DESIGN.pdf ASGN4

Nathan Ong

October 24, 2021

1 Description

This collection of files contains a pathfinder that operates based off of Depth-first Search (DFS). The DFS searches a graph and finds the shortest possible path that visits all of the graph's vertices (Hamiltonian path). The main file used to initiate the pathfinder and its supporting files has the following command line options:

- -h: Prints out help message then exits the program.
- -v: Enables verbose printing; prints out all Hamiltonian paths that were found as well as the total number of recursive calls to dfs().
- -u: Specifies the graph to be undirected.
- -i infile: Specifies the input file path containing the graph, edges, and cities within it. The default input is stdin.
- -o outfile: Specifies the output file path to print the output to. The default output is stdout.

1.1 Sample Input Graph

```
4 /* number of vertices */
Asgard /* names of vertices */
Elysium
Olympus
Shangri-La
0 3 5 /* vertices and edge weights */
3 2 4
2 1 10
1 0 2
```

2 Files Included in the Directory

1. graph.h
   (a) This file is a header file that contains the function syntax for graph.c and the graph ADT.
2. graph.c
   (a) This file contains the source code for the functions that implement the graph ADT.
3. path.h
   (a) This file is a header file that contains the function syntax for graph.c and the path ADT.
4. path.c
   (a) This file contains the source code for the functions that implement the path ADT.

5. stack.h
   (a) This file is a header file that contains the function syntax for stack.c and the stack ADT.
6. stack.c
   (a) This file contains the source code for the functions that implement the stack ADT.
7. tsp.c
   (a) This file contains the source code for the main function that includes the command prompt parser and the DFS algorithm to find the shortest Hamiltonian path through the graph.
8. vertices.h
   (a) This file is a header file that contains the limit of the number of vertices present in a graph and an array that stores the vertices of a given graph.

3 Pseudocode and Structure

3.1 graph.c

1. graph vertices
```
return number of vertices
```
2. graph add edge
```
check if vertex is within bounds
apply weight to the vertex
if undirected, apply weight to opposite vertex too
return bool value for success
```
3. graph has edge
```
return true if vertices in bounds and edge isnt 0
return false if vertices out of bounds or if edge is 0
```
4. graph edge weight
```
return weight of edge between vertices if they are within bounds
return 0 if vertices not in bounds or no edge
```
5. graph visited
```
return true if vertex is visited (marked true in visited array)
return false if not
```
6. graph mark visited
```
if vertex in bounds, mark true in visited array
```
7. graph mark unvisited
```
if vertex in bounds, mark false in visited array
```
8. graph print
```
print adjacency matrix
```

3.2 stack.c

1. stack empty
   return true if top equals 0
   return false if not

2. stack full
   return true if top equals maximum capacity
   return false if not

3. stack size
   return top value (number of elements currently in stack)

4. stack push
   if stack isnt full
   append element to top of stack
   increment top by 1
   return true
   otherwise return false

5. stack pop
   if stack isnt empty
   remove element from top of stack
   decrement top by 1
   return true
   otherwise return false

6. stack peek
   if stack isnt empty
   return top element of stack
   if not return false

7. stack copy
   for all elements in stack
   newstack[n] = stack[n]

8. stack print
   print all items in stack

3.3 path.c

1. path create
   allocate memory for Path
   create stack for vertices
   length = 0
   return path

2. path delete
   if pointer s and pointer s points to items
   free pointer to items
   free pointer

null pointer

3. path push vertex
   get element on top of the stack
   if pushing vertex to stack is successful
   check if vertexes are the same
   increment length by edge weight of vertex
   return true
   return false if push is unsuccessful

4. path pop vertex
   if popping stack is successful
   get vertex on top of stack
   check if vertexes are the same
   decrement edge weight from length
   return true
   if pop unsuccessful return false

5. path vertices
   return the vertices in the path

6. path length
   return length of path

7. path copy
   call stack copy
   make length equal

8. path print
   call stack print
   print path length and path vertices stack

3.4 tsp.c

```
while opt isnt -1
    indice through arguments
    check for required arguments if necessary
    execute arguments and scan input file to get all information needed to run DFS
execute depth first search algorithm
    label v as visited
    for all edges connected to vertex
        if conncted vertex is labelled as visited
            call DFS algorithm for connected vertex
        if back at start vertex and all vertices visited
            add start vertex to path
            if shortest path is 0 or current path is shorter than the shortest path
                copy path and print path if verbose printing enabled
        pop vertex (end of else if)
    pop vertex (before vertex marked unvisited
    label vertex as unvisited
```

4 Additional Credits

1. Sloan's section on October 5th provided the Makefile's format.
2. The code for Stack and Graph create, along with delete, and the data structs were provided in the asgn4.pdf document.
3. The pseudocode for the depth-first search algorithm were provided in the asgn4.pdf document.
4. Eugene's in person section on October 21st provided the fscanf and fgets examples and format needed to complete the assignment.
5. The method for eliminating the new line character after fgets (strtok) was found online. `https://stackoverflow.com/a/2693826`

5 Error Handling

1. During testing of tsp.c, there was a segmentation fault (core dumped) and a double free error.
   (a) Solution: The for loop to free all the elements of the cities array used an incorrect variable that did not change. This variable was set to the correct one in the for loop, and the segmentation fault disappeared.
2. The dfs function would not work for some graphs.
   (a) Solution: Inequality for copying the path and the location of the first pop call caused an incorrect manipulation of the path. After changing the logic of the inequality and the location of the pop, the issue was resolved.
3. For the solar system sample graph, the path is longer than it should be. (path length 30 vs path length 28).
   (a) Solution: Path pop and push are pushing the same vertexes and adding the edge of that vertex as a result. Added a conditional in path push and path pop in order to prevent.