Draft Report - Assignment 7

By Mira Saini

Purpose

This program is meant to sort and find the quickest path (smallest amount of weight) from an adjacency matrix of vertices and edges given a file that contains the number of vertices, the names of those vertices, the number of edges, and an adjacency list in the format ‘start,’ ‘end,’ and ‘weight.’ The program will implement a depth-first search (dfs) algorithm in order to create the shortest undirected (if not specified directed in the command line) Hamiltonian cycle path. If multiple Hamitonian paths are found that have the same length, the first one will be kept.

Testing

List what you will do to test your code. Make sure this is comprehensive. This question is a whole lot more vague than it has been the last few assignments. Continue to answer it with the same level of detail and thought. Be sure to test inputs with delays and a wide range of files/characters.

To test each of my functions in my stack, path, and graph implementation, I decided to create three separate C programs for each file. In my test_stack.c file, I decided to first use stack_create to create my stack s. I then created a variable of type uint32_t which is used as a return pointer value in my stack_peek function. I first checked that the stack_empty function was working prior to pushing anything to the stack. I assigned bool p to the return value of stack_empty then used an assert to ensure that the function returned true. Then I pushed 3 values onto the stack using stack_push and then used stack_empty again to ensure that this time it would return false. I also printed the return value of stack_size to check its functionality and then used stack_peek and printed that value. Then I decided to use stack_pop and then stack_peek again to print the last item on the stack (should be different since a value was just popped). I called stack_full to make sure it returned false (my capacity was very large) and then I changed the capacity when I created my stack and checked that stack_full reported back true. Then I checked the functionality of stack_copy by declaring a new pointer of type Stack and initializing it to NULL. Then I used stack_copy and then used stack_peek to make sure that the value of stack_peek(s) matched this stack_peek value. Then I used stack_free and then ensured that all memory was properly freed using valgrind ./test_stack.

To test my graph.c functions, I created a test_graph.c program. In this program, I used graph_create to create my graph called g, with parameters 3 (vertices) and false (meaning undirected). I then used graph_add_vertex 3 times and used graph_visit_vertex to add those vertices to a list of visited vertices. I then added an edge weight using graph_add_edge. Then I checked to make sure that graph_get_weight was returning the right weight by assigning it to a variable named weight and then printing it. I then used graph_get_vertex_name, assigning the return value to a pointer and then printing the value it pointed to to ensure that it was returning the right name. Then I used graph_print to check that the graph was correct so far. I tested graph_vertices by printing out its return value and then used an assert on the return value of graph_visited to make sure the value was expected. Then I used graph_unvisit_vertex to remove the vertex from the list of visited vertices and then called graph_visited again to ensure it was false this time. Then I assigned a double pointer to the return value of graph_get_names and then used a for loop to print the names of the vertices. Then using another loop, I freed the memory pointed to by the double pointer. I also freed the double pointer itself and the graph using graph_free. I tested to make sure the memory was properly freed using valgrind ./test_graph

To test my path.c functions, I created my test_path.c program. In my test_path.c program, I first used my path_create function to create a path called p with capacity 32. I then used an assert to ensure the path was successfully created. Then I also created an undirected graph called g using graph_create (I knew this function was working since I tested my graph functions before). I then used graph_add_vertex to add multiple vertices and then graph_visit_vertex to add those vertices to my list of visited vertices. Then I added edges between those vertices using graph_add_edge. Then I added those vertices from my graph to my path using path_add. I then assigned my distance variable of type uint32_t to the return value of path_distance and then printed that value out to ensure it was the expected value. I did the same thing for path_vertices and checked that the number of vertices in my path was as expected. Then I printed out the whole path to ensure that everything was added correctly. I then used path_remove, assigning the returned value to my removed variable and then printing the value of removed out. Then I used path_free and graph_free to make sure all memory was properly freed. I tested the memory leakage using valgrind ./test_path.

Questions

What benefits do adjacency lists have? What about adjacency matrices?
Adjacency list benefits:
1. Space/memory efficient compared to matrix
2. Easy to handle changes, adding or removing a vertex/edge
3. Easy to traverse

Matrix Benefits:
1. Checking if edge exists is easy
2. Better for bigger graphs with lots of edges

Which one will you use? Why did we chose that (hint: you use both)
I’ll be using both a list and matrix. For edge checks between my vertices, having a matrix will be a lot easier and more efficient when implementing my dfs algorithm. For iterating through

neighboring vertices to explore potential paths in my dfs implementation, it will be more optimal to use a list.

