Merilyn Kuo
CSE 13S
Spring 2021

Assignment 2: A Small Numerical Library
Design Document

Purpose:

The purpose of this assignment is to learn about graph data structures. We learn how directed graphs are represented in matrices and will build an algorithm to find the most optimal route to a location. We learn how to find a Hamiltonian path and use Depth First Search to find paths that pass through all vertices of our graph.

Introduction:

This program will be solving the well known, traveling salesman problem. Denver Long is a traveling salesman that got lost in Mexico after taking a wrong turn. He tasked Professor Long's class with the job of creating a program to find the optimal route to take him back home. To do so, I will be implementing a graph ADT, path ADT, stack ADT and use Depth First Search to come up with the best solution.

Program Files:

Besides the header files mentioned in the Assignment PDF and provided to us in the repository, these will be the files containing the puzzle pieces to the entire program:

- graph.c : contains the implementation of the graph ADT
- stack.c : contains the implementation of the stack ADT
- path.c : contains the implementation of the path ADT
- tsp.c : contains the main function, command line option parsing, and the depth first search recursive algorithm to find the Hamiltonian path and solve the traveling salesman problem.

The provided header files are:

- vertices.h : defines macros about the vertices
- graph.h : declares the function headers for the graph ADT
- stack.h : declares the function headers for the stack ADT
- path.h : declares the function headers for the path ADT

Program Implementation:

Graph ADT:

The program will use an adjacency matrix to represent a graph. In C, matrices are represented as arrays of arrays where each row in the matrix is an array and all row arrays are in an array. I will need to create a graph ADT for this.

The following struct definition was provided in the Assignment 3 PDF. The VERTICES macro is defined in the vertices.h header file provided to us. The structure contains the number of vertices, a boolean value of whether the graph is undirected or not, an array of the visited vertices, and the adjacency matrix that represents the graph itself.

```
struct Graph {
    uint32_t vertices;
    bool undirected;
    bool visited[VERTICES];
    uint32_t matrix[VERTICES][VERTICES];
};
```

The function headers of the following graph functions were provided in the Assignment 3 PDF. The graph constructor needs to allocate memory for the graph and matrix and define the graph structure's variables. All the elements in visited[] must be false and the matrix elements initialized to zero. Here is the pseudocode:

```
Graph *graph_create(uint32_t vertices, bool undirected):
    Graph *g = (Graph *) malloc(sizeof(Stack))
    if graph memory allocated:
        g -> vertices = vertices
        g -> undirected = undirected
        for i in g -> visited[]:
            visited[i] = false
        g -> matrix = (uint32_t *) calloc(vertices, sizeof(uint32_t)
        if matrix memory not allocated:
            Free graph memory
            g = null
    return g
```

The graph destructor needs to free any allocated memory for the graph and matrix. The function takes in a double pointer parameter or the address of the pointer to prevent errors when the program uses a double pointer to freed memory.

```
void graph_delete(Graph **g):
    if (*g exists and (*g)->matrix exists):
       free graph matrix memory
       free graph memory
       *g = null
    return
```

This function returns the number of vertices in the graph.

```
uint32_t graph_vertices(Graph *g):
    return g->vertices
```

This function adds an edge of weight k if the edge's vertices are in bounds. If the graph is undirected it will add in its inverse spot.

```
bool graph_add_edge(Graph *g, uint32_t i, uint32_t j, uin32_t k):
    if i < g->vertices and j < g->vertices:
        g->matrix[i][j] = k
        if undirected:
            g->matrix[j][i] = k
        return true
    else return false
```

This function returns true if the edge exists at the specified vertices.

```
bool graph_has_edge(Graph *g, uint32_t i, uint32_t j):
    if i < g->vertices and j < g->vertices:
        if g->matrix[i][j] > 0:
            return true
    return false
```

This function returns the weight at the specified edge or 0 if the edge doesn't exist or is out of bounds.

```
uint32_t graph_edge_weight(Graph *g, uint32_t i, uint32_t j):
    if i < g->vertices and j < g->vertices:
        return g->matrix[i][j]
    return 0
```

This function returns true if the vertice has been visited and false elsewise.

```
bool graph_visited(Graph *g, uint32_t v):
    if g->visited[v]:
       return true
    return false
```

This function marks a vertice as visited (true) if it is within bounds.

```
void graph_mark_visited(Graph *g, uint32_t v):
    if v < VERTICES:
       g->visited[v] = true
    return
```

This function marks a vertice as not visited (false) if it is within bounds.

```
void graph_mark_unvisited(Graph *g, uint32_t v):
    if v < VERTICES:
       g->visited[v] = false
    return
```

