DESIGN - Assignment 4

Brian Nguyen

October 21, 2021

1 Description

This program has a collection of files consisting of abstract data types and interfaces and functions to solve the Hamiltonian path problem using a Depth-first search recursive algorithm. It will compute and return the shortest and most optimal route or path possible to get from origin to destination as well as its length and amount of recursive calls (and possibly prints all path if prompted).

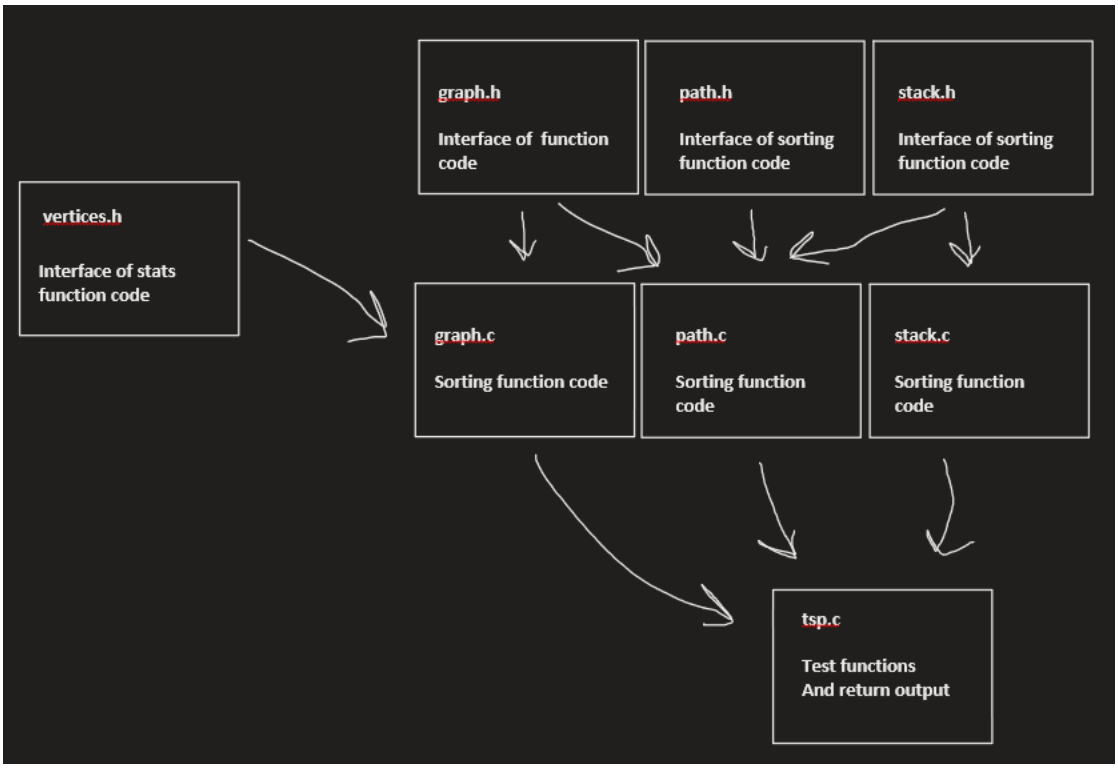
2 Files

1. graph.c
   - This source file contains the code for the graph abstract data type.
2. path.c
   - This source file contains the code for the path abstract data type.
3. stack.c
   - This source file contains the code for the stack abstract data type.
4. tsp.c
   - This source file contains the main() function that calls on all functions and find/return the best path.
5. graph.h
   - This header file contains the interface for the stack abstract data type.
6. path.h
   - This header file contains the interface for the path abstract data type.
7. stack.h

- This header file contains the interface for the stack abstract data type.

8. vertices.h
   - This header file contains the macros for vertices in the graph.
9. Makefile
   - This make file contains the code that builds and compiles the program to be run. It also cleans all compiler generated files and formats the code to be submitted.
10. README.md
    - This markdown file describes the path finding program, how to build it, how to run it, and also lists and explains all the command-line options that the program accepts. It also documents any false positives given by scan-build.
11. DESIGN.pdf
    - This pdf is the manual that explains the program, files included, layout or structure, and pseudo-code of the path finding algorithm.

