

Deeraj Gurram

dgurram@ucsc.edu

21 October 2021

CSE 13S Fall 2021

Assignment 4 - The Perambulations of Denver Long

Design Document

Description of the Program:

This assignment uses 3 separate programs to calculate the shortest path from point A to point B. The first program, graph.c contains a variety of functions that deal with graph creation, deletion, specification of directed vs undirected vertices, and mapping of edges. The second program, stack.c contains functions that are quite similar to those found in graph.c, but deal with a stack. The file contains functions for stack creation, deletion, checking for full or emptiness, checking the size of the stack, returning the last element, pushing more elements into a stack, and printing the contents of the stack. The last program file, path.c, contains functions that create and delete a path, adding vertices to the path, removing a vertex from the path, finding the length and number of vertices of the path, and printing the entirety of the path. Also included is a self-created test harness that takes in command line input and changes various variables. By using the functions contained in these program files, I will be able to map out a path and go through the different Hamiltonian paths to choose the shortest one.

Deliverables:

1. graph.c

- This file contains the following functions:
 - i. Graph *graph_create() is the constructor for a graph.
 - ii. graph_delete() is the destructor for the graph
 - iii. graph_vertices() returns the number of vertices in the graph
 - iv. graph_add_edge() adds an edge assuming the vertices are within bounds.
 - v. graph_has_edge() returns true if there exists an edge between two vertices
 - vi. graph_edge_weight() returns the weight of the edge between two vertices
 - vii. graph_visited() checks if a vertex has been visited
 - viii. graph_mark_visited marks a vertex as visited
 - ix. graph_mark_unvisited() marks a vertex as unvisited
 - x. graph_print() debugs the graph ADT and makes sure it works as expected

2. graph.h

- This file specifies the interface to graph.c

3. stack.c

- This file contains the following functions:
 - i. Stack *stack_create() which is the constructor for a stack
 - ii. stack_delete() is the destructor function for a stack
 - iii. stack_empty() checks if the stack is empty
 - iv. stack_full() checks if the stack is full
 - v. stack_size() returns the number of items in the stack

- vi. `stack_push()` pushes an item to the top of the stack
- vii. `stack_pop()` pops an item off of the specified stack
- viii. `stack_peek()` checks the top of the stack without popping it
- ix. `stack_copy()` copies the top of the stack to the destination stack
- x. `stack_print()` prints out the contents of the stack to the outfile

4. `stack.h`

- This file specifies the interface to `stack.c`

5. `path.c`

- This file contains the following functions:
 - i. `Path *path_create()` is the constructor for the path
 - ii. `Path_delete` is the destructor for the path
 - iii. `path_push_vertex()` pushes a vertex onto the path, with all corresponding attributes getting updated
 - iv. `path_pop_vertex()` pops the vertices stack and sets the pointer equal to the popped value
 - v. `path_vertices()` returns the number of vertices in the path
 - vi. `path_length()` returns the length of the path
 - vii. `path_copy()` copies the source path to the destination path along with the vertices stack and source path length
 - viii. `path_print()` prints the path to the outfile

6. `path.h`

- This file specifies the interface to `path.c`

7. `tsp.c`

- This file holds the main test harness and reads the command line inputs that specify the starting value for the stack

8. `vertices.h`

- Defines the macros regarding vertices

9. Makefile

- Builds the tsp executable, removes files that are compiler generated, and formats all source code including the header files

10. README.md

- Gives a brief description of the program and Makefile and also lists the command line options that the program accepts. Also notes any errors or bugs found in the code and gives credit to sources of code or help.

11. DESIGN.pdf

- Provides a description of the assignment and program files, the deliverables, and pseudocode for each of the .c files. Also describes the design and design process for the program.

