Computer Science 3270
Project 1
C Programming Project

Notes

Some parts of this project contain mathematical formulas, but nothing more than you have seen in Math 165 or before. There may be parts of this project description that are ambiguous (not intentionally), but it is your responsibility to clarify the program specifications provided in natural language. Please read the assignments early and ask questions in class. Questions that lead to further specification or changes in the project specification will be posted in Piazza or in the announcements section of the Canvas course.

Introduction

Map data and location data are utilized in various ways and are ubiquitous in mobile applications. In this project, map data will be used to find routes and other properties of the map data for Ames, Iowa. However, your program must operate and will be tested on data from other locations.

Data Format

Road data is stored in a file with two distinct sections: the Points of Interest section and the Road section. The first section is the Points of Interest, with the first line of the file giving the number of entries. This is followed by a Point of Interest (POI), one per line. Each line consists of comma-separated values as follows: An ID number of the POI, the name of the POI, the latitude of the POI, and the longitude of the POI. Note that Ames is at 42 degrees North latitude and 93 degrees West longitude. North and East latitudes and longitudes are represented by positive numbers, while negative numbers represent South and West. Thus, Ames is located at a longitude of -93 degrees.

The second part of the file represents roads over a region. The format is similar to the POI section, with a single line that gives the number of entries in the section. This is followed by entries that define roads in terms of from/to nodes, along with the distance of the road segment in meters, the latitude and longitude of the segment's start, and the name of the road. An example entry looks like the following:

```
161179915,161133478,50.19,42.02475,-93.6468,Union Drive
```

Note that some road lengths may contain NaN, indicating that there is no length associated with the road. For the purpose of any calculation, these are considered to be zero.

Project 1, Part a

The purpose of this part of the project is to create the infrastructure for the assignment and to do a simple C program that will get you familiar with the process. This part of the project is divided into several parts.

1. Remove or rename any previous Git repositories that you have created with the following name, and create a new repository named "**coms3270P1**" Also remember that part of your grade will depend on README and DEVELOPER files. Please see the syllabus.

2. Create at least 3 small data test files in the format described above that will help you debug your program. These test files should be reasonably small so that any computation your program does can be verified by hand. As an example, consider a test file with one long road with only one intersection and two points of interest along the road. In this case, the road is divided into four segments with three intersections. The road section of the data file would contain four lines.

3. Create a function named "validate" that reads a data file from standard input and determines if the data file is valid. (Note that this will be updated in other parts of this project.) For this part of the project, the following conditions give an invalid file.

    a. Latitude or Longitude out of legal range. For example, the longitude is greater than or equal to 180.
    b. There is an empty TO or FROM Id or the road name is empty.
    c. The Id numbers contain any other character other than the digits 0 through 9.
    d. If either of the numbers that give the number of entries is 0 or negative.
    e. If latitude and longitudes have characters that do not allow conversion to a number using scanf.
    f. If the POI name is empty.

    The function returns an integer of 0 if the file is valid, and otherwise returns the line number of the file for the first invalid entry. Place and save this function in a file named data.c.

4. Create a "main" function that calls the "validate" function and place this function in the file "mapper.c" The main function then outputs if the data file read is valid by printing the single word "VALID" to stdout, or the line number where the first invalid data was found.

5. Create a header file named "data.h" that creates a correct prototype for the function defined in 3 above. Be sure to include correct guards for the header file.

6. Create a makefile for this project so that when a user types make in the top directory for the repository, the executable program named "mapper" that if run, the main program in the "mapper.c" file is executed.

7. Create and store your project in the gitlab repository git.las.iastate.edu. When completed and your program is correct and documented, create a tag with the name "mapperAcomplete"

8. At least 10% of points will come from documentation and following the syllabus requirements for the project.

Project 1, Part b

In Part b of the project, you will create a graph data structure to be used in processing the road data in Part c. Part b is divided into several parts.

1. The following data types are used in this project.

```c
typedef struct edge edge_t;
typedef struct node node_t;

struct edge
{
	node_t* toNode;
	float weight;
	void* data; // Pointer to additional edge data
	edge_t* next; // Pointer to the next edge in the adjacency list
};

struct node
{
	int id;
	void* data;    // Pointer to additional node data
	edge_t* edges; // Pointer to the head of the adjacency list
};

typedef struct
{
	node_t** nodes; // Array of pointers to nodes
	int nodeCount;
	int edgeCount;
	int nodeSpace;  // Current allocated space for nodes
} graph_t;
```

Place these structures in the file graph.h. Be sure to include any appropriate guards and other header files that clients of the graph code need.

