Assignment 4 - Perambulations of Denver Long Design

Sean Carlyle

October 19 2021

1 Introduction

This is my Assignment 4, Perambulations of Denver Long Design document. It is a program made out of functions that create ADT's or abstract data types for graphs, stacks, and paths. It then uses these functions to perform depth-first search to find the shortest path between vertices that is Hamiltonian. A Hamiltonian path is one where every vertex is visited only once except the first and last vertex which are the same. It finds the shortest path by comparing lengths of path which are determined by the length of edges between vertices. The program requires a map to input either through an infile or through stdin. It accepts commands through a command-line environment. -h prints out a help message. -v enables verbose printing, which prints out all Hamiltonian paths instead of just the shortest one. -u specifies the graph to be undirected, where edges travel both ways. -i infile specifies the input file and -o specifies the outfile. By default these are stdin and sdout respectively. The program prints out the shortest path along with it's length and the number of recursive calls to the outfile. It can also print all Hamiltonian paths found if verbose printing is active.

2 Pseudocode with Explanation

2.1 tsp.c

tsp.c is where the main function and depth first search function are in. This file does puts together all the helper functions to find all Hamiltonian paths and find the shortest. main handles all user input selections. It creates and allocates memory to all the structure ADT's we need to perform the search. It also free's these structures once we are done using them. dfs is a function that finds the shortest Hamiltonian path by finding each path one at a time using recursion. It starts marking vertices and going down them adding them to the path until every vertex has been visited. Once this is done, it sees whether there is an edge from the end of the path to the start, and if there is we know we have found a Hamiltonian path. It then checks if this path is shorter than the current shortest path. If it is we swap them. To make the program more efficient, we reject paths that are greater than the shortest path before we even go down through all of them. Once a path is found, dfs travels back up the

path unpopping and unvisiting vertices until it finds another edge to go down from. From this point it will keep going until all paths (except those we've deemed to long too early) have been found.

2.1.1 main function

Use getopt to get command line input with switch case
- If -i is selected store optarg file in infile and set to read
- If -o is selected store optarg file in infile and set to write
- If -h is selected print help message
- If -u is chosen set graph to be undirected
- if -v is chosen set all paths to be printed
- Get first line of infile as number of vertices
- Create graph
- Allocate memory to an array of strings called cities
- Add city names to city array of strings from infile
- Add edges to graph from infile
- Create two paths called shortest and current path
- Call DFS to find shortest path
- Print shortest path to outfile
- Print total recursive calls
- Free all allocated memory
- Delete graphs and paths

2.1.2 DFS function

Accepts graph and vertex
- Make recursive counter go up
- Mark vertex as visited
- Push vertex to current path
- If current path is shorter than longer path and the shortest path has been set
  - Mark vertex unvisited
  - Pop vertex
  - Return
- indent If all vertices have been visited and there is an edge at the last vertex to the first
  - Mark the first vertex as unvisited
- If path is complete
  - Print path if verbose is on
  - Swap current path with shorter if it is shorter
- Unvisit vertex
- Pop vertex from current path

2.2 graph.c

Graph.c is a helper file that creates the graph ADT we need to use in our dfs function. In graph.c we create a graph structure that holds number of vertices, a Boolean telling whether graph is undirected, a Boolean array of visited vertices, and an adjacency matrix. We then create various functions to edit the graph in ways that would be helpful. For example, we create a function that finds whether a two vertices create an edge in the graph. This is useful for checking if there are edges to traverse in our dfs function. All of these functions help in various ways. It also makes sense to have a graph ADT because it is natural that we need to store edges and vertices somewhere from the input.

2.2.1 graph create function

Accepts parameters number of vertices and whether graph is directed
    Allocate memory to the graph
    Set number vertices to parameter vertices
    Set Boolean undirected to parameter directed.
    Return the graph

2.2.2 graph delete function

Accepts pointer address of graph as parameter
    Free graph memory
    Point the memory to NULL
    Return

2.2.3 graph vertices function

Accepts graph as parameter
    Return the number of vertices in the graph

2.2.4 graph add edge function

Accepts parameters of the graph, i, j, k where ij is the matrix position of the edge and k is weight of edge
    If i or j are not in the bounds of the graph
        Return false
    Else
        Add k to value matrix position ij.
        If graph is undirected
            Add k to value matrix position ji.
        Return true

2.2.5 graph has edge function

Accepts parameters graph and i,j where ij is the matrix position of the edge.
    If i,j are not in bounds or their weight equals 0
        Return false.
    Else
        Return true.

2.2.6 graph edge weight function

Accepts parameters graph and i,j where ij is the matrix position of the edge.
    If edge doesn't exist or ij are not in bounds
        Return 0.
    Else
        Return the weight of edge at ij.

