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 create the Hamiltonian path algorithm. It will compute the shortest and most optimal route or path possible to get Denver back home.

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

5 Code

```
#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)

define graph_create (constructor)
    dynamically allocate memory of graph array (0 out allocated memory)
    set all vertices in graph to not visited
    return graph
```

```
define graph_delete (deconstructor)
    free memory
    set graph to null
    return

define graph_vertices (accessor)
    return vertices in graph

define graph_add_edge (manipulator)
    add edge of i to j with weight k
    if graph is undirected
        add edge of j to i with weight k
        if vertices are within bounds
            return True
        else
            return False

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

define graph_edge_weight
    if i and j are not within bounds or i and j are not an edge
        return 0
    else
        return weight of the edge

define graph_visited
    if vertex is visited
        return True
    else
        return False

define graph_marked_visited
    if vertex in bounds
        set visited to True
```

```
define graph_marked_unvisited
    if vertex in bounds
        set visited to False

define graph_print
    print graph (to debug and make sure graph runs properly)
```

5.1 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

define stack_create (constructor)
    dynamically allocate Stack array using top, capacity, and items
    return s

define stack_delete (deconstructor)
    free stack

define stack_empty
    if stack is empty
        return True
    else
        return False

define stack_full
    if stack is full
        return True
    else
        return False

define stack_size
    return amount of items in stack

define stack_push (manipulator)
```

```
    if items is not full
        push item
        return True
    else
        return False

define stack_pop
    if items is empty
        return False
    else
        pop items (using asgn code)
        return True

define stack_peek
    if stack is empty
        return False

define stack_copy
    set destination stack to source stack (dst -> items = src -> items)

define stack_print
    print stack to outfile (using asgn code)
```

5.2 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)

define path_create (constructor)
    set vertices to stack
    set length to 0

define path_delete (deconstructor)
    free pointer p
    set pointer p to null

define path_push_vertex
```

```
    push vertex v to path p
    increment length by weight of edge
    if push
        return True
    else
        return False

define path_pop_vertex
    set pointer v to popped vertex
    decrement length by weight of edge
    if pop
        return True
    else
        return False

define path_vertices
    return amount of vertices in path

define path_length
    return path length

define path_copy
    set destination path to source path (like stack but with path)

define path_print
    print stack to outfile (using asgn code from stack)
```

5.3 TSP

```
enumerate set of command line options

define main function
    use getopt
    use switch cases to parse through input
    call on necessary functions or variables for output
    use if statements to return or print output based on cases
```

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.