ENGR 476-02

Lab 3: Dijkstra Routing Project

By: Ahmad El Shakoushy

12-3-18

```c
//Name: Ahmad El Shakoushy
//ID: 915814671
//ENGR 476-02: Dijkstra Routing Project

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>


//NOTE: I am defining these types outside of the main function because I'd like to be able
to make separate helper functions that use these types
typedef int bool; //defining a boolean type variable to make our code more readable
enum {false,
      true};

typedef struct    //Declaring a Node data type
{
    char name[2]; //holds name of the Node
    int cost_from_start;
    bool is_start;
    bool is_end;
    bool visited;
    char prev_name[2]; //holds the name of the previous object
    char neighbors[100][2]; //100 rows x 2 columns
    int num_neighbors;
} Node;

typedef struct    //Declaring an Edge data type
{
    char name1[2];
    char name2[2];
    int cost_of_edge;
} Edge;

bool exister(Node* arr, int arr_size, char target[2]); //takes in an array of Nodes and the
array size as well as the name of a certain node and returns whether that node exists in the
array; false --> not in array, true --> in array
void node_setter(Node* n, char name[2], bool start, bool end, bool visited, int inital_cost,
char prev[2]);
void edge_setter(Edge* e, char name1[2], char name2[2], int edge_cost);

int main()
{
    char choice_start[2];
    char choice_end[2];
    int num_visited = 0;
    int num_nodes = 0;


    Node nodes[100];
    Node last_obj;
    int start_index = -2;  //-2 because we want it throw an exception if something is wrong
    int last_obj_index = -2; //-2 because we want it throw an exception if something is wrong
    Edge edges[500];
    Node* current; //this pointer will always point to the current node in the dijkstra
traversal algorithm

    //user input
    printf("Please enter the start node: ");
    scanf("%s", choice_start);

    printf("Please enter the end node: ");
    scanf("%s", choice_end);

    FILE* fpointer = fopen("dijkstra_input.txt","r"); //opening file via file pointer
    int num_lines = 0; //holds number of edges, which is the same as the number of lines in
```

```
the input file

    rewind(fpointer); // ensuring that pointer is at the start of the file

//this code block handles Edge and Node creation, ensuring no duplicate nodes
    while(!feof(fpointer))
    {
        bool name1_exists = false;
        bool name2_exists = false;
        char n1[2];
        char n2[2];

        int edge_cost;
        fscanf(fpointer, "%s %s %d", n1, n2, &edge_cost); //reading line

        //NOTE: for edges, we add both cases of edges: name1 -> name2 and name2 -> name1,
because edges are bidirectional and to make retrieval from the list easy
        edge_setter(&edges[num_lines], n1, n2, edge_cost);
        num_lines++;

        edge_setter(&edges[num_lines], n2, n1, edge_cost);
        num_lines++;

        name1_exists = exister(nodes, num_nodes, n1);
        name2_exists = exister(nodes, num_nodes, n2);

        if (!name1_exists)    //if the node for name1 doesn't exist, we create it and add to
nodes list
        {
            if (strcmp(choice_start, n1) == 0)    //start node
            {
               node_setter(&nodes[num_nodes], n1 ,true, false, false, 0, "");
            }
            else if (strcmp(choice_end, n1) == 0)      //end
            {
             node_setter(&nodes[num_nodes], n1 ,false, true, false, 1000, "");
                last_obj_index = num_nodes;
            }
            else      //not start or end
            {
                node_setter(&nodes[num_nodes], n1 ,false, false, false, 1000, "");
            }

            num_nodes++;
        }
        if (!name2_exists)    //if the node for name1 doesn't exist, we create it and add to
nodes list
        {

            if (strcmp(choice_start, n2) == 0)    //start node
            {
                node_setter(&nodes[num_nodes], n2 ,true, false, false, 0, "");
            }
            else if (strcmp(choice_end, n2) == 0)      //end
            {
              node_setter(&nodes[num_nodes], n2 ,false, true, false, 1000, "");
              last_obj_index = num_nodes;
            }
            else      //not start or end
            {
            node_setter(&nodes[num_nodes], n2 ,false, false, false, 1000, "");
            }

            num_nodes++;
        }

    }
```

```
    for(int i = 0; i < num_nodes; i++)
    {
        if(nodes[i].is_start == true)
        {
            start_index = i;
        }
    }


    rewind(fpointer); //puts fpointer at start of file again

    char n1[2];
    char n2[2];

    while(!feof(fpointer))    //this sets neighbors
    {

        Node obj1;
        Node obj2;

        int edge_cost;
        fscanf(fpointer, "%s %s %d", n1, n2, &edge_cost); //reading line

        int obj1_index = -2;
        int obj2_index = -2;

        for (int i = 0; i < num_nodes; i++)      //after this loop, obj1 and obj2 will hold
two adjacent nodes
        {
            if (strcmp(nodes[i].name, n1) == 0)
            {
                obj1_index = i;
                obj1 = nodes[i];  //set obj1 to first node
            }
            if (strcmp(nodes[i].name, n2) == 0)
            {
                obj2_index = i;
                obj2 = nodes[i]; //set obj2 to second node
            }
        }
        //checking if obj1's neighbors contain obj2, if not we add it to its neighbors
        bool has_neighbor1 = false;
        bool has_neighbor2 = false;

        for(int i = 0; i < obj1.num_neighbors; i++)
        {
            if(strcmp(obj1.neighbors[i], obj2.name) == 0)
            {
                has_neighbor1 = true;
            }
        }
        for(int i = 0; i < obj2.num_neighbors; i++)
        {
            if(strcmp(obj2.neighbors[i], obj1.name) == 0)
            {
                has_neighbor1 = true;
            }
        }

        if(has_neighbor1 == false)
        {
            strcpy(nodes[obj1_index].neighbors[obj1.num_neighbors], obj2.name);
            nodes[obj1_index].num_neighbors++;
        }
        if(has_neighbor2 == false)
```

