CSE13s Fall 2021
Assignment 3 :- The Circumnavigations of Denver Long
By: Ruhin Gharai

The assignment:-

Is creating these files which implement a graph, stack, and path in order to use dfs to find the most optimal path based on a user defined input graph by optimal path its a hamiltonian path so it has to reach every single vertex once

Structure of the program:-

Graph.c:-

```
struct Graph {
        uint32_t vertices ; // Number of vertices .
        bool undirected ; // Undirected graph ?
        bool visited [ VERTICES ]; // Where have we gone ?
        uint32_t matrix [ VERTICES ][ VERTICES ]; // Adjacency matrix .
 };
```

```
#define START_VERTEX 0 // Starting ( origin ) vertex .
# define VERTICES 26 // Maximum vertices in graph .
```

Graph *graph_create(uint32_t vertices, bool undirected)

The constructor for a graph. A constructor function is responsible for initializing and allocating any memory required for the type it is constructing. It is through this constructor in which a graph can be specified to be undirected. Make sure each cell of the adjacency matrix, matrix, is set to zero. Also make sure that each index of the visited array is initialized as false to reflect that no vertex has been visited yet. The vertices field reflects the number of vertices in the graph. A working constructor for a graph is provided below. Note that it uses calloc() for dynamic memory allocation. This function is included in .

void graph_delete(Graph **G)

Any memory that is allocated using one of malloc(), realloc(), or calloc() must be freed using the free() function. The job of the destructor is to free all the memory allocated by the constructor.A pointer to a pointer is used as the parameter because we want to avoid use-after-free errors. A use-after-free error occurs when a program uses a pointer that points to freed memory. To avoid this, we pass the address of a pointer to the destructor function. By dereferencing this double pointer, we can make sure that the pointer that points to allocated memory is updated to be NULL.

uint32_t graph_vertices(Graph *G)

Since we will be using typedef to create opaque data types, we need functions to access fields of a data type. These functions are called accessor functions. An opaque data type means that users do not need to know its implementation outside of the implementation itself. This accessor function returns the number of vertices in the graph.

bool graph_add_edge(Graph *G, uint32_t i uint32_t j, uint32_t k)

The need for manipulator functions follows the rationale behind the need for accessor functions: there needs to be some way to alter fields of a data type.This function adds an edge of weight k from vertex i to vertex j. If the graph is undirected, add an edge, also with weight k from j to i. Return true if both vertices are within bounds and the edge(s) are successfully added and false otherwise.

bool graph_has_edge(Graph *G, uint32_t i uint32_t j)

Return true if vertices i and j are within bounds and there exists an edge from i to j. Remember: an edge exists if it has a non-zero, positive weight. Return false otherwise.

uint32_t graph_edge_weight(Graph *G, uint32_t i uint32_t j)

Return the weight of the edge from vertex i to vertex j. If either i or j aren’t within bounds, or if an edge doesn’t exist, return 0.

bool graph_visited(Graph *G, uint32_t v)

Return true if vertex v has been visited and false otherwise.

void graph_mark_visited(Graph *G, uint32_t v)

If vertex v is within bounds, mark v as visited.

void graph_mark_unvisited(Graph *G, uint32_t v)

If vertex v is within bounds, mark v as unvisited.

void graph_print(Graph *G)

A debug function you will want to write to make sure your graph ADT works as expected.

Stack.c

```
struct Stack {
 uint32_t top; // Index of the next empty slot
 uint32_t capacity ; // Number of items that can be pushed
 uint32_t * items ; // Array of items , each with type uint32_t
 };
```

Stack *stack_create(uint32_t capacity)

The constructor function for a Stack. The top of a stack should be initialized to 0. The capacity of a stack is set to the specified capacity. The specified capacity also indicates the number of items to allocate memory for, the items in which are held in the dynamically allocated array items. A working constructor for a stack is provided below. Note that it uses malloc(), as well as calloc(), for dynamic memory allocation. Both these functions are included from <stdlib.h>

void stack_delete(Stack **s)

The destructor function for a stack. A working destructor is provided below. Pay attention to the things that are freed and what is set to NULL.

