Diwa Ashwini Vittala
CSE 13S - Fall 2021

⁰❀ Assignment 4: Design ❀。

THE PERAMBULATIONS OF DENVER LONG

About the program:

The program maps the shortest path from one point to another using Depth-First Search (DFS). The user has various command options that they can use to build a path from a graph (either existing or one that they can create as they run the program).

About the file sorting.c:

Input(s): in line command of one of the following commands:
-u use undirected graph
-v verbose printing of the path as well as how many times dfs used recursion
-i infile or standard input
-o outfile or standard output
-h help
Must specify a file for -i or -o.
Output: The program outputs paths starting from the first location in the graph, travels to each point in the graph, then makes it back to the first location. For each path, the program outputs the path length. In the end, the number of recursive calls used is printed. If -v was not chosen, only the shortest path is shown along with the number of recursive calls and shortest path length.

Other files involved:

Makefile, README.md, graph.c, graph.h, path.c, path.h, stack.c, stack.h, vertices.h, tsp.c, and a folder “graphs” containing diego.graph, short.graph, texas.graph, mythical.graph, solarsystem.graph, ucsc.graph.

Pseudocode/solution breakdown for:

tsp.c:

Include all the necessary header files for the graphs, path, stacks, etc.
Include basic header files from the inbuilt library such as standard input, unistd, etc.

Begin the help message function:
    Print out proper usage instructions

Begin the DFS function:
    Set a 32 bit unsigned integer variable to 0 as a filler (or insignificant) variable
    Mark the vertex as visited
    Add one to the recursive count

If we are able to add the vertex to the current path and the number of vertices in this path is not equal to the number of vertices in the entire graph (meaning its not full):

For a vertex "next" from 0 to the number of vertices in the graph:

If next is adjacent to the vertex (there is an edge connecting them) and it has not been visited:

Perform a recursion by passing next in as the vertex

Else if the graph has an edge from the vertex to 0 (connects back to origin) and we can push the origin into the current path:

If the current path is shorter than the previous shortest path, or if there is not shortest path yet:

Copy the current path into the shortest path

If the user wanted verbose printing:

Print the length of the path and the path itself

Pop the top of the current path off and store it in the insignificant variable initialized in the beginning of the function

Pop the top of the current path off and store it in the insignificant variable initialized in the beginning of the function

Mark the variable we popped off as unvisited

Define the possible command line options: u, v, h, i, and o

Begin the main function:

Boolean for undirected graph and verbose printing set to false initially

Set the standard input variable as well as standard output variable

While reading user input with the previously defined options:

Begin a switch statement to evaluate the option chosen

If h was specified for help:

Use the message function to print proper usage

Return 0 to end the program

If u was specified because the graph is undirected:

Set the variable for undirected to true

If v was specified and all paths must be print:

Set verbose printing to true

If i for an input file:

Take in file input using optarg and file open in the read option

If the input is not valid, use the default standard input

If o for an output file:

Take in file for output using optarg and file open in the write option

If the input is not valid, use the default standard output

The default case is printing the help message and ending the program

Set a 32 bit unsigned integer as the number of locations

If the value scanned for the number of locations is not valid (is not 1 or not within range):
    Print the error that there is a malformed number of vertices
    Return 0 to end the program
Set up the graph using the number of locations and undirected value
Set a character array for a location name with the capacity for 1024 characters
Set another character array but to hold “number of locations” amount of strings
For the number of locations specified:
    Get the user input and store it as the name with a capacity of 1024
    Get rid of the null character
    Store the name in the locations array
Set three 32 bit unsigned integers to i, j, and k so that i is the first vertex in a tuple, k is the second, and k is the weight of the edge between them
While we read these values from the user input/files and we have not hit the end of the file:
    If the values are valid:
        Add the respective k weight to the edge i, j
    Otherwise, print the error message, free all memory, and end program
If the values of i, j, and k are valid and the number of locations in the graph is valid:
    Set a 32 bit unsigned integer as the recursion count
    Create paths for the current and shortest path
    Perform a DFS
    If the length of the shortest path if 0, there were no hamiltonian paths found
        End the program and free all memory
    Print the path, path length, and recursion
    Free memory dedicated to the paths
Otherwise, print that there is nowhere to go
Delete memory allocated to the graph, locations, etc.

vertices.h:
Provided by the professor
Declares the lowest and highest possible value for the number of vertices

graph.h:
Provided by the professor
Declares all the functions for graph.c and the structure for graph

graph.c:
Include all header files such as graph, path, stack, and vertices with necessary standard headers
Define the Graph structure (given struct Graph by professor):
    Vertices are 32 bit unsigned integers, the 2D matrix is made up of

32 bit unsigned integers, undirected and visited vertices are boolean

Graph create (given by professor):