2. Add the following prototypes for graph functions to your header file. Do not change these as we will link our own code with your code to test it. If you think there is an error (logic or syntax) with these prototypes, please post to Piazza as soon as possible.

```c
/**
* Creates a new graph and returns a pointer to it. T
* The graph is initialized with no nodes or edges,
* and returns NULL if memory allocation fails.
* All elements of the graph are stored on the heap.
* The node’s adjacency list is a simple single linked list
* The pointers to all the nodes in the graph are stored in
* an array of pointers.  These array is initialized to an
* initial capacity of 100 nodes.  If more nodes are added,
* it behaves like an ArrayList in Java, where if the list is
* full, the list of node pointers is expanded to double its size
* and then a new node pointer is added.
**/
graph_t* createGraph();

/**
* Frees the memory used by the graph.
* All nodes and edges in the graph are also freed.
* If the graph pointer is NULL, the function does nothing.
**/
void freeGraph(graph_t* graph);

/**
* Adds a new node to the graph.
* @param graph Pointer to the graph.
* @param id Unique identifier for the new node.
* @param data Pointer to additional node data.
* @return Pointer to the newly created node, or NULL on failure.
**/
node_t* addNode(graph_t* graph, int id, void* data);

/**
* Adds a new edge to the graph.
* @param graph Pointer to the graph.
* @param fromId ID of the source node.
* @param toId ID of the destination node.
* @param weight Weight of the edge.
* @param data Pointer to additional edge data.
* @return Pointer to the newly created edge, or NULL on failure.
* The function fails if either node does not exist or if an edge already
* exists between the two nodes.
* All edges are directed from the source node to the destination node.
* **/
edge_t* addEdge(graph_t* graph, int fromId, int toId, float weight, void*
data);
```

```c
/**
* Retrieves a node from the graph by its ID.
* @param graph Pointer to the graph.
* @param id ID of the node to retrieve.
* @return Pointer to the node, or NULL if not found.
* **/
node_t* getNode(graph_t* graph, int id);

/**
* Retrieves an edge from the graph by its source and destination node IDs.
* @param graph Pointer to the graph.
* @param fromId ID of the source node.
* @param toId ID of the destination node.
* @return Pointer to the edge, or NULL if not found.
**/
edge_t* getEdge(graph_t* graph, int fromId, int toId);

/**
* Removes a node from the graph.
* @param graph Pointer to the graph.
* @param id ID of the node to remove.
* @return 1 if the node was removed successfully, 0 if not found.
**/
int removeNode(graph_t* graph, int id);

/**
* Removes an edge from the graph.
* @param graph Pointer to the graph.
* @param fromId ID of the source node.
* @param toId ID of the destination node.
* @return 1 if the edge was removed successfully, 0 if not found.
**/
int removeEdge(graph_t* graph, int fromId, int toId);
```

```c
/**
* Prints the entire graph to the console.
* @param graph Pointer to the graph.
* The function prints each node and its outgoing edges in a readable format.
* If the graph pointer is NULL, the function does nothing.
*
* Example output:
* Node 1: (data)
* -> Node 2 (weight: 1.0, data)
* -> Node 3 (weight: 2.5, data)
* Node 2: (data)
* -> Node 3 (weight: 1.5, data)
* Node 3: (data)
* (no outgoing edges)
*
**/
void printGraph(graph_t* graph);
```

3. Implement the graph prototype functions above and place them in the file graph.c

4. Create files testgraph.h and testgraph.c. Write a main function that reads a road data file from stdin, defined in part a, and then stores the information in a graph data structure. Nodes in the graph come from the points of interest and edges in the graph are defined by the roads. The weight for the edge is the distance of the road given in the file. The extra "data" in the node is the name of the POI, latitude, and longitude. (You will want to make a structure in testgraph.h to define this) The extra information in the edge is the name of the road. The program then calls printGraph to output the graph to stdout.

5. Update the makefile with a target of testgraph that creates the executable testgraph with "make testgraph" is executed. Note that your makefile should perform the minimum number of steps necessary to create the testgraph executable.

6. Document your code and update the README and DEVELOPER files as necessary.

9. Update your repository and tag the working version of your Part b code with the tag: "mapperBcomplete"

Project 1, Part c

At this point, you have a graph data structure, and you can read in city road data into a graph data structure. In Part c, we will use this graph structure to compute some data about road data. This is accomplished through a single executable file named **citydata** using different command-line parameters as detailed below. You may assume all road data distances (weights) are in meters.

1. Command-line parameters may come in any order.

2. If multiple command-line parameters produce output, they are output in the order they appear in the command.

3. Running citydata with no parameters returns a usage statement. A usage statement lists all the possible options and parameters for a command. You may chose any format to convey the appropriate information. Note that some parameters may only be valid/or depend on other parameters. In other words, you may have to print several lines. A simple example of a usage statement is the ssh command.

4. The required parameter -f <filename> specifies the name of the tab-separated city file to load. This is used in conjunction with other parameters that specify an operation.

5. The parameter -location <locationname> finds a point of interest named <locationname> and outputs the latitude and longitude to stdout.

6. The parameter -diameter find the largest distance in meters (not using roads, but as the crow flies) between two locations (nodes) in the city data. Note that you will have to look up how to convert the distance between two latitude/longitude locations into meters. Output the latitude and longitude of the two points followed by the distance between them in meters to stdout.

7. The parameter -distance <name1, name2> finds the two named locations (nodes) in the file and determines the distance (not using roads, but as the crow flies) in meters between the two locations. The output to stdout is simply the distance in meters.

8. The parameter -roaddist <name1, name2> finds the two named locations (nodes) in the file and determines the shortest route using roads from the name1 location to the name2 location. The algorithm for doing this should use Dijkstra's algorithm, as an exhaustive search may take days to run on larger files. The output is to stdout and is in meters.

10. Update your repository and tag the working version of your Part c code with the tag: "mapperCcomplete"

Example outputs from commands. Note locations and distances are not accurate; they are just examples of the format.

```
./citydata -f datafile.tsv -location "Atanasoff Hall"
42.0403 -93.6317
```

```
./citydata -f datafile.tsv -diameter
42.01043 -93.4323 42.0643 -93.7643 40342.55
```

```
./citydata -f datafile.tsv -distance "Atanasoff Hall" "Kildee Hall"
1048.432
```

```
./citydata -f datafile.tsv -roaddist "Ames Highschool" "Coffee Place"
20343.342
```

Note that the quotes around names in the above examples are moved in argv but will remain one parameter. For example, in the first example above argv[1] = Atanasoff Hall.

Don't forget to create tags (not just a branch) for each of the parts.

Don't forget to include queries you used to LLM models in your DEVELOPER file if you used an AI model to complete this assignment.

Don't forget to test each part on pyrite. You should clone to Pyrite using the appropriate tag, cd to the repository directory, type make, and then run the correctly named executable file from that directory.