Pseudocode:

graph.c

Define the Graph struct

Struct Graph

Initialize a variable vertices

Initialize a boolean undirected

Initialize a boolean visited with argument VERTICES

Initialize a variable matrix[VERTICES][VERTICES]

Graph *graph_create(vertices, undirected)

Allocate enough memory for the graph and set it equal to the graph's place in memory

Dereference the struct vertices and set it equal to the argument vertices

Dereference the struct undirected and set it equal to the argument undirected

Return the graph

graph_delete(pointer to graph *G)

Free the memory that was allocated to *G to free all of the memory used by the constructor

Set *G equal to NULL

Return

graph_vertices(graph *G)

Return G-> vertices

graph_add_edge(*G, i, j, k)

Check if the vertices are greater than 0 and less than VERTICES and the edge weights are greater than 0

Set $G \rightarrow \text{matrix}[i][j]$ equal to weight k

Check if graph is undirected

Add a $\text{matrix}[j][i]$ of weight k

Return boolean true

Else return boolean false

graph_has_edge(*G, i, j)

Check if the vertices are greater than 0 and less than VERTICES and the edge weights are greater than 0

Return the boolean true

Else return the boolean false

graph_edge_weight(*G, i, j)

Check if the vertices are greater than 0 and less than VERTICES and the edge weights are greater than 0

Return the edge weight of $\text{matrix}[i][j]$

Else return 0

graph_visited(*G, v)

Check if vertex v is inside $G \rightarrow \text{vertices}$ and is equal to true

Return boolean true

Else return boolean false

graph_mark_visited(*G, v)

Check if v is greater than 0 and less than VERTICES

Set the vertex v inside G->vertices equal to true

graph_mark_unvisited(*G, v)

Check if v is greater than 0 and less than VERTICES

Set the vertex v inside G->vertices equal to false

graph_print(*G)

Initialize a variable rows

Initialize a variable columns

Create a for loop to go through the number of vertices, set row equal to 0, row must be

less than G->vertices, increment row by 1

Create for loop to go through the number of vertices, set column equal to 0,

column must be less than G->vertices, increment column by 1

Print out the current matrix[row][column]

stack.c

Create a struct for Stack

struct Stack

Initialize a variable top

Initialize a variable capacity

Initialize a pointer to items

Stack *stack_create(capacity)

Allocate enough memory to the stack and set it equal to the stack's place in memory

Check if s is true

Dereference the struct top and set it equal to 0

Dereference the struct capacity and set it equal to the argument capacity

Dereference the struct items and set it equal to a dynamically allocated memory
of size capacity to a 32 bit integer

Check if the dereferenced struct items is not true

Free the allocated memory of stack s

Set stack s equal to NULL

Return the stack s

stack_delete(**s)

Check if *s and the dereferenced struct items are true

Free the memory allocation of the dereferenced struct items

Free the memory allocation of *s

Set *s equal to NULL

Return

stack_empty(*s)

Check if s->top is 0

Return boolean true

Else return boolean false

stack_full(*s)

Check if s->top is equal to s->capacity

Return boolean true

Else return boolean false

stack_size(*s)

Return s->top

stack_push(*s, x)

Check if s->top is equal to s->capacity

Return boolean false

Else

Set s->items[s->top] equal to x

Increment s->top by 1

Return boolean true

stack_pop(*s, *x)

Check if s->top is equal to 0

Return the boolean false

Else

Decrement s->top by 1

Set the pointer in memory of x equal to s->items of the top element

Return boolean true

stack_peek(*s, *x)

Check if s->top is equal to 0

Return the boolean false

Else

Set the pointer in memory of x equal to s->items of the top element - 1

Return boolean true

stack_copy(*dst, *src)

Create a for loop to go through src->capacity, set i equal to 0, where i is less than the src->capacity, increment in steps of 1

Set dst->items[i] equal to src->items[i]

Set the dst top equal to the src top

stack_print(*s, *outfile, *cities[])

Create a for loop to go through the s->top, set i equal to 0, where i is less than the s->top, increment in steps of 1

Print the outfile and the cities[s->items[i]]

Check if i + 1 doesn't equal s->top

Print outfile and "->"

