Devan O'Boyle

Assignment 4 Design

Purpose: In this lab, given a graph of vertices, the program will find the shortest Hamiltonian path, meaning that each node is visited only once, that connects all of the vertices. Along with this, the path will also be displayed showing the order in which each node was reached.

Throughout the program, there are a few values that should be kept track of: the number of vertices, the origin vertex, the length of the shortest path along with booleans like verbose and

In order to implement this program, a few structures will be needed to represent the graph, the current path, and the stack.

For the graph, there will be several functions that will be used in order to set and check the edge of the graph, as well as to set and check whether or not a vertex has been visited or not.

The graph is designed using a 2d matrix where each element from indexes [i][j] represents a path that goes from vertices i to j. The value that each element holds is known as the edge weight, which is the same as the distance of the path between i to j. Therefore in order to find the total length of the path, each path from i to j will add their edge weights. In order to keep track of which vertices have been visited yet, a boolean array will be used to keep track of whether or not each vertex has been visited or not.

Pseudocode:
create graph:
allocate memory for the graph
initialize variables
initialize boolean array to keep track of visited vertices
initialize 2d matrix

delete graph:
free memory from the graph
delete the graph

get vertices:
return the number of vertices in the graph

add edge:
take in two vertices and their weight
check if vertices are within bounds (each vertex should be greater than 0 and less than the max amount of vertices)
set edge weight using the vertices as indexes for the graph matrix

has edge:

takes in two vertices
checks if vertices are within bounds
if the edge weight at the matrix index of the two vertices is greater than zero, then there is an edge

edge weight:
take in two vertices
return the weight at the matrix index of the two vertices

visited:
take in a vertex
check the vertex index in the visited array to see if it is true or false
if true then the vertex has been visited
otherwise it hasn't

mark visited:
take in a vertex
set the vertex index in the visited array to true to indicate that it has been visited

mark unvisited:
take in a vertex
set the vertex index in the visited array to false to indicate that it is now unvisited

print graph:
iterate through the matrix
print out each edge weight value in the matrix

*note that print graph should only be used for testing purposes

The stack function is very similar to the one from assignment 3. In it, there are functions that check the size of the stack, whether or not it is empty or full, and also functions that push, pop, copy, or print from the stack.

Pseudocode:
create stack:
allocate memory for the stack
initialize variables
allocate memory for the items in the stack

delete stack:
free allocated memory for the stack and the items
delete the stack by setting it to null

size:

return the size of the stack, which is the same as the top index

empty:
if the top index is 0, then the stack is considered empty and should return true
otherwise the stack isn't empty and should return false

full:
if the top index is equal to the capacity of the stack, then the stack is full

push:
take in an integer
check if the stack is full
if not then set the new integer element to the top of the stack and increment the top index of the stack

pop:
take in a pointer
check if the stack is empty
if not then decrement the top index of the stack and set the popped value to the pointer

peek:
take in a pointer
checks if the stack is empty
if not then set the topmost value in the stack to the pointer

copy:
take in two stacks: a destination and a source
iterate through and set each of the source's items to the items in the destination
set the source's top to the destination's top
check that the tops are the same to make sure that the stack was successfully copied

print:
take in an outfile and the array that stores all of the vertex names also known as cities
iterate through the items in the stack
print each city of index item[i] from the stack
print an arrow after each city so long as the iterated value, i, is not the last index in items

For the path, it will make use of the stack function. In the constructor a stack will be created to hold a set of vertices from the current path. While the program is looking for a path, new vertices will be pushed onto the stack. However once a Hamiltonian path is found, vertices will begin to get popped in order to create room on the stack to find other potential paths.

Pseudocode:

create path:
allocate memory for the path
initialize the stack and path length

delete path:
delete that stack by calling the stack delete function
free the allocated memory for the path
delete the path by setting it to null

push vertex:
take in the vertex and graph
initialize an integer and use it as a parameter to peek the stack
then push the vertex to the stack
if the push was successful, then add the edge weight of the graph going from the peeked value and the vertex by calling the edge weight function from the graph

pop vertex:
take in a pointer and graph
initialize an integer
pop from the stack and by using the pointer as a parameter for the popped value
peek the stack using the initialized integer
if the pop was successful then subtract the edge weight of the graph going from the peeked value and the popped vertex by calling the edge weight function from the graph

vertices:
return the size of the stack since it is the same as the number of vertices currently on the stack

length:
return the length of the path

copy:
take in a destination path and a source path
set the destination path length to the source path length
call the stack copy function with the destination's and source's corresponding stacks to copy the source's onto the destination's

print:
print the word path
call the stack print function to print out the cities

After we have created all of these structures, we can now implement the algorithm used to find the shortest Hamiltonian path. In this case, we will be using depth-first search function via recursion which will set the current vertex as visited and then iterate through all of the edges

starting at the current vertex until it finds one that has not been visited. Once it finds an edge that hasn't been visited, the function will call itself using the graph and current vertex as the new parameters. This means that the new vertex will also be set to visited. This recursion will continue until the loop runs through and finds no unvisited edges. Once this happens, after the loop the current vertex will be set back to unvisited, allowing all of the other recursive calls to complete their loops. It should also be noted that there will be a counter in the function, so that each recursive call counts how many times the function was called on. Along with this, the path should push each vertex to it's stack before each recursive call.

After a path has been found, if it is the first path then it should be set to the shortest path. Otherwise, it should be compared to the current shortest path. If the path is shorter, then the path should be copied and set to be the new shortest path.

All of this will be looped through for each of the different starting points of each vertex.

Pseudocode:
dfs:
take in: the graph, the origin vertex (0), current path, shortest path, cities array, number of vertices, number of recursive calls, outfile
call the mark visited function from the graph using the given graph and vertex as parameters
push the vertex onto the path
increment the number of recursive calls
iterate through the vertices in the graph by calling the graph's vertices function using an iterated value: w
if the graph has an edge at the vertex going to w and the graph hasn't visited vertex w yet, then check if the current path is shorter than the shortest path or if the shortest path's length is zero
if so, then dfs should call itself with all of the same parameters except by using vertex w in place of the origin vertex
after the iteration of the graph's vertices
check if the current path is Hamiltonian
if so then push the origin vertex onto the path
then check if the current path is shorter than the smallest path or check if the shortest path's length is still 0
  if either of those are true, set the current path as the new shortest path by copying the current path to the shortest path
  if verbose printing is true, print out the length of the path along with the path itself
after the last two of these conditionals, pop from the origin vertex from the stack
after all of the conditionals, pop from the stack and mark the current vertex as unvisited

For taking in input, the program should either read from a file, or read the input line by line. The first line of the input should indicate the number of vertices. The next few vertices number of lines will contain the names of each of the vertices. The next vertices number of lines will contain the edges of the graph.

The names of the locations should be obtained by iterating through a for loop for the number of vertices and adding the names to an array of vertices. As for the edges, they should be looped through until the end of the file is hit and added to the graph as it goes.

The user can also type in a variety of commands in the command line when they run the file.
-h allows the user to read the help manual which will print and stop the program if selected
-v enables verbose printing which prints all of the shortest Hamiltonian paths as they are found rather than only the shortest one found
-u enables undirected graphs, meaning that each pair of vertices point to each other, so i->j and j<-i, this modification takes place in the add edge function of graphs
-i allows the user to specify an input file to read from, without this command the program defaults to reading input from the command line
-o allows the user to specify an output file to write to, without this command the program defaults to printing output to the command line

these options will be parsed through using getopt and cases

at the end of the program the shortest path length, the path itself, and the total amount of recursive calls should be printed