CSE 13S Fall 2021
James Gu
jjgu@ucsc.edu
21 October 2021

Assignment 4: Perambulations of Denver Long

Description of Program:

The goal of this program is to find the shortest path to traverse every location on a map only once when given an n x n matrix. This shortest path is called a Hamiltonian path. Each location is called a vertex and each edge between the vertices holds a different weight. These vertices and edges are used to represent the graph where a graph is a data structure $G = \langle V, E \rangle$ where $V = \{v_0,...,v_n\}$ and $E = \{\langle v_i, v_j \rangle,...\}$. The paths can be either directed or undirected. For example, an undirected path means that v_1 can go to v_2 and v_2 can go to v_1 and a directed path means that v_1 can go to v_2, but that doesn't necessarily mean v_2 can go to v_1.

Files:

vertices.h
In this header file, the VERTICES macro will be defined and there is another macro START_VERTEX which defines the origin vertex of the shortest Hamiltonian path we will be searching for.

graph.[ch]
graph.h specifies the interface to the graph ADT
graph.c implements the graph ADT

stack.[ch]
stack.h specifies the interface to the stack ADT
stack.c implements the stack ADT

path.[ch]
path.h specifies the interface to the path ADT
path.c implements the path ADT

tsp.c
Contains main() and may contain any other functions necessary to complete the assignment.

Makefile
- make
- make all
- make tsp
- make clean
- make format

Each of make, make all, and make tsp produce tsp, compiled with the required compiler flags.

README.md
- in Markdown
- briefly describes the program
- describes how to build program
- describes how to run program
- describes the command-line options your program accepts

DESIGN.pdf
- must be a pdf (do not just rename a text file)
- covers the purpose of the program
- layout/structure of the program
- clear description/explanation of how each part of the program should work
- supporting pseudocode (C is not considered pseudocode)

Pseudocode:

vertices.h
Set START_VERTEX to 0
Set VERTICES to 26

graph.c
struct Graph contains:
    vertices of type uint32_t
    undirected of type bool
    visited[VERTICES] of type bool
    matrix[VERTICES][VERTICES] of type uint32_t

Graph *graph_create(uint32_t vertices, bool undirected):
    Set up Graph *G with every element being 0
    Set G's vertices to the value of vertices
    Set G's undirected to value of directed
    Return G

void graph_delete(Graph **G):
    free *G
    Set *G to NULL

uint32_t graph_vertices(Graph *G):
    Return G's vertices

bool graph_add_edge(Graph *G, uint32_t i, uint32_t j, uint32_t k):
    Set G's matrix[i][j] to k
    If G's undirected is true:
        Set G's matrix[j][i] to k

Bool graph_has_edge(Graph *G, uint32_t i, uint32_t j):
    If the value in G's matrix[i][j] is greater than 0:
        Return true
    Return false

uint32_t graph_edge_weight(Graph *G, uint32_t i, uint32_t j):
    If i is greater than or equal to VERTICES or j is greater than or equal to VERTICES or graph_has_edge(*G, i, j) is false:
        Return 0
    Return the value in G's matrix[i][j]

Bool graph_visited(Graph *G, uint32_t v):
    If G's visited[v]:
        Return true
    Return false

Void graph_mark_visited(Graph *G, uint32_t v):
    If v is less than VERTICES:
        Set G's visited[v] to true

Void graph_mark_unvisted(Graph *G, uint32_t v):
    If v is less than VERTICES:
        Set G's visited[v] to false

Void graph_print(Graph *G):
    For i in every row:
        For j in every column:
            Print G's matrix[i][j]

stack.c
struct Stack contains:
    top of type uint32_t
    capacity of type uint32_t
    *items of type uint32_t

Stack *stack_create(uint32_t capacity):
    Set up a Stack called s
    If s doesn't need to be freed:
        Set s's top to 0
        Set s's capacity to the value of capacity
        Set s's items to an array of all 0's
        If s needs to be freed:
            Free s
            Set s to NULL

```
Return s

Void stack_delete(Stack **s):
        If *s and s's items need to be freed:
                Free s's items
                Free s
        Return

Bool stack_empty(Stack *s):
        If s's items array is empty:
                Return true
        Return false

Bool stack_full(Stack *s):
        If the s's items[capacity] is not 0;
                Return true
        Return false

uint32_t stack_size(Stack *s):
        Set size to 0
        Set i to 0
        For the elements of s's items array:
                If items[i] is not 0:
                        Add 1 to size
                Else:
                        Break
                Add one to i
        Return size

Bool stack_push(Stack *s, uint32_t x):
        If stack_full(s) is true:
                Return false
        Add x to s's items[s's top]
        Add one to set's top
        Return true

Bool stack_pop(Stack *s, uint32_t *x):
        If stack_empty(s) is true:
                Return false
        Set *x to s's items[s's top]
        Set s's items[s's top] to 0
        Subtract 1 from s's top
        Return true
```

```
Bool stack_peek(Stack *s, uint32_t *x):
        If stack_empty(s) is true:
                Return false
        Set *x to s's items[s's top]
        Return true

Void stack_copy(Stack *dst, Stack *src):
        Set dst's items to src's items
        Set dst's top to src's top
        Set dst's capacity to src's capacity

Void stack_print(Stack *s, FILE *outfile, char *cities[]):
        For s's top times:
                Print cities[s's items[i]
```

path.c

```
struct Path contains:
        *vertices of type Stack
        Length of type uint32_t

Path *path_create(void):
        Set p's vertices to stack_create(VERTICES)
        Set p's length to be 0
        Return p

Void path_delete(Path **p):
        Free *p
        *p = NULL
        Return

Bool path_push_vertex(Path *p, uint32_t v, Graph *G):
        Set temp to 0
        stack_peek(p's vertices, temp)
        Set v_length to graph_edge_weight(G, v, temp)
        Set push to stack_push(p's vertices, v)
        If push is false:
                Return false
        Else:
                graph_mark_visited(G, v)
                Add v_length to p's length
                Return true

Bool path_pop_vertex(Path *p, uint32_t v, Graph *G):
        Set pop to stack_pop(p's vertices, v)
```

```
        If pop is false:
                Return false
        Else:
                Set temp to 0
                stack_peek(p's vertices, temp)
                Set v_length to graph_edge_weight(G, v, temp)
                graph_mark_unvisited(G, v)
                Subtract v_length from p's length
                Return true

uint32_t path_vertices(Path *p):
        Return stack_size(p's vertices)

uint32_t path_length(Path *p):
        Return p's length

Void path_copy(Path *dst, Path *src):
        Set dst's vertices to src's vertices
        Set dst's length to src's length

Void path_print(Path *p, FILE *outfile, char *cities[]):
        stack_print(p's vertices, outfile, cites[])

tsp.c:
        Set h_flag, v_flag, u_flag, i_flag, and o_flag to false
        Initialize file
        Set input to stdin
        Set output to stdout
        While the user wants to test functions using the command line:
                If the case is 'v':
                        Set v_flag to true
                        break
                If the case is 'u':
                        Set u_flag to true
                        Set graph to undirected
                If the case is 'i':
                        If file was specified by user:
                                Set input to file
                        Set i_flag to true
                        Break
                If the case is 'o':
                        If file was specified by user:
                                Set output to file
                        Set o_flag to true
```

Break
If the case is ‘h’:
Set all other flags to false
Set h_flag to true
Print the help message
break
Read from input to generate the shortest path