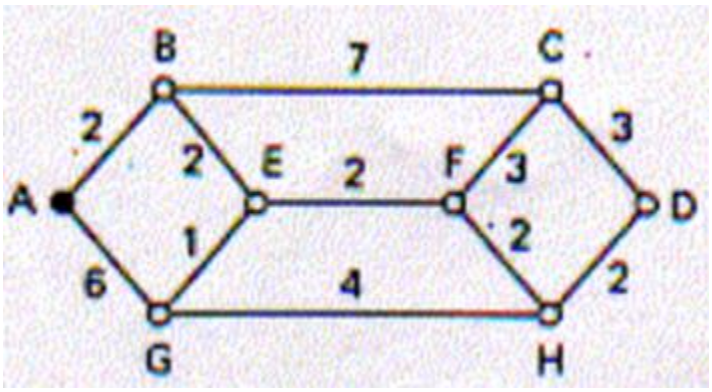
```
        {
            strcpy(nodes[obj2_index].neighbors[obj2.num_neighbors], obj1.name);
            nodes[obj2_index].num_neighbors++;
        }

    }
    fclose(fpointer);

    Node temp_val = nodes[0];
    nodes[0] = nodes[start_index];
    nodes[start_index] = temp_val;


    current = &nodes[0];



    while (num_visited < num_nodes)
    {
        int temp_min = 100000;
        Node* temp_min_node;

        for (int j = 0; j < (*current).num_neighbors; j++)     //loop through all neighbors
for the node
        {
            char current_neighbor_name[2];
            strcpy(current_neighbor_name, (*current).neighbors[j]); //holds current
neighbor's name, is used to get actual neighbor
            int current_neighbor_real_index = -2; //this holds the index of the current
neighbor in the actual nodes array

            for(int i = 0; i < num_nodes; i++)
            {

                if(strcmp(nodes[i].name,(current_neighbor_name))== 0)       //    0 ==> equal
                {
                    current_neighbor_real_index = i;
                }
            }

            int curr_neighbor_cost = nodes[current_neighbor_real_index].cost_from_start;
            int curr_node_cost = (*current).cost_from_start;
            int edge_to_neighbor_cost;

            //we need to loop through the edges to get the value of the edge to the current
neighbor
            for(int i = 0; i < num_lines; i++)
            {
                if( (strcmp(edges[i].name1, (*current).name) == 0) && (strcmp(edges[i].name2,
current_neighbor_name) ==0) )  //if name1 is current name, name2 is neighbor name, we get the
edge value
                {
                    edge_to_neighbor_cost = edges[i].cost_of_edge;
                }
            }

            (*current).visited = true; //setting as visited after checking all neighbor nodes

            if ( curr_neighbor_cost > curr_node_cost + edge_to_neighbor_cost &&
((nodes[current_neighbor_real_index].visited) == false) )   //is neighbor cost greater? (if
so we should overwrite it)
            {
                nodes[current_neighbor_real_index].cost_from_start = curr_node_cost +
edge_to_neighbor_cost; //updating neighbor cost
                strcpy(nodes[current_neighbor_real_index].prev_name, (*current).name);
//updating prev_name of neighbor to hold current name
            }
```

```
        }
        num_visited++;

        for (int i = 1; i < num_nodes; i++)    //searching for next minimum node
        {
            if (i != start_index)    //make sure you don't compare node with itself; 1 ==> not
equal
            {

                if (nodes[i].cost_from_start < temp_min && (nodes[i].visited == false))
//if current neighbor has a smaller cost, set it to that as long as its unvisited
                {
                    temp_min = nodes[i].cost_from_start;
                    temp_min_node = &nodes[i];
                }
            }

        }

        (current) = temp_min_node; // advancing current
        start_index++;


    }


    start_index -= num_nodes;
    char str_result[50]; //holds result before flipping
    char str_result_flipped[50]; //holds result after flipping: this will eventually hold the
final path
    int str_result_max_index = 0;
    int result_cost;



    for(int i = 0; i < num_nodes; i++)
