_1 and a_1 is 0
83.             }
```

```
84.                 else
85.                 {
86.                         FloydTable[k][j] = infinity; //For now we assume
    there is no other connections between cities, so I fill them as an infinity
87.                 }
88.
89.
90.             }
91.             }
92.             //Filling PathTable
93.         for (int i = 0; i < NumbCities; i++)
94.         {
95.
96.                         for (int j = 0; j < NumbCities; j++)
97.             {
98.                 PathTable[i][j] = 0; //Assume that there are no connections
    available -> 0
99.             }
100.
101.            }
102.
103.
104.
105.
106.
107.
108.
109.
110.            //Time for getting city names from file into arrays
111.            char **dynCNames;
112.            dynCNames = (char **)malloc(NumbCities *
    sizeof(char*));//Also here, we allocate memory for it , two dimensional
    char array)
113.            for (int i = 0; i < NumbCities; i++)
114.            {
115.                dynCNames[i] = (char *)malloc(MAX_NAME_LEN *
    sizeof(char));
116.
117.            }
118.            //Getting combination of cities and distances between them into
          array
119.            char **dynCDist;
120.            dynCDist = (char **)malloc((NumbCities*NumbCities) *
    sizeof(char*));
121.            for (int i = 0; i < NumbCities*NumbCities; i++)
```

```
122.        {
123.            dynCDist[i] = (char *)malloc(MAX_NAME_LEN *
    sizeof(char)); //allocating memory
124.
125.        }
126.
127.            //      Taking City Names
128.        const size_t line_size = 300;
129.        char *line = (char *)malloc(100);
130.
131.
132.
133.        printf("LIST OF DECLARED CITIES\n");
134.        int counter = 1; //Used to get only city names, it is counter of
    lines to know, what we're getting - only city names, or distance
    combinations
135.        while (!feof(fp))
136.        {
137.            while (fgets(line, line_size, fp) != NULL)//Taking lane from
    file to buffer line
138.            {
139.                if (counter >= 2 && counter <= NumbCities + 1)
140.                {
141.                    strcpy(dynCNames[counter - 2], line);//copying
    from buffer to array with city names
142.                    printf("%s", dynCNames[counter - 2]);
143.                    counter++;//increasing counter to move to
    another line
144.
145.                }
146.                else
147.                {
148.                    if (counter > NumbCities)//counter is bigger
    than number of cities so we move to combinations
149.                    {
150.                        strcpy(dynCDist[counter - (NumbCities +
    1)], line);//copying from buffer to array with whole combinations
151.                        counter++;
152.                    }
153.                    else
154.                        counter++; //it is used only in first step, to
    avoid putting first line into array
155.
156.                }
157.            }
```

```
158.        }
159.
160.        printf("\n\n");
161.
162.
163.        char FirstCChecker[20];   //Used as a buffer for first city name
164.        char SecondCChecker[20];//Used as a buffer for second city
    name
165.        int DistanceValue;              //Used as a buffer for distance
    between mentioned first and second city
166.        //Dividing Cities&Distance combinations
167.        for (int k = 1; k < NumberOfLines - (NumbCities); k++)
168.        {
169.
170.            sscanf(dynCDist[k], "%s %s %d", FirstCChecker,
    SecondCChecker, &DistanceValue);// sscanf used to partialy divide
    array dynCDist
171.
172.            for (int i = 0; i < NumbCities; i++)
173.            {
174.                    if (strncmp(FirstCChecker, dynCNames[i],
    strlen(FirstCChecker)) == 0)//We check if first buffer is same as one
    element of dynCNames for length of first buffer
175.                            for (int j = 0; j < NumbCities; j++) //if and only
    if upper condition is satisfied
176.                            {
177.                            if (strncmp(SecondCChecker, dynCNames[j],
    strlen(SecondCChecker)) == 0)//we do the same but with second buffer
178.                            {
179.                                    FloydTable[i][j] = DistanceValue;//if
    every condition was satisfied so put in floydtable at position [i][j] value
    of distance buffer
180.                            }
181.                            }
182.            }
183.            memset(FirstCChecker, 0, strlen(FirstCChecker));//reseting
    1st buffer
184.            memset(SecondCChecker, 0,
    strlen(SecondCChecker));//reseting second buffer
185.        }//there is no need to reset DistanceValue because it is integer
    and it can be overwritten
186.        //Filling PathTable
187.        for (int k = 1; k < NumberOfLines - (NumbCities); k++)
188.        {
189.
```

```
190.            sscanf(dynCDist[k], "%s %s %d", FirstCChecker,
   SecondCChecker, &DistanceValue);
191.
192.            for (int i = 0; i < NumbCities; i++)
193.            {
194.                    if (strncmp(FirstCChecker, dynCNames[i],
   strlen(FirstCChecker)) == 0) //Same as in filling FloydTable
195.                            for (int j = 0; j < NumbCities; j++)
196.                            {
197.                            if (strncmp(SecondCChecker, dynCNames[j],
   strlen(SecondCChecker)) == 0)//Same as in filling FloydTable
198.                            {
199.                                    PathTable[i][j]=i+1; //Whole procedure is
   to fulfill PathTable with possible connections
200.                            }
201.                            }
202.            }
203.            memset(FirstCChecker, 0, strlen(FirstCChecker));
204.            memset(SecondCChecker, 0,
   strlen(SecondCChecker));//Again    cleaning buffers in every loop
   transition- I will use them later too
205.        }
206.        //Floyd Warshall Algorithm
207.        for (int k = 0; k < NumbCities; k++)
208.        {
209.            for (int i = 0; i < NumbCities; i++)
210.            {
211.                    for (int j = 0; j < NumbCities; j++)
212.                    {
213.                            if (FloydTable[i][k] == infinity || FloydTable[k]
   [j] == infinity)continue; //Operations on places with infinity
214.                            if (FloydTable[i][j] > FloydTable[i][k] +
   FloydTable[k][j]) //If distance from i to j is bigger than distances from i
   to k and from k to j then
215.                                    //we overwrite dist[i][j] as a sum of these
   2 distances
216.                            {
217.                                    FloydTable[i][j] = FloydTable[i][k] +
       FloydTable[k][j];
218.                                    PathTable[i][j] = k+1; //to avoid zeros,
   doing changes in pathtable
219.                            }
220.                    }
221.            }
222.        }
```

