Assignment 4 - The Perambulations of Denver Long

Teresa Joseph

CSE 13S13S - Professor Long

Fall 2021 - RD October 21/ FD October 24

Purpose

Assignment 4 is the implementation of Hamiltonian paths, where two vertices are connected by a path each vertex is only visited once. These paths can be directed or undirected. Three files called graph.c, path.c, and stack.c will be used to create and manipulate the paths. tsp.c can test these three files and take on command line arguments h, v, u, i, and o for testing purposes.

Breakdown of Files

- graph.h specifies the interface to the graph ADT
- graph.c implements the graph ADT
- path.h specifies the interface to the path ADT
- path.c implements the path ADT
- stack.h specifies the interface to the stack ADT
- stack.c implements the stack ADT
- tsp.c contains main() and may contain any other necessary functions
    - Takes command line options h (help message), v (verbose printing), u (specifies graph is undirected), i infile (specifies input file), o outfile (specifies output file)
- vertices.h defines macros regarding vertices
- Makefile contains commands for executing, cleaning, and formatting
- README.md describes program and Makefile
- DESIGN.pdf includes design/design process, pseudocode, and explanations

Pseudocode

graph.c

define Graph structure (with vertices int, undirected bool, visited bool, and matrix int)

*graph_create

use calloc to initialize graph memory (calling it G, a pointer) with 0's

G to vertices = vertices

```
        G to undirected = undirected
        return G
graph_delete
        free pointer G
        set G to null/none
        return nothing (graph_delete is void type)
graph_vertices
        G to vertices = vertices
        return vertices
graph_add_edge
        matrix[i][j] = k (says i by j matrix has a weight of k)
        if G is undirected:
                matrix[j][i] also = k
        if START_VERTEX <= i, j <= VERTICES (&& edges are successfully added):
                return true
        else:
                return false
graph_has_edge
        if START_VERTEX <= i, j <= VERTICES && edge from i to j > 0:
                return true
        else:
                return false
graph_edge_weight
        if START_VERTEX <= i, j <= VERTICES:
                matrix[i][j] = k
                if k > 0:
                        return k
                else:
                        return 0
        else:
                return 0
```

graph_visited

if v was visited:

return true

else:

return false

graph_mark_vistited

if START_VERTEX <= v <= VERTICES:

mark v as visited with visited[v] = true

graph_mark_unvisited

if START_VERTEX <= v <= VERTICES:

mark v as unvisited with visited[v] = false

graph_print

check that i and j are in bounds

check that k exists for i and j (0 or not)

if any of the checks fail, print an error message and identify which check

path.c

define Path structure (*vertices Stack and length int)

*path_create

call stack_create and set it = to vertices

length = 0

path_delete

free pointer p

set p to null/none

return nothing (path_delete is void type)

path_push_vertex

push v onto p

if vertex was pushed successfully:

return true

else:

return false

path_pop_vertex

pop vertice and pass it into pointer v (length of path should decrease by weight k
of top and popped vertex)
if vertex was popped successfully:
return true
else:
return false
path_vertices
return p->vertices
path_length
return p->length
path_copy
copy the vertices stack
copy the length
make dst a copy of the source path src
path_print
for each iter in the stack (from 0 to path length):
print iter (use call to stack_print)
// stack content should correlate with path locations
// I will think through this as I complete the rest

stack.c

define Stack structure (with top int, capacity int, and array of ints items)
*stack_create
use malloc to initialize stack memory (calling it s, a pointer)
if s:
s to the top of s = 0
s to the capacity = 0
s to items = calloc(capacity, the size of 32 bits) (this allocates enough
memory for items given s's capacity)
if not s has items:
free pointer s
set pointer s to null/none

```
        return s
stack_delete
        is pointer s && pointer s to items:
                free what pointer s to items is
                free pointer s
                set pointer s to null/none
        return nothing (stack_delete is void type)
stack_empty
        for i starting at 0 until the top of the stack, incrementing i by 1:
                if s to items[i] doesn’t = 0:
                        return false
        return true
stack_full
        if s top = s capacity:
                return true
        else:
                return false
stack_size
        return s top
stack_push
        if stack is at capacity:
                return false
        else:
                push x to stack
                return true
stack_pop
        if stack is empty:
                return false
        else:
                pop x from stack
                 set the value in the memory that x points to as popped item (something
```

like *x = s->items [s->top] from the assignment document)

return true

stack_peek

if stack is empty:

return false

else:

set the value in the memory that x points to as peeked item (which will be the top of the stack, so *x = s->items [s->top] from the asgn doc)

return true

stack_copy

for i starting at 0 until the top of the stack, incrementing i by 1:

push items[i] to dst

return nothing (stack_copy is void type)

stack_print

for i starting at 0 until the top of the stack, incrementing i by 1:

print outfile and cities[stack items[i]]

if i+1 isn't the top of the stack:

print outfile and "->"

print outfile and new line

<u>tsp.c</u>

define the command line options h, v, u, i, and o

main function

use getopt to parse through the options

use switch and cases for each of the five options

if h, print help message

if v, print all Hamiltonian paths found and total number of calls to dfs

if u, make graph undirected

if i, specify input file path containing cities and edges of graph

if not specified, default input = stdin

if o, specify output file path to print to

if not specified, default output = stdout scan input for number of vertices/cities with fscanf

print error if number is out of bounds (0 and 26 as provided in vertices.h)

use fgets to read lines (each line represents name of city) and add to array

if line is malformed, print error and exit

create graph

if -u is an inputted option, make graph undirected

add each edge to graph

again, if line is malformed, print error and exit

create a path for tracking current traveled path

create a path for tracking the shortest path

use dfs (from assignment document) to find shortest path

will use recursion

print length of shortest path, path itself, and number of calls to dfs after the search

if -v is an inputted option, print all paths and their stats as they are found

Overall Description

graph.c will create the needed functions for initializing the graph. path.c will do the same for creating the path, and stack.c will do the same for creating the stack. Each file will make use of a structure that defines variables and pointers needed to connect the functions. All files are of type uint32_t, bool, or void, which will return a 32-bit integers, true/false, or nothing respectively. All functions in each of these files will be used to implement the main function in tsp.c. This is the main test harness, the file that will take on command line arguments and simulate Hamiltonian paths based on combinations of arguments.

Notes

- The pseudocodes provided in the assignment document will be mostly used (as I finalize my code, I will specify and cite which ones I end up using)
- Header files will be included as necessary (vertices.h in graph.c, all headers in tsp.c, etc.)
    - Header files are taken from the resources repository