If we have found a valid path, do we have to keep looking? Why or why not?
Yes, we have to keep looking because a valid path doesn't mean it is the most optimal path. In order to find the shortest, most efficient path, I need to implement my dfs algorithm to find the shortest path that visits all vertices and returns to the first vertex it started with.

If we find 2 paths with the same weights, which one do we choose?
We use the first path that was found first.

Is the path that is chosen deterministic? Why or why not?
Yes, the path is deterministic meaning that it is the same every time for the exact same command line input and file/stdin input. The vertices and edges don't change so the shortest path remains the same each time.

What type of graph does this assignment use? Describe it as best as you can
The assignment uses an undirected graph as the default. An undirected graph means that an edge from point A to point B is the same for point B to point A. The assignment also uses directed graphs (if prompted through the command line), which is basically the opposite, meaning that an edge between point A and point B will not be the same automatically (unless specified) for point B to point A.

What constraints do the edge weights have (think about this one in context of Alissa)? How could we optimize our dfs further using some of the constraints we have?
Edge weights can't be negative and edges might be symmetrical or asymmetrical (depending if it's an undirected or a directed graph). Using these constraints, its apparent that once the current path exceeds that of the shortest path, then it cannot subtract weight and therefore can be thrown out, which will save us time and optimize the dfs algorithm, Additionally, I can also sort the neighbors of the current vertex by edge weight to get to the shortest path faster.

How to Use the Program

Audience: Write this section for the user of your program. You are answering the basic question, ``How do I use this thing?". Don't copy the assignment exactly; explain this in your own words.

To use this program, enter the command ./tsp [any command options: -d -i -h -o] [name of graph, if -i included].

The command line options include ‘-d’ which makes the graph a directed graph (default is undirected), ‘-i’ which receives input from a specified file, ‘-h’ which prints the help message, and ‘-o’ which writes the output to a specific file. Using ./tsp -d -i basic.graph will run tsp and create a directed graph using the input file basic.graph.

Program Design

Audience: Write this section for someone who will maintain your program. In industry you maintain your own programs, and so your audience could be future you! List the main data structures and the main algorithms. You are answering the basic question, ``How is this thing organized so that I can have a chance of fixing it?". This section will be longer for a more complicated program and shorter for a less complicated program.

My program has a few separate files that manage the stack, path, graph, and then the main tsp.c file which includes the sorting dfs algorithm. In my stack.c file, I have all of my stack functions and I’ve also defined my Stack struct here. In my stack.h file, I declare all of my functions and Stack struct. My path.c file contains all of my path functions and my Path struct, while my path.h file contains all of my declarations for my functions and the struct. My graph.c file contains all of my graph function definitions, while my graph.h file has all of my function declarations and my Graph struct definition + declaration.
In my tsp.c, I define fourmore functions and I also include a copy of my Path struct definition at the top of the file. My three other functions are print_help (prints help message), graph_read_from_file (which reads in the input from the provided file and adds them to create a graph), dfs which is my sorting algorithm that actually finds the most optimal path, and path_print_reverse which prints my path in reverse order (since my path is in the opposite order). Inside my main function, I use getopt to get the input options from the command line and then I open a file, initializing it to stdin. If the input_filename is not NULL (what is was initially set to), meaning that the command option ‘-i’ was provided, then input_filename is reassigned to optarg, then the file is opened to be read. Then a graph is created using the function graph_read_from_file, which will also read in the input from the file and create the full graph. An adjacency list is created called visited and two paths are created: current_path and shortest_path. I then call my dfs function here and once it returns and if a path is found, then the path is printed. Everything is then freed from memory and the input_filename (if it wasn’t NULL) is closed and if the outfile wasn’t set to stdout (command option ‘-o’ was provided), then the outfile is closed too.