```
223.        /*printf("RESULTS BELOW CITY COMBINATION\n");
224.        for (int i = 0; i < NumbCities; i++)
225.        {
226.            for (int j = 0; j < NumbCities; j++)
227.            {
228.                    //printf("%d  ", i);// I used them to check if answers
    were correct. (To avoid big names I've just used numbers)
229.            //      printf("%d  ", j);
230.
231.                    printf("%s", dynCNames[i]); //printing name of city
    which is     in table with i index
232.                    printf("%s", dynCNames[j]);// same for j
233.                    if (FloydTable[i][j] == infinity)//
234.                    {
235.                            printf("No path\n");
236.                    }
237.                    else
238.                    {
239.                            printf("%d\n", FloydTable[i][j]);
240.                    }
241.                    printf("\n");
242.            }
243.        }
244.        */// These loops were used to write combination of cities + their
    distances, no needed but I will keep them in code
245.
246.
247.        /* I WILL LEAVE IT IN SOURCE CODE JUST IN CASE
    SOMEONE WILL WANT TO SEE MATRIX
248.        for (int i = 0; i<NumbCities; i++)
249.        {
250.
251.        for (int j = 0; j < NumbCities; j++)
252.        {
253.        printf("%d\t", FloydTable[i][j]);
254.
255.        }
   1.       printf("\n");
256.        }
257.        printf("\n\n");
258.        */
259.        //PART WHEN WE GET DATA FROM USER TO OUTPUT
          NEEDED DISTANCE
260.        // I can use same char 'buffers' like FirstCChecker because
    they were erased in last invocation in for loop
```

```
261.        // So there is no need to create another buffer-like table
262.        int test = 1; //This is variable that is changed during while loop
    (used to hold loop - possibility to ask for other distances)
263.        char *statement = (char *)malloc(sizeof(char));//Statement will
    be the input from user if he wants to continue looking for.
264.        while (test)// test=1 so loop is always realised till test = 0
265.        {
266.            printf("Please write down name of first city then press
    enter\n");
267.            scanf("%s", FirstCChecker);//Getting first buffer
268.
269.            printf("Please write down name of second city then press
        enter\n");
270.            scanf("%s", SecondCChecker);//Getting second buffer
271.            for (int i = 0; i < NumbCities; i++)
272.            {
273.                if (strncmp(FirstCChecker, dynCNames[i],
    strlen(FirstCChecker)) == 0)//If first buffer is same as name of city
274.                    for (int j = 0; j < NumbCities; j++)
275.                    {
276.                    if (strncmp(SecondCChecker, dynCNames[j],
    strlen(SecondCChecker)) == 0)//if second buffer is same as name of city
277.                    {
278.                        printf("Distance : %d\n", FloydTable[i]
    [j]); //we get distance value for buffer1 and buffer2
279.                        printf("Intermediate city (Next in
    way) :\t");
280.                        if (PathTable[i][j] == i+1) //if next city is
    same city as first one (moving from a to b through a)
281.                        {
282.                            printf("No intermediate
    cities\n\n");//It makes no sense, so there is no intermediate city
283.                        }
284.                        else
285.                        {
286.                            if (PathTable[i][j] != 0) //we don't
    need same cities
287.                            {
288.                                printf("%s\n",
    dynCNames[PathTable[i][j] - 1]);//
289.                                printf("Distance to
    intermediate city :\t\t");
290.                                printf("%d\n\n",
    FloydTable[i][PathTable[i][j] - 1]); //Taking distance from FloydTable
    with coordinates : i and intermediate city
```

```
291.
292.                                     }
293.                                     else
294.                                             printf("No intermediate
    cities\n");
        1.                           }
295.                         }
296.                         }
297.             }
298.
299.
300.             printf("Do you want to try other cities? \n\tYes/No\n");
    //Condition to continue checking values
301.
302.
303.             scanf("%s", statement);
304.             if (strncmp(statement, "Yes", 1) == 0)
                 {
305.                     memset(statement, 0, strlen(statement));
306.                     continue;
307.             }
308.             if (strncmp(statement, "No", 1) == 0)
309.             {
310.                     memset(statement, 0, strlen(statement));
311.                     test = 0;
312.             }
313.             memset(statement, 0, strlen(statement));
314.             memset(FirstCChecker, 0, strlen(FirstCChecker));
315.             memset(SecondCChecker, 0, strlen(SecondCChecker));
316.             }
317.     }
318.
319.
320.
321.
322.
323.
324.     fclose(fp);
325.
326.     getchar();
327.     return 0;
328.   }
```

6. Testing

All tests were made on examples from the internet and programm was giving correct output. I won't adjoin here

something special. In folder Project1 there will be another folder named Example, there is examplary test with given inputs, fulfilled file. Also in Folder Project there is adjoined city.txt (ready for use) this is also example from internet. Programm works correctly on this data. So there are 2 examples which I prepared.

7. Conclusions

I have nothing to say about already created (on my own) code. The whole programm isn't ready. I have to make correctly whole map of steps. From one destination to another. I was trying to use recursion function, visual studio didnt find error, but there was memory crash after debugging. So it is not completed code but time is about run out. So I think I will stop at this moment.