This function prints the matrix in a matrix formation.

```
void graph_print(Graph *g):
    for i in g->matrix:
        for j in g->matrix:
            print(“ ” + g->matrix[i][j])
        print(new line)
    return
```

Stack ADT:

The path ADT will be using a stack. The structure definition for the stack abstract data path is given in the Assignment PDF as well as the function headers. Elsewise, the pseudocode is as follows:

```
struct Stack {
    uint32_t top;
    uint32_t capacity;
    uint32_t *items;
};

Stack *stack_create(uint32_t capacity):
    Stack *s = (Stack *) malloc(sizeof(Stack))
    if s memory allocated:
        s->top = 0
        s->capacity = capacity
        s->items = (uint32_t *) calloc(capacity, sizeof(uint32_t)
        if items not memory allocated:
            free s from memory
            s = null
    return s
```

```
void stack_delete(Stack **s):
    if (s exists and s->items exists):
        free s->items from memory
        free s from memories
        *s = null
    return

bool stack_empty(Stack *s):
    return top == 0

bool stack_full(Stack *s):
    return top == capacity

uint32_t stack_size(Stack *s):
    return top

bool stack_push(Stack *s, int64_t x):
    if top == capacity
        return false
    items[top] = x
    top += 1
    return true

bool stack_peek(Stack *s, uint32_t *x):
    x = s->items[s->top]

bool stack_pop(Stack *s, int64_t *x)
    if top == 0:
        return false
    top -= 1
    *x = items[top]
    return true

void stack_copy(Stack *dst, Stack *src):
    for item in src->items:
      stack_push(dst->items, item)
      dst->top = src->top
```

The following function for printing a stack has been given to us in the assignment PDF.

```
void stack_print(Stack *s, File *outfile, char *cities[]):
    for (uint32_t i = 0; i < s->top; i += 1) {
        fprintf(outfile, “%s”, cities[s->items[i]]);
        if (i + 1 != s->top) {
            fprintf(outfile, “ -> “);
        }
    }
    fprintf(outfile, “\n”);
}
```

Path ADT:

Vertices will be added and removed from the path of travel. The abstract data type of a path is as follows. The structure definition was given in the Assignment PDF. It contains a stack of vertices that make up the path and the total length of the path.

```
struct Path {
    Stack *vertices;
    uint32_t length;
};
```

The path ADT uses the graph and stack ADT. The following headers for the functions have been provided in the pdf. The following function is the path constructor. The vertices stack should be created to hold up to the number of vertices that exist and the initial length of the path is to be 0.

```
Path *path_create(void):
    Path *p = (Path *) malloc(sizeof(Path))
    if (p memory allocated):
        p->length = 0
        Stack p->vertices = stack_create(VERTICES)
        if p->vertices memory not allocated:
            free p->vertices of memory
            p = null
```

The path destructor function frees any memory that was allocated to the path.

```
Path path_delete(Path **p):
    if p exits and p->vertices exists:
        stack_delete(vertices)
        free p from memory
        *p = null
```

This function adds a vertex onto the path and increases the length by the edge weight of the vertex at the top of the stack and v.

```
bool path_push_vertex(Path *p, uint32_t *v, Graph *g):
    stack_push(p->vertices, v)
    stack_peek(p->vertices, &x)
    g->length += graph_edge_weight(g, x, v)
    return true
```

This function removes a vertex from the vertices stack and passes it through the pointer v. The path length is also decreased by the edge weight connecting the vertex at the top of the stack and the vertex that was popped.

```
bool path_pop_vertex(Path *p, uint32_t *v, Graph *g):
    popped = stack_pop(p->vertices, &v)
    peeked = stack_peek(p->vertices, &x)
    g->length -= graph_edge_weight(g, x, v)
    if popped and peeked:
        return true
    else return false
```

This function returns the number of vertices in the path.

```
uint32_t path_vertices(Path *p):
    return stack_size(vertices)
```

This function returns the length of the path.

```
uint32_t path_length(Path *p):
    return p->length
```

This function takes in the destination path that has been initialized and copies the source path into it. It copies all members of the path.

```
void path_copy(Path *dst, Path *src):
    stack_copy(dst, src)
    dst->length = src->length
```

This function prints the path to the outfile.