Pseudocode

Give the reader a top down description of your code! How will you break it down? What features will your code have? How will you implement each function?

tsp.c file implementation:

```
void print_help(){
   fprintf(stdout, "Usage: tsp [options]")
   fprintf(stdout, options and their descriptions)
}

Graph *graph_read_from_file(FILE *infile, bool directed){
        //error checks
        if(infile == NULL){print error message}

        uint32_t num_vertices;
        //fscanf
        if (fscanf(infile, “%u\n”, &num_vertices != 1){
                //print error message
                fclose(infile)
                exit(1)
        }

        Graph *g = graph_create(num_vertices, directed)
        if(g == NULL){
                print error
                fclose(infile)
                exit(1)
                }

        //declare adjacency list
        for(uint32_t i = 0; i < num_vertices; i++){
                if (fgets(name, max, infile) == NULL){
                        Print error
                        Free graph
                        Close infile
                }
                //replace newline with terminating 0
                graph_add_vertex(g, name, i)
        }
```

```
	//initialize num_edges var
	if(fscanf(infile, "%u\n", &num_edges) != 1){
		Print error
		Free graph
		Close file
	}

	//for loop from 0 to num_edges
	//initialize from, to, weight
	if fscanf(infile, "%u %u %u\n", &from, &to, &weight) != 3){
		Print error
		Free graph
		Close file
	}
	graph_add_edge(g, from, to, weight)
	//if undirected, then do graph_add_edge(g, to, from, weight)


dfs(graph, current_path, shortest_path, visited, current_vertex, start_vertex, found_cycle){
	//mark current_vertex as visited
	path_add(current_path, current_vertex, g)

	if num_vertices of current_path == graph_vertices(g) &&  graph_get_weight > 0{
		path_add (current_path, start_vertex, g)
		//set found_cycle to true
		if path_distance(current) < path_distance(shortest) or path_distance(shortest)
	==0{
			path_copy(shortest, current)
		}
		path_remove(current_path,g)

	}else{
		//create array of vertices to sort by weight
		//sort those vertices by weight to make dfs efficient (find shortest path first)
		For loop from 0 to graph_vertices(g){
			Uint32_t next_vertex = vertex[i]
			if vertex is not visited and graph_get_weight(g, current, next)>0){
				//recursive call to dfs
			}
		}
```

```
        }
        path_remove(current_path, g)
        //set visited of current_vertex to false


path_print_reverse(path, outfile, graph){
        //check that p, outfile, and g aren't NULL
        //create reversed_stack using stack_create of size p->vertices
        //create temp_stack using stack_create of size p->vertices

        //initialize variable val

        while stack of p->vertices is not empty{
                stack_pop(p->vertices, &val)
                stack_push(temp_stack, val) //push popped val to temp
                stack_push(reversed_stack, val)
        }

        while temp_stack is not empty{
                stack_pop(temp_stack, &val)
                stack_push(p->vertices, val) //restore og stack
        }

        While reversed stack is not empty{
                stack_pop(reversed_stack, &val)
                print popped value
        }

        //free temp_stack and reversed_stack



void main(argc, *argv){
        //initialize directed to false
        //initialize input_filename and output_filename to NULL
        //initialize outline to stdout
        //initialize opt

        //implement getopt for the various command options here
```

```
//initialize infile to stdin
if(input_filename != NULL){
	Infile = fopen(input_filename, “r”)
	If infile is NULL{
		Print error
	}
}

//do same thing for output_filename

Graph *g = graph_read_from_file(infile, directed)

Bool *visited = calloc //set aside memory
//initialize current and shortest_path using path_create
//initialize bool found_cycle to false
dfs(g, current_path, shrtest_path, visited, 0,0,&found_cycle)

If found_cycle == true{
	Print “Alissa starts at:”
	path_print_reverse(shotest_path, outfile, g)
	Print “Total distance:%u, path_distance(shortests_path)”
}else{
	Print “no path found! Alissa is lost!”
}
//free graph, visited, current_path, shortest_path
If input_filename != NULL {fclose(infile)}

If outfile!= stdout{fclose(outfile)}
```