Print outfile and newline

path.c

Create a struct Path

struct Path

Initialize a stack *vertices which holds the vertices comprising the path

Initialize a variable length which represents the length of the path

*path_create(length)

Allocate enough memory to the Path and set it equal to Path p's place in memory

Dereference the struct vertices and set it equal to stack_create of VERTICES

Dereference the length and set it equal to 0

Return the Path p

path_delete(**p)

Check if *p and the dereferenced struct vertices are true

Free the memory allocation of the dereferenced struct vertices

Free the memory allocation of *p

Set *p equal to NULL

Return

path_push_vertex(*p, v, *G)

Initialize a variable x

Check if stack_empty(p->vertices) is true

Check if stack_push(p vertices, v) is true and the graph has an edge from x to v

Set p length equal to p length + graph_edge_weight(G, x, v)

Return boolean true

Else

Return boolean false

path_pop_vertex(*p, *v, *G)

Initialize variable x

Initialize variable p_top

Check if stack_empty of p->vertices is true

Return the boolean false

Else

Set p_top equal to the stack size of p->vertices

Check if stack_peek(p vertices, memory address of x) is true

Stack pop x from p->vertices

Set the dereferenced value of v to x

Decrease p->length by the graph_edge_weight(G, p_top, x)

Return boolean true

Else

return boolean false

path_vertices(*p)

Return stack_size of p->vertices

path_length(*p)

Return p->length

path_copy(*dst, *src)

Call stack_copy on dst->vertices and src->vertices

Set dst->length equal to src->length

path_print(*p, *outfile, *cities[])

Call stack_print(p->vertices, outfile, cities)

tsp.c

Include the OPTIONS “hvuio” to run the various test cases

Initialize boolean verbose to false

Initialize boolean undirected to false

Initialize a File input to Null

Initialize a File output to Null

Create function main() with arguments argc, **argv

 Initialize an array cities[]

 Initialize an input_file to Null

 Initialize an output_file to Null

 Create a boolean no_input equal to true

 Create a boolean help equal to false

 Initialize a variable vertices to 0

 Initialize a variable atm to Null

 Initialize a variable i

 Initialize a variable j

 Initialize a variable k

 Initialize the variable opt and set it equal to 0

Create a while loop to go through all command line inputs that were specified, the loop should make sure (opt equals getopt()) which has arguments argc, argv, OPTIONS) and

that this whole thing doesn't equal -1. The -1 signals that all command line options have been read

Create a switch statement with argument opt to check for each case specified in

OPTIONS

Case 'h':

Set no_input to false

Set help to true

Case 'v':

Set no_input to false

Set boolean verbose to true

Case 'u':

Set no_input to false

Set boolean undirected to true

Case 'i':

Set no_input to false

Set variable input_file to command line input

Case 'o'

Set no_input to false

Set variable output_file equal to the command line input

Check if no_input is true

Print out the help message describing graph purpose and command line options

Check if help is true

Print out the help message describing graph purpose and command line options

Set input equal to a file read through of input_file

Initialize an array of size 1024 called buffer

Get the buffer using the input

Set variable vertices equal to buffer typecasted as an integer

Create a graph called G

Check if vertices is less than or equal to the VERTICES from vertices.h

Create a for loop to go through the number of vertices and append the location names to an array cities[], i starts at 0, make sure i is less than vertices and increment in steps of 1

Get the buffer from the input

Set variable atm equal to a string duplicate of buffer

Set cities[i] equal to atm

Create while loop to go through the rest of the input and add the graph vertices and edge weight to the graph G, the three variables i, j, and k must not be the

End of File

Add a new graph edge to G using i, j, and k

Else

Print an error message

Free the memory allocation of atm

Set atm equal to Null

Credit:

Used the provided C code and pseudocode from the asgn4.pdf from Dr. Long

Implemented TA Christian Ocon's examples to structure my main()

Copied the error messages and help messages from the resources repo

Used TA Omar's code for input scanning and the general structure of the input code

Used code from TA Eugene's section to get the vertex and edge weight from the input file