3 Structure



4 Pseudo-code

All pseudo code is based off asgn4.pdf code

4.1 Graph

```
#initializes all fundamental variables necessary to make graph (its definition)
structure of Graph
    initialize vertices (number of vertices in the graph)
    initialize undirected (shows if it is or not)
    initialized visited vertices (shows if vertex has been visited or not)
    initialize matrix of vertices (makes the adjacency matrix of 26 by 26)

#Dynamically allocates memory and creates a graph array
define graph_create (constructor)
    dynamically allocate memory of graph array (0 out allocated memory)
    set all vertices in graph to not visited
    return graph

#Frees the memory of the graph and 0s the memory out to delete it
define graph_delete (deconstructor)
    free memory
    set graph to null
    return

#Returns the amount of vertices in the graph
define graph_vertices (accessor)
    return vertices in graph

#Adds an edge to the graph given the vertices coordinates and the weight of the edge if within b
define graph_add_edge (manipulator)
    if i and j within bounds
        add edge of i to j with weight k
        if graph is undirected
            add edge of j to i with weight k
        return True
    return False

#Shows whether there is an edge at the given vertex if within bounds
```

```
define graph_has_edge
    if i and j are within bounds
        if it is an edge
            return True
    return False

#Gives the weight of the edge if there exists an edge there if within bounds
define graph_edge_weight
    if i and j are within bounds
        if i and j are an edge
            return weight of the edge
    return 0;

#Shows whether the vertex provided has been visited or not
define graph_visited
    return visited vertex (true or false)

#If the vertex is within bounds then the vertex will be set to visited
define graph_marked_visited
    if vertex in bounds
        set visited to True

#If the vertex is within bounds then the vertex will be set to unvisited
define graph_marked_unvisited
    if vertex in bounds
        set visited to False

#Debug print function to check if graph is proper
define graph_print
    for i in vertices
        for j in vertices
            print matrix
```

4.2 Stack

```
#initializes all fundamental variables necessary to make a Stack (its definition)
structure of Stack
    initialize top (index of next empty slot)
    initialize capacity (shows amount of items that can be pushed)
```

```
    initialize array of items

#Creates the a dynamically allocated array of the Stack based on inputs
define stack_create (constructor)
    dynamically allocate Stack array using top, capacity, and items
    return s

#Frees the stack array and its items then nulls the memory out
define stack_delete (deconstructor)
    free stack

Shows whether the stack is empty based on if the top is zero
define stack_empty
    if stack top is empty
        return True
    return False

#Shows whether the stack is full based on if the top is capacity
define stack_full
    if stack capacity is full
        return True
    return False

#Gives the size of the stack based on where the index slot top is at
define stack_size
    return amount of items in stack

#Pushes or adds an item onto the stack if the stack is not full
define stack_push (manipulator)
    if items is not full
        push item
        increment top
        return True
    return False

#Pops or removes an item from the stack if the stack is not empty
define stack_pop
    if stack is not empty
```

```
        decrement top
        pop items (using asgn code)
        return True
    return False

#Gives the item at the top of the stack based on top
define stack_peek
    if stack is empty
        use pop code but do not decrement top
        return False

#Copies the items of the stack from its source to another destination
define stack_copy
    set destination stack to source stack (dst -> items = src -> items)

#Prints the cities of the path to an outfile for output
define stack_print
    print stack to outfile (using asgn code)
```

4.3 Path

```
#initializes all fundamental variables necessary to make a path (its definition)
structure of Path
    initialize stack of vertices (vertices compromising path)
    initialize length (total length of path)

#Dynamically allocates memory and creates the path of vertices using stack
define path_create (constructor)
    set vertices to stack
    set length to 0

#Frees the path stack array and its vertices then zeros its memory
define path_delete (deconstructor)
    free pointer p
    set pointer p to null

#Pushes or adds the vertex onto the path array and adds to path length based on vertex edge weig
define path_push_vertex
    push vertex v to path p
```

```
    if push
        increment length by weight of edge
        return True
    return False

#Pops or removes the vertex from the path array and decrements to path length based on vertex ed
define path_pop_vertex
    if pop
        set pointer v to popped vertex
        decrement length by weight of edge
        return True
    return False

#Gives the amount of vertices on the path
define path_vertices
    return amount of vertices in path

#Gives the length of the path
define path_length
    return path length

#Copies the vertices of the path from the source to a destination array
define path_copy
    set destination path to source path (like stack but with path)

#Prints out the path length and path of cities to an outfile
define path_print
    print stack to outfile (using asgn code from stack)
```

4.4 TSP

```
define options for command line
define dfs using pseudocode from asgn

define main function
    initialize infile and outfile
    make bool for switch case options
    use getopt
    use switch cases to parse through input (open infile and outfile if necessary + switch bools
```

```
if help is prompted
    print help msg

scan through first line to get number of vertices

make array to store cities
make an input char buffer to scan lines
scan through vertices number of lines to find cities in path

make graph
scan through rest of lines to find the vertices and its edge weights to add to graph

create path of shortest
create current path

print out shortest path, path length, and recursion calls to outfile (print all paths if ver

free memory
```

5 Error Handling

NOTE: recursion calls is different likely to a different variation of the algorithm

6 Credits

1. I used the asgn4.pdf from Professor Long for explanations and pseudocode.
2. I watched the Lab Section recording from Eugene held on 10/19.
3. I watched the Section recording from Christian held on 10/22.