2.2.7 graph visited function

Accepts parameters graph and v for the vertex.
    If vertex visited
        Return true
    Else
        Return false

2.2.8 graph mark visited function

Accepts parameters graph and v for the vertex.
    If v is in bounds
        Mark v as visited

2.2.9 graph mark unvisited function

Accepts parameters graph and v for the vertex.
    If v is in bounds
        Mark v as unvisited

2.2.10 graph mark print function

Use a double for loop to print out entire matrix of graph

2.3 stack.c

stack.c is a helper file which creates the stack ADT, a structure that holds index of next empty slot in stack, capacity of the stack and an array of uint32_t items. We need to create a stack ADT because it works in unison with the path ADT to create a path. A stack is

almost like a stack of pancakes where only the top can be pulled off. We need a stack so we can store vertices in order, almost like we are walking down a path. So this function creates a stack that will help us in creating a path which will help us in doing a depth first search.

2.3.1 stack create function

Accepts the wanted capacity of stack as uint32_t
    Allocate memory to stack
    If allocating worked properly
        Set stack top of stack to 0
        Set stack capacity to wanted capacity
        Dynamically allocate array of items
        If items allocated incorrectly
            Free stack and set to null
    Return stack

2.3.2 stack delete function

Accepts the stack pointer address as a parameter
    If both stack and stack items exist
        Free them both
        Set stack pointer to null
    Return

2.3.3 stack empty function

Accepts stack pointer as parameter
    If top of stack is index 0
        Return true
    Else
        Return false

2.3.4 stack full function

Accepts stack pointer as parameter
    If top of stacks index is above capacity
        Return true
    Else
        Return false

2.3.5 stack size function

Accepts stack pointer as parameter
    Return top of stack index minus one

2.3.6 stack push function

Accepts stack and a uint32_t item
    If stack is full
        Return false
    Else
        Put x on top of stack
        Make top of stack plus one
        Return true

2.3.7 stack pop function

Accepts stack and a uint32_t item
    If stack is empty
        Return false
    Else
        Put top of stack value in x
        Make top of stack minus one
        Return true

2.3.8 stack peek function

Accepts stack and a uint32_t item
    If stack is empty
        Return false
    Else
        Put top of stack value in x
        Return true

2.3.9 stack copy function

Accepts destination and source stack
    copy contents of source stack to destination stack

2.3.10 stack print function

Accepts a stack pointer, outfile pointer, and character array pointer
    For each index in the stack
        print city names to outfile
        If the next index is not the top
            print → to outfile
    Print newline

2.4 path.c

path.c is a helper file that creates the path ADT. A path is a structure with a stack of vertices and the length of the path held in it. The path uses a stack to track what is in the path. It also adds the weight of the edge between every vertex in the path to calculate the path length. We need paths in our dfs function so that we can track which path has a shortest length. We have to create functions that tell us how big the path is, and to copy the path in order to perform dfs. For example, we have to copy the current path to the shortest path if it is actually shorter.

2.4.1 path create function

Accepts no parameters
    Allocate memory to path
    Set vertices to a stack that can hold 26 vertices.
    Make length equal 0
    Return path

2.4.2 path delete function

Accepts the path pointer as parameter
    Free path
    Set pointer to null
    Return

2.4.3 path push vertex function

Accepts path pointer, vertex being pushed, and a graph pointer
    Push vertex onto paths stack
    Check whether path as any vertices
    If push returned false
        Return false
    else
        If there are vertices, increase path length by vertex and the vertex that was on the top of the stack
        Return true

2.4.4 path pop vertex function

Accepts path pointer, pointer to uint32_t variable, and Graph pointer
    Pop top of path stack into the variable
    If push returned false
        Return false
    else

If path has more than 1 vertex, Decrease weight by the edge weight connecting the new top vertex and the popped vertex

Return true

2.4.5 path vertices function

Accepts path pointer parameter

Return stack size of stack in path

2.4.6 path length function

Accepts path pointer parameter

Return length in path

2.4.7 path copy function

Accepts destination path pointer, and source path pointer

Copy stack to destination path from source path

Copy length to destination path from source path

2.4.8 path print function

Accepts path pointer, outfile pointer, and character array pointer

print length of path to outfile

print stack using print stack function to outfile

3 Structure of program

The program is structures in a way so that there are three ADT's created in separate files, then they are all brought together in the tsp.c file. graph.c creates the graph needed for storing vertices and their edges. stack.c creates the stack needed by path.c to store vertices in the path. path.c creates the path needed by dfs to track which path has the shortest length. This is all brought together by tsp.c which contains the main function and calls all separate ADT's and functions after getting user input. tsp.c also prints out the final results.