    {
        if(nodes[i].is_end)
        {
            last_obj_index = i;
            break;
        }
    }

    result_cost = nodes[last_obj_index].cost_from_start;

    if(strcmp(choice_end, choice_start)!=0)
    {
        char current_obj_name[2];
        Node* current_obj = &nodes[last_obj_index]; //starting with last obj
        Node* prev;
        char previous_name[2];
        strcpy( previous_name,(*current_obj).prev_name);

        for(int i = 0; i < num_nodes; i++)  //getting actual before object, can be turned
into function
        {
            if(strcmp(previous_name,nodes[i].name) == 0)
            {
                current_obj = &nodes[i];
                //break;
            }
        }

        while( (strcmp((*current_obj).name, choice_start) != 0) )  //while current object's
name is not the start name, we keep going back
        {
```

```
            strcat(str_result, ">");
            str_result_max_index++;
            strcat(str_result, "-");
            str_result_max_index++;
            strcat(str_result, (*current_obj).name);
            str_result_max_index++;

            for(int i = 0; i < num_nodes; i++) //getting actual before object, can be turned
into function
            {
                if(strcmp((*current_obj).prev_name,nodes[i].name) == 0)
                {
                    current_obj = &nodes[i];
                    break;  //we break to avoid skipping a node in edge cases
                }
            }

        }

        strcat(str_result, ">");
        str_result_max_index++;
        strcat(str_result, "-");
        str_result_max_index++;
        strcat(str_result, choice_start);

        int index = 0;

        for(int i = str_result_max_index; i >= 0; i--)
        {
            str_result_flipped[index] = str_result[i];
            index++;
        }
        strcat(str_result_flipped,nodes[last_obj_index].name);

    }
    else
    {
        result_cost = 0;
        strcpy(str_result_flipped, choice_start);
        strcat(str_result_flipped,"->");
        strcat(str_result_flipped, choice_end);
    }

    printf("\n");
    printf("%s", "Total cost = ");
    printf("%d", result_cost);
    printf("\n");
    printf("%s", str_result_flipped);
    printf("\n");

    return 0;

}

bool exister(Node* arr, int arr_size, char target[2]){
    bool result = false; //holds final result
            for(int i = 0; i < arr_size; i++) //looping through all nodes
            {
                if(strcmp(arr[i].name,target) == 0)
                {
                    result = true;
                }

            }
    return result;
}
```

```
void node_setter(Node* n, char name[2] ,bool start, bool end, bool visited, int inital_cost,
char prev[2]){
    strcpy((*n).name,name);
    (*n).is_start = start;
    (*n).is_end = end;
    (*n).visited = visited;
    (*n).cost_from_start = inital_cost;
    strcpy((*n).prev_name, prev);
}

void edge_setter(Edge* e, char name1[2], char name2[2], int edge_cost){
    strcpy((*e).name1, name1);
    strcpy((*e).name2, name2);
    (*e).cost_of_edge = edge_cost;
}
```

Map used for map input file:



Actual input file for the map:

```
A B 2
A G 6
B E 2
B C 7
G E 1
E F 2
F C 3
C D 3
F H 2
G H 4
H D 2
```

Sample Output:

```
Please enter the start node: H
Please enter the end node: A

Total cost = 8
H->F->E->B->A
```