bool stack_empty(Stack *s)

Returns true if the stack is empty and false otherwise.

bool stack_full(Stack *s)

Returns true if the stack is full and false otherwise.

uint32_t stack_size(Stack *s)

Returns the number of items in the stack.

bool stack_push(Stack *s, uint32_t x)

The need for manipulator functions follows the rationale behind the need for accessor functions: there needs to be some way to alter fields of a data type. stack_push() is a manipulator function that pushes an item x to the top of a stack.

bool stack_pop(Stack *s, uint32_t *x)

This function pops an item off the specified stack, passing the value of the popped item back through the pointer x. Like with stack_push(), this function returns a bool to indicate either success or failure.

bool stack_peek(Stack *s, uint32_t *x)

Peeking into a stack is synonymous with querying a stack about the element at the top of the stack. If the stack is empty prior to peeking into it, return false to indicate failure. This functions like stack_pop(), but doesn't modify the contents of the stack.

void stack_copy(Stack *dst, Stack *src)

Assuming that the destination stack dst is properly initialized, make dst a copy of the source stack src. This means making the contents of dst->items the same as src->items. The top of dst should also match the top of src.

void stack_print(Stack *s)

Prints out the contents of the stack to outfile using fprintf(). Working through each vertex in the stack starting from the bottom, print out the name of the city each vertex corresponds to. This function will be given to aid you.

Path.c:-

```
struct Graph {
        uint32_t vertices ; // Number of vertices .
        bool undirected ; // Undirected graph ?
        bool visited [ VERTICES ]; // Where have we gone ?
        uint32_t matrix [ VERTICES ][ VERTICES ]; // Adjacency matrix .
};
```

Path *path_create(void)

it holds the sum of the edge weights between consecutive vertices in the vertices stack.

void path_delete(Path **p)

The destructor for a path. Remember to set the pointer p to NULL.

bool path_push_vertex(Path *p, uint32_t v, Graph *G)

Pushes vertex v onto path p. The length of the path is increased by the edge weight connecting the vertex at the top of the stack and v. Return true if the vertex was successfully pushed and false otherwise.

bool path_pop_vertex(Path *p, uint32_t *v, Graph *G)

Pops the vertices stack, passing the popped vertex back through the pointer v. The length of the path is decreased by the edge weight connecting the vertex at the top of the stack and the popped vertex. Return true if the vertex was successfully popped and false otherwise.

uint32_t path_vertices(Path *p)

Returns the number of vertices in the path.

uint32_t path_length(Path *p)

Returns the length of the path.

void path_copy(Path *dst, Path *src)

Assuming that the destination path dst is properly initialized, makes dst a copy of the source p

void path_print(Path *p, FILE *outfile, char *cities[])

Prints out a path to outfile using fprintf(). Requires a call to stack_print(), in order to print out the contents of the vertices stack.

tsp.c

Your program must support any combination of the following command-line options using getopt function to make switch cases to call the file and its functions when needed to run.

```
procedure DFS (G,v) :
        label v as visited
        for all edges from v to w in G. adjacentEdges (v) do
        if vertex w is not labeled as visited then
                recursively call DFS (G,w)
        label v as unvisited
```

• -h : Prints out a help message describing the purpose of the graph and the command-line options it accepts, exiting the program afterwards. Refer to the reference program in the resources repo for an idea of what to print.

• -v : Enables verbose printing. If enabled, the program prints out all Hamiltonian paths found as well as the total number of recursive calls to dfs().

• -u : Specifies the graph to be undirected.

• -i infile : Specify the input file path containing the cities and edges of a graph. If not specified, the default input should be set as stdin

. • -o outfile : Specify the output file path to print to. If not specified, the default output should be set as stdout.

Design Process:-

So I first worked on graph.c then stack.c and path.c.My first error was small errors in graph_delete which one line of code is wrong so I deleted it then in my stack stack_full function i added -1 which i needed to delete when i am checking if the stack is empty . These two small errors caused my path to fail in the pipeline .Path.c was the hardest to code it was confusing how to implement the other files in that one and also wording in the pdf was really confusing and hard to translate to C which took me two days to do.