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 from a starting point to a destination, and navigate through the different vertices via the shortest path.

Deliverables:

1. graph.c

- This file contains the following functions:
    1. Graph *graph_create() is the constructor for a graph.
    2. graph_delete() is the destructor for the graph
    3. graph_vertices() returns the number of vertices in the graph
    4. graph_add_edge() adds an edge assuming the vertices are within bounds.
    5. graph_has_edge() returns true if there exists an edge between two vertices
    6. graph_edge_weight() returns the weight of the edge between two vertices
    7. graph_visited() checks if a vertex has been visited
    8. graph_mark_visited marks a vertex as visited
    9. graph_mark_unvisited() marks a vertex as unvisited
    10. 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:
        1. Stack *stack_create() which is the constructor for a stack
        2. stack_delete() is the destructor function for a stack
        3. stack_empty() checks if the stack is empty
        4. stack_full() checks if the stack is full
        5. stack_size() returns the number of items in the stack
        6. stack_push() pushes an item to the top of the stack
        7. stack_pop() pops an item off of the specified stack
        8. 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
```

<u>graph_delete(pointer to graph *G)</u>

```
    Free the memory that was allocated to *G to free all of the memory used by the
    constructor
    Set *G equal to NULL
    Return
```

<u>graph_vertices(graph *G)</u>

```
    Initialize a variable amount
    Set amount equal to G->vertices
    Return amount
```

<u>graph_add_edge(*G, i, j, k)</u>

```
    Add a matrix[i][j] of weight k
    Check if graph is undirected
        Add a matrix[j][i] of weight k
    Check if the vertices are within the bounds and the edge(s) were added
        Return boolean true
    Else return boolean false
```

<u>graph_has_edge(*G, i, j)</u>

```
    Check if i and j are within the bounds and if there’s an edge between i and j
        Return boolean true
    Else return boolean false
```

```
graph_edge_weight(*G, i, j)
    Check if i and j are within bounds and an edge exists
        Return weight of edge from i to j
    Else return 0

graph_visited(*G, v)
    Check if vertex is inside array visited
        Return boolean true
    Else return boolean false

graph_mark_visited(*G, v)
    Initialize static array visited
    Check if v is within bounds
        Add v to array visited

graph_mark_unvisited(*G, v)
    Create for loop to go through array visited, i equals 0, i must be less than length of visited, increment in steps of 1
        Check if visited[i] is the same as v
            Pop the element

graph_print(*G)
    Return the vertices of graph G along with the edges and weight between vertices
```

Depth First Search

```
procedure DFS(G, v)
    Add v to array visited
    Create for loop to loop through all the edges from v to w in the adjacent edges of G
        Check if the vertex w is unvisited
            Do a recursive call on DFS(G, w)
    Call graph_mark_unvisited on v
```

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
```

<u>stack_delete(**s)</u>

```
        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
```

<u>stack_empty(*s)</u>

```
        Check if stack is empty
            Return boolean true
        Else return boolean false
```

<u>stack_full(*s)</u>

```
        Check if stack is full
            Return boolean true
        Else return boolean false
```

<u>stack_size(*s)</u>

```
        Initialize variable size to 0
        Create for loop to go through *s, i starts at 0, i must be less than length of *s, increment
        in  steps of 1
            Set variable size equal to size + 1
```

```
    Return variable size

stack _push(*s, x)
    Check if stack_full(*s) returns false
        Push x to the stack
        Return boolean true
    Else return boolean false

stack_pop(*s, *x)
    Check if stack_empty(*s) returns false
        Pop the item at the top of stack s
        Set the pointer in memory of x equal to s->items of the top element
        Return boolean true
    Else return boolean false

stack_peek(*s, *x)
    Check if stack_empty(*s) returns false
        Set *x equal to s->top
        Return boolean true
    Else return boolean false

stack_copy(*dst, *src)
    Initialize a variable peek
    Create a for loop to go through the stack *src, set i equal to 0, where i is less than the
    size of the stack *src, increment in steps of 1
        Set *dst->items[i] equal to stack_peek(*src, peek)
```

<u>stack_print(*s, *outfile, *cities[])</u>

```
    Create a for loop to go through the stack *s, 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

<u>Create a struct Path</u>

```
struct Path
    Initialize a stack *vertices which holds the vertices comprising the path
    Initialize a variable length which represents the length of the path
```

<u>*path_create(length)</u>

```
    Allocate enough memory to the stack vertices and set it equal to vertices' place in
    memory
    Check if vertices 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 vertices
                Set stack vertices equal to NULL
        Return the stack vertices
        Dereference the struct length and set it equal to 0
```

<u>path_delete(**p)</u>

```
        Free the memory allocation of the dereferenced struct *vertices
        Free the memory allocation of *p
        Set *p equal to NULL
        Return
```

<u>path _push_vertex(*p, v, *G)</u>

```
        Push vertex v onto path *p
        Set struct length equal to struct length +  the weight of (stack_peek(*p), v)
        Check if the vertex was successfully pushed onto *p
                Return boolean true
        Else return boolean false
```

<u>path_pop_vertex(*p, *v, *G)</u>

```
        Set *v equal to *p->*vertices->items[vertices->top]
        Decrease *p->length by the weight of (stack_peek(*p->*vertices) and *v)
        Check if the vertex was successfully popped
                Return boolean true
        Else return boolean false
```

path_vertices(*p)

```
    Return *p->*vertices->*items
```

path_length(*p)

```
    Return *p->length
```

path_copy(*dst, *src)

```
    Create a for loop to go through the vertices of *src, i should equal 0, with i less than path_vertices(*src), increment in steps of 1
        Set *dsc->*vertices->*items[i] equal to *src->*vertices->items[i]
    Set *dsc->length equal to *src->length
```

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

```
    Create a for loop to go through the path *p, set i equal to 0, where i is less than the path_length(*p), increment in steps of 1
        Print the outfile and the cities[p->items[i]]
        Call stack_print(*p->*vertices)
    Print outfile and newline
```

tsp.c

```
Include the OPTIONS “hvuio” to run the various test cases
Create function main() with arguments argc, **argv
    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':
    Print out the help message describing graph purpose and command line options
Case 'v':
    Enable verbose printing
    Print out all hamiltonian paths found along with the number of recursive calls to dfs()
Case 'u':
    Set the graph's boolean undirected to true
Case 'i':
    Specifies the input file path containing the cities and edges of a graph.
    Default value is stdin
Case 'o'
    Specify the output file path to print to
    Default output is stdout
```

Credit:

I used the provided code and pseudocode from the asgn4.pdf from Dr. Long