path.c implementation:

```
	//define and initialize path struct

Path *path_create(uint32_t capacity){
	Path *p = Path * malloc(sizeof(Path))
	If p == NULL, return NULL

	//set p->total_weight = 0 and p->vertices = stack_create(capacity)
```

```
        If p->vertices == NULL, free p and return NULL
        }
        return p
}

void path_free(Path **pp){
        If pp or *pp == NULL, return
        Path *p = *pp
        free *p->vertices
        free p
        *pp = NULL
}

uint32_t path_vertices(const Path *p){
        If p == NULL, return 0
        otherwise return stack_size(p->vertices)
}

uint32_t path_distance(const Path *p){
        If p == NULL, return 0
        otherwise return p->total_weight
}

Void path_add(Path *p, uint32_t val, const Graph *g){
        If p or g == NULL, return
        //initialize prev_vertex
        If stack_size of p->vertices > 0{
                If stack_peek of p->vertices {
                        uint32_t weight = g->weights[prev_vertex][val]
                        If weight > 0, p->total_weight += weight
                }
        }
}
stack_push(p->vertices, val)
}

uint32_t path_remove(Path *p, const Graph *g){
        If p or g == NULL or stack_size of p->vertices == 0{print error}
        //initialize removed_vertex
        if stack_pop of p->vertices returns false{print failed to pop vertex}
```

```
        If stack_size(p->vertice) > 0{
                //initialize prev_vertex
                If stack_peek(p->vertices, *prev_vertex)){
                        //assign weight t g->weights[prev_vertex][removed_vertex]
                }
                if(weight>0){subtract weight from p's total weight}
                return removed_vertex
        }


void path_copy(Path *dst, const Path *src){
        If dst or src == NULL{return}
        stack_free(dst->vertices)

        //initialize size to stack_size of src->vertices
        dst->vertices = stack_create(size) //same size as src

        //create and initialize temp_stack
        while stemp_stack is not empty{
                //initialize val
                if stack_pop of temp_stack fails, then free temp_stack and return
                stack_push(src->vertices, val)
                stack_push(dst->vertices, val)
        }
        stack_free(&temp_stack)
        //Set dst total weight to src's total weight
}

Void path_print(const Path *p, FILE *outfile, const Graph *g){
        if p, outfile, or g == NULL, then return
        //create a temp_stack
        //initialize top var

        While the p->vertice stack is not empty{
                //stack_pop(p->vertice, top)
                Print g->names[top]
                stack_push(temp_stack, top)
        }

        While the temp_stack is not empty{
```

```
        stack_pop(temp_stack, top)
        stack_push(p->vertices, top)
    }
    Free stack
}
```

graph.c implementation:

```
Graph *graph_create(uint32_t vertices, bool directed){
    Graph *g = calloc(1,sizeof(Graph))
    g->vertices = vertices
    g->directed = directed

    //allocate memory for g->visited and g->names and g->weights

    For loop from 0 to vertices{
        //allocate memory for g->weights[i]
    }
    return g
}

void graph_free(Graph **gp){
    if !gp and !*gp, then return
    Graph *g = *gp
    For loop from 0 to g->vertices{
        Free g->weights[i]
        Free g->names[i]
    }
    Free g->names
    Free g->weights
    Free g->visited
    Free g
    *gp = NULL
}

Uint32_t graph_vertices(const Graph *g){
    If !g, return 0
```

```
        Otherwise return g->vertices
}

Void graph_add_vertex(Graph *g, const char *name, uint32_t v){
        If g->names[v], then free it
        Otherwise g->names[v] = strdup(name)
}

Const char *graph_get_vertex_name(const Graph *g, uint32_t){
        If !g or v is more than g->vertices, return NULL
        Otherwise return g->names[v]
}

char **graph_get_names(const Graph *g){
        If !g, return NULL
        //allocate memory for names_array
        If !names_array, return NULL

        For loop from 0 to g->vertices{
                Names_array[i] = strdup(g->names[i])
                If !names_array[i]{free it using for loop}
                Free names_array and return NULL
        }

        return names_array
}

graph_add_edge(Graph *g, uint32)t start, uint32_t end, uint32_t weight){
        If !g or start or end is more than g->vertices, then return
        If !g->weights{
                //allocate memory for g->weights
                For loop from 0 to g->vertices{
                        //allocate memory for g->weights[i]
                }
        }
        g->weights[start][end] = weight
        If !g->directed, then g->weights[end][start] = weight
}
```

```
Uint32_t graph_get_weight(const Graph *g, uint32_t start, uint32_t end){
	If start or end is greater than g->vertices, return 0
	//initialize weight to g->weights[start][end]
	If g->directed &&weight == 0, return 0
	If !g->directed && weight == 0, weight = g->weights[end][start];
	return weight;
}

Void graph_visit_vertex(Graph *g, uint32_t v){
	If v is more than g->vertices, return
	Otherwise g->visited[v] set to true
}

Void graph_unvisit_vertex(Graph *g, uint32_t v){
	If v is more than g->vertices, return
	Otherwise set g_.visited[v] to false
}

Bool graph_visited(const Graph *g, uint32_t v){
	If v is more than g->vertices, then return false
	Otherwise return g->visited[v]
}

void graph_print(const Graph *g){
	Print g->vertices //num of vertices
	Print directed //true or false
	Print vertex names using for loop from 0 to g->vertices
	Print visited status using for loop from 0 to g->vertices
	Print edge weights using two for loops (matrix)
}
```