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

Specifics:

The main part of the program will be the tsp.c file and will contain the main function. Following the specifics in the Assignment PDF:

1. I need to parse the command-line options. The options as specified in the assignment PDF is:

`-h`: print out the help/usage message
`-v`: (verbose printing) enables the program to print out all Hamiltonian paths found and the total number of recursive class in dfs()
`-u`: specifies that the graph is undirected

`-i infile`: specifies the input file, otherwise the default input should be `stdin`
`-o outfile`: specifies the output file, otherwise the default output should be `stdout`

Like in previous assignments I can use a while loop with getopt() and a switch case for parsing. Since there are default input and outputs we will need to specify that before the while loop. For case v, I think I will set a boolean variable to true and handle the printing in the program later based on whether the user used this command line option. Similarly, for the case u, I will use a boolean variable that will later be passed into the graph_create function. With case i and o, if a file was specified, I can use fopen and read/write to them accordingly. The pseudocode for parsing command line options is as follows:

```
opt = 0
File *input_file = stdin
File *output_file = stdout
while ((opt = getopt(argc, argv, “hvui:o:”)) != 0):
    switch (opt):
        case ‘h’:
            print help/usage message
        case ‘v’:
            print_all = true
        case ‘u’:
            undirected = true
        case ‘i’:
            *input_file = fopen(argv, read)
            if the file specified is null:
                print stderr message
        case ‘o’:
            *output_file = fopen(argv, write)
            if file specified is null:
                print stderr message
        default:
            print help/usage message
```

2. The first line from the input is the number of cities (vertices) on the graph. I can use fscanf to scan in the first line and set it equal to a variable called num_cities. If the number is greater than the macro VERTICES, print an error message. This pseudocode is based on the code Eugene walked through in his section.

```
int num_cities
fscanf(input_file, “%d\n”, &num_cities)
if num_cities > VERTICES:
    print stderr message
```

3. For the number of cities, scan the next lines containing city names from the input with fgets and store them in an array. Since fgets takes in the maximum number of characters to be read I

will use a BUFFER macro of 1024 for 1024 max characters. This pseudocode is based on the code Eugene walked through in his section.

```
char cities[num_cities]
char city_name[BUFFER]
for i in range num_cities:
    fgets(city_name, BUFFER, output_file)
    cities[i] = strdup(city_name)
```

4. By now, I have the information I need to create an empty graph.

```
Graph G = graph_create(num_cities, undirected)
```

5. Now to add edges to the graph. I will scan the rest of the lines with fscanf and add the edge in. If the line is malformed, print an error message then exit the program. This pseudocode is based on the code Eugene walked through in his section.

```
int i, j, k
while (fscanf(intput_file, “%d %d %d\n”, &i, &j, &k) != EOF):
    if malformed:
        print error malformed
        break
    graph_add_edge(G, i, j, k)
```

6. Two paths are to be created. One for the current traveled path and one for the shortest path.

```
travel_path = path_create()
shortest_path = path_create()
```

7. The next step is to use depth first search to find the shortest Hamiltonian path. I can create a function for doing so with the function header provided and referencing the dfs pseudocode procedure the assignment PDF has. Then I can call the function in the main(). After the search, I can check if it is a Hamiltonian path. If it is, I check if it is a shorter path than the last. If it is, I copy the path to the shortest path and search for another Hamiltonian path. If verbose printing was enabled, print out all the Hamiltonian paths as they are found, else just print the shortest path at the end.
The pseudocode for the recursive DFS is as follows:

```
void dfs(Graph *G, uint32_t v, Path *curr, Path *shortest, char
*cities[], FILE *outfile):
    if graph_has_edge(G, stack_peek(curr, &x), START_VERTEX):
        if (path_length(curr) < path_length(shortest))
            path_copy(shortest, curr)
        if print_all == true:
            path_print(curr, outfile, cities[])
    if print_all == false
        path_print(shortest, outfile, cities[])
    if graph_length(curr) > graph_length(shortest):
        return;
    recursive_calls += 1
    graph_mark_visited(G, v)
    path_push_vertex(curr, v, G)
    for w in G->vertices:
        if graph_has_edge(G, v, w) and graph_visited(G, w) == false:
            dfs(G, w)
    graph_mark_unvisited(G, v)
    path_pop_vertex(curr, v, G)
```

Process and Changes to Design:

1. I made changes to my DFS algorithm. I had to add some more conditions to the if statements to account for edge cases. I also added in the beginning a check for a longer path. If the path is longer, I return and stop the recursive process on that path.

Credits and Resources Used:

Thank you to tutor Eric for helping me debug my DFS function and find edge cases to look out for. Thank you to Eugene for always hosting an extremely informative and helpful section.

Resources:
Introduction to Algorithms
man pages