Allocate memory to graph G

Set vertices and undirected components of the graph

Return the graph G

Graph delete (given by professor):

Free the memory allocated to graph G and set the pointer equal to null to confirm that there is no memory allocated

Graph vertices:

Return the number of vertices in graph G

Graph add edge:

If the graph is valid:

If the two vertices are within bounds:

Set the value of the matrix at the point (first, second) to the weight

If the graph is undirected:

Set the value of the matrix at the point (second, first) to the weight

Return true to indicate success

Otherwise, return false to indicate failure

If graph is not valid, return false to indicate failure

Graph has edge:

If the vertices are within bounds and edge value (found in the matrix at point (first matrix, second matrix)) is greater than 0:

Return true to indicate success

Otherwise, return false to indicate failure

Graph edge weight:

If vertices are not within bounds:

Return 0 to indicate fail

Otherwise, return the value of the weight (which is the matrix at point (first,second))

Graph visited:

If the vertex has been visited, return true

Otherwise, return false

Graph mark visited:

If the vertex is within bounds:

Set the value of the vertex's visit status in Graph structure to true

Graph mark unvisited:

If the vertex is within bounds:

Set the value of the vertex's visit status in Graph structure to false

Graph print:

For columns, as long as columns are less than the number of vertices in the graph:

For rows, as long as rows are less than the number of vertices in the graph:

Print the value of the edge weight
Print next line

stack.h:

Provided by the professor
Declares all the functions for stack.c and the structure for stack

stack.c:

Include all header files such as graph, path, stack, and vertices with necessary standard headers
Define the Stack structure (given struct by professor):
Top of the stack and capacity of the stack are 32 bit unsigned integers,
the array of items contains numbers that are 32 bit unsigned integers
Stack create (given by professor):
Allocate memory to the stack
If stack was successfully made:
Set the top of the stack to 0, capacity to the given value
Allocate memory to the items array
If the items were not able to be made correctly:
Free the memory allocated to the stack and set it to null
Return the stack
Stack delete (given by the professor):
If the stack and items array exist:
Free the memory allocated to both and set the pointer to the stack to null
Stack size:
Return the top of the stack
Stack empty:
If the top of the stack is 0 (which means the stack is empty):
Return true
Otherwise, return false
Stack full:
If the stack size is at capacity:
Return true
Otherwise, return false
Stack push:
If the stack is full:
Return false because you cannot add to a full stack
Otherwise:
Store the top element of the items array to the given value
Increment top of stack by 1
Return true

Stack pop:
    If the stack is empty:
        Return false because you cannot pop from an empty stack
    Otherwise:
        Decrement the top
        Store the top of the stack item in the given pointer
        Return true to indicate success
Stack peek:
    If the stack is empty:
        Return false because there is nothing at the top
    Otherwise:
        Store the value at the top of the stack in the given variable
        Return true
Stack copy:
    If the capacity of both the array and the copy array are the same:
        For the index value, as long as it is less than the capacity:
            Set the item at the index value of the copy
            stack equal to that of the array we are trying to copy
        Set the copy array's top of stack equal to the one of the stack we are
        trying to copy (so they have the same top of stack)
Stack print (given by the professor):
    For the index value, as long as it is less than the top of the stack:
        Print the cities
        If the next value is not the top of the stack:
            Print the arrow
    Print next line

path.h:
Provided by the professor
Declares all the functions for path.c and the structure for path

path.c:
Include all header files such as graph, path, stack, and vertices with necessary standard headers
Define the Path structure:
    Vertices array is a stack, and length of the path is a 32 bit unsigned integer
Path create:
    Allocate memory for the path
    If p was assembled correctly:
        Create a stack for vertices
        Set the length of the path to start at 0

Return the path

Path delete:

If the path and vertices stack exist:

Delete the stack and free the memory allocated to the path

Set the pointer to the path to null to make sure the content is cleared

Path push vertex:

Set a 32 bit unsigned integer variable as the top of stack

Peek into the path vertices to get the top of the stack value, assign it to above variable

If pushing the vertex to the path was successful:

If the vertex is not the top of stack:

Increment the path length by the weight

connecting top to the vertex

Return true to indicate success in pushing to stack

Otherwise:

Return false to indicate fail to push into stack

Path pop vertex:

Set a 32 bit unsigned integer variable as top of stack

If popping vertex out of the path vertices was successful:

Peek into the top of the stack and store the top of stack

If the vertex was not pointing to the top of stack:

Decrement length by the edge connecting top to the vertex

Return true to indicate successful pop

Otherwise:

Return false to indicate fail to push into stack

Path vertices:

Return the stack size of the path vertices

Path length:

Return the path length

Path copy:

If the given paths are valid:

Copy stack of first into second

Set their lengths equal to each other

Path print:

If the path is valid:

Print “Path:” to the outfile

Print the stack of vertices to the outfile