stack.c implementation:

```
//include Stack struct

Stack *stack_create(uint32_t capacity){
	Stack *s //use malloc to allocate memory
	s->capacity = capacity
	s->top = 0
	s->items //allocate memory for items using calloc
	return s
}

bool stack_full(const Stack *s){
	If s == NULL, return false
	Otherwise, return true if  s->top == s->capacity
}

Void stack_free(Stack **sp){
	If sp or *sp != NULL{
		If *sp ->items{
			Free *sp -> items and set *sp->items to NULL
		}
		free(*sp)
	}
	If sp != NULL, set *sp to NULL
}

Bool stack_push(stack *s, uint32_t val){
	If s stack is full, return false

	s->items[s->top] = val
	//Increment s->top
	return true
}

Bool stack_pop(Stack *s, uint32_t *val){
	If s == NULL or s->top = 0, return false
}

	*val = s->items[s->top-1]
```

```
	Decrement s->top
	return true


bool stack_peek(const Stack *s, uint32_t *val){
	If s is NULL or s->top = 0{return false}
	*val = s->items[s->top-10
	return true
}

Bool stack_empty(const Stack *s){
	If s == NULL, return false
	return true if  s->top == 0
}

Uint32_t stack_size(const Stack *s){
	If s == NULL, return 0
	return s->top
}

void stack_copy(Stack *dst, const Stack *src){
	If dst or src == NULL, return
	Assert that dst->capacity is more than or equal to src capacity
	For loop from 0 to number of element in src{
		Set dst item[i] to src item[i]
	}
	Set dst number of elements to src number of elements
	return;
}

void stack_print(const Stack *s, FILE *outfile, char *cities[]){
	For loop from 0 to the number of items in stack{
		Print cities[s->items[i]])
	}
	return
}
```

Function Descriptions

The inputs of every function (even if it's not a parameter)
The outputs of every function (even if it's not the return value)
The purpose of each function, a brief description about a sentence long.

Note* The pseudocode for each of these functions is in the previous section.

Functions in tsp.c:
void print_help(void): This function simply prints the help message when an error occurs or when the command option '-h' is given.

Graph *graph_read_from_file(FILE *infile, bool directed):
This function reads from the file pointed to by infile and reads in each line and creates a graph from those inputs. The function uses graph_add_edge and graph_create functions to initialize and define the graph. This function takes in a file and a true/false var to determine if the graph is directed or undirected as parameters. It then returns a pointer to the created graph.

void dfs(Graph *g, Path *current_path, Path *shortest_path, bool *visited, uint32_t current_vertex):
This function is my depth first search sorting algorithm. As parameters, it takes in a graph, two stored paths, a boolean value to determine if a vertex has been visited, and then current vertex. This function first sorts the vertices by weight to find the shortest path as quickly as possible. If the current_path is shorter than the shortest_path it reassigns current_path to shortest_path. The dfs algorithm uses an adjacency list to move to each vertex's neighboring vertex. Nothing is returned from this function, but the pointer to the shortest_path is updated and used in the main function of tsp.

void path_print_reverse(const Path *p, FILE *outfile, const Graph *g):
This function just reverses my path and prints it. It takes in the path, the outfile for where the path will be printed to and a graph. It does not return anything.

Functions in path.c:
Path *path_create(uint32_t capacity):
The input of this function is the capacity of the path (how much can it hold). The output is a pointer to the path of type Path. The purpose of this function is to create a functioning, empty path with a specific capacity.

void path_free(Path **p):

The input of this function is a double pointer to a path. The function doesn't return anything, but it frees the path and all of its memory.

uint32_t path_vertices(const Path *p):
The input of this function is a pointer to a path. This function finds the number of vertices in a path and then returns that number.

uint32_t path_distance(const Path *p):
This function finds the total distance covered by a path and returns the value. It takes a path in as a parameter.

void path_add(Path *p, uint32_t val, const Graph *g):
This function adds a vertex value from graph g to the path. It also updates the distance and length of the path. Adding a vertex to an empty path means the distance remains 0. This function takes in a path, a value to add to the path, and a graph

uint32_t path_remove(Path *p, const Graph *g):
This function removes the most recently added vertex from the path. It updates the distance and length of the path based on the adjacency matrix in the graph pointed to by g. If the last vertex is removed from the path, the distance should be 0. The distance can only be non-zero when there are at least two vertices in the path. It returns the index of the removed vertex.

void path_copy(Path *dst, const Path *src):
This function copies a path from src (source) to dst (destination). It doesn't return anything. It takes in two pointers of type path - one pointing to the source path and the other pointing to the destination path.
void path_print(const Path *p, FILE *outfile, const Graph *g):
This function prints the stored path using vertex names from the graph, printing to the outfile. It should only print the names of the vertices.

Stack.c implementation

Stack *stack_create(uint32_t capacity):
This function creates a stack. It returns a pointer to a stack. The capacity is the number of elements it takes for the stack to be full.

bool stack_full(const Stack *s):
This function returns true if the stack is full, otherwise false. It checks if the stack is at capacity.

void stack_free(Stack **sp):
Frees all associated memory in the stack using parameter double pointer to the stack.

bool stack_push(Stack *s, uint32_t val):
This pushes an element onto the stack. It takes in the stack and the value to be pushed as a parameter. It returns true if successful, otherwise false.

bool stack_pop(Stack *s, uint_32 val):
This pops the last value on the stack and sets the integer pointed to by val to popped item. It returns true if successful, otherwise false.

bool stack_peek(const Stack *s, uint32_t val):
This sets the integer pointed to by val to the last item on the stack, but it does not modify the stack. It returns true if successful, otherwise false.

bool stack_empty(const Stack *s):
The function returns true if the stack is empty, otherwise false.

uint32_t stack_size(const Stack *s):
This returns the number of elements in the stack.

void stack_copy(Stack *dst, const Stack *src):
This overwrites dst with all of the items from src. It updates dst->top to know how many items are now in the stack.

void stack_print(const Stack *s, FILE *outfile, char*cities[]):
This function simply prints the stack contents to the outfile. It takes in the stack, outline, and a pointer to an array of names.

graph.c implementations:

Graph *graph_create(uint32_t vertices, bool directed):
This function creates a graph and allocates memory for it. It takes in the number of vertices and a bool value to determine if the graph is directed (true) or undirected (false).

void graph_free(Graph **gp):
This function frees all memory used by the graph and sets the graph pointer to NULL. The parameter of this graph is a double pointer to the graph.

uint32_t graph_vertices(const Graph *g):
This function finds the number of vertices in a graph and returns it.

void graph_add_vertex(Graph *g, const char *name, uint32_t v):
This function adds a vertex to the graph g and to the a name array.

const char *graph_get_vertex_name(const Graph *g, uint32_t v):
This function gets the names of the vertex. It does not allocate a new string, it just returns the one stored in the graph.

char **graph_get_names(const Graph *g):
This function gets names of every vertex in an array and returns a double pointer to it.

void graph_add_edge(Graph *g, uint32_t start, uint32_t end, uint32_t weight):
This function adds an edge between start and end with a weight to the adjacency matrix of the graph.

uint32_t graph_get_weight(const Graph *g, uint32_t start, uint32_t end):
This function looks up the weight of the edge between start and end and then returns it.

void graph_visit_vertex(Graph *g, uint32_t v):
This graph adds the vertex v to the list of vested vertices.

void graph_unvisit_vertex(Graph *g, uint32_t v):
This graph removes the vertex v from the list of visited vertices.

void graph_print(const Graph *g):
This function prints some data about the graph (for debugging purposes). It prints the names of the vertice,the visited status of each vertex, and the weights of each edge.