Assignment 6 – Sufin' USA

Jayden Sangha

CSE 13S – Winter 24

Purpose

Audience for this section: Pretend that you are working in industry, and write this paragraph for your boss. You are answering the basic question, "What does this thing do?". This section can be short. A single paragraph is okay.

Do not just copy the assignment PDF to complete this section, use your own words.

The purpose of this program is to make a map that shows various cities and beaches and the distances between them.

Questions

Please answer the following questions before you start coding. They will help guide you through the assignment. To make the grader's life easier, please do not remove the questions, and simply put your answers below the text of each question.

- What benefits do adjacency lists have? What about adjacency matrices?
  Adjacency lists do not take up much memory and it is easier to iterate over the vertices. Adjacency matrices are able to add and change edges without having to iterate through the every entry.
- Which one will you use. Why did we chose that (hint: you use both)
  I am going to use both, because I need to be able to edit and add edges instantaneously and I need to be able to easily iterate through the names.
- If we have found a valid path, do we have to keep looking? Why or why not?
  We only have use for one path so once we find a valid path we no longer need to keep searching.
- If we find 2 paths with the same weights, which one do we choose?
  When we have 2 paths with the same weight we should choose the path that we discover first.
- Is the path that is chosen deterministic? Why or why not?
  The chosen path is deterministic, because it we need it to make the same graph every time with the same conditions.
- What type of graph does this assignment use? Describe it as best as you can
  This assignment uses directed and undirected graphs, because the user can travel both ways on a path or only go one way depending on their trip.
- 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 represent the distance, time, energy, cost, and difficulty to get from city to city. We could optimize this by getting rid of less efficient paths.

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. This section will be longer for a more complicated program and shorter for a less complicated program. You should show how to compile and run your program. You should also describe any optional flags or inputs that your program uses, and what they do.

To use this code you must first compile it by using

```
make
```

you then use

```
./tsp
```

followed by

```
-i (.graph file name)
```

to set the file to read from the given file or stdin by default.

```
-o (output file name)
```

to set the file to write to the output file.

To show "code font" text within a paragraph, you can use `\lstinline{}`, which will look like this: `text`.

For a code block, use `\begin{lstlisting}` and `\end{lstlisting}`, which will look like this:

```
Here is some code in lstlisting.
```

And if you want a box around the code text, then use `\begin{lstlisting}[frame=single]` and `\end{lstlisting}`

which will look like this:

```
Here is some framed code (lstlisting) text.
```

Want to make a footnote? Here's how.[1]

Do you need to cite a reference? You do that by putting the reference in the file `bibtex.bib`, and then you cite your reference like this[**?**][**?**][**?**].

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.

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.

Here is my Pseudocode for Graph.c

```
Graph *graph\_create(uint32\_t vertices, bool directed)

Create a new graph object
Set the number of vertices and whether it's directed
```

[1] This is my footnote.

```
Initialize arrays for visited vertices, vertex names, and the adjacency matrix
Return the created graph


void graph_free(Graph **gp)

Dereference the double pointer to get the graph pointer
Free memory for vertex names, weights, visited array, and the graph itself
Set the graph pointer to NULL to prevent accessing freed memory


uint32_t graph_vertices(const Graph *g)
Return the number of vertices stored in the graph structure

void graph_add_vertex(Graph *g, const char *name, uint32 t v)
If there was already a name assigned to the vertex, free the old name
Allocate memory for a copy of the new name and store it in the graph


const char* graph_get_vertex_name(const Graph *g, uint32_t v)
Retrieve the name of the vertex stored in the graph structure.


char **graph_get_names(const Graph *g)
Set the weight of the edge from start to end
If the graph is undirected, set the weight of the edge from end to start as well


void graph_add_edge(Graph *g, uint32_t start, uint32_t end, uint32_t weight)
Set the weight of the edge from start to end
If the graph is undirected, set the weight of the edge from end to start as well


uint32_t graph_get_weight(const Graph *g, uint32_t start, uint32_t end)
Retrieve the weight of the edge between start and end stored in the graph

void graph_visit_vertex(Graph *g, uint32_t v)
Set the visit status of the vertex to true in the visited array

void graph_unvisit_vertex(Graph *g, uint32_t v)
Set the visit status of the vertex to false in the visited array


bool graph_visited(const Graph *g, uint32_t v)
Return true if the vertex is marked as visited, false otherwise

char **graph\_get\_names(const Graph *g)
Return all of the city names in the graph

void graph\_add\_vertex(Graph *g, const char *name, uint32\_t v)
if name not already in the list add to the list of names.

const char *graph\_get\_vertex\_name(const Graph *g, uint32\_t v
Return the name of the given vertex.
```

Here is my Psudocode for stack.c

```
function stack\_create(capacity)
s = allocate memory for a Stack
s.capacity = capacity
```

```
set top to 0
s.items = allocate memory for an array of size capacity, initialized with zeros return s

function stack\_free(sp)
if the array of items is not null
deallocate memory for the array of items
deallocate memory for the stack

bool stack\_push(Stack *s, uint32\_t val)
if stack\_full return false
set items[top] to val
increment top
return true

bool stack\_pop(Stack *s, uint32\_t *val)
decrement s.top
set val to s.items[s.top]
return true

bool stack\_peek(const Stack *s, uint32\_t *val)
if not stack\_empty
set val to s.items[s.top - 1]
return true

bool stack\_empty(const Stack *s)
if s.top equals 0
return true

bool stack\_full(const Stack *s)
if top equals capacity return true


uint32\_t stack\_size(const Stack *s)
return top

void stack\_copy(Stack *dst, const Stack *src)
for all items in list
dst = src

void stack\_print(const Stack *s, FILE *f, char *vals[])
for i from 0 to s.top - 1
print vals[s.items[i]] to file f
```

Heres my Psudocode for path.c

```
Path *path\_create(uint32\_t capacity)
p = allocate memory for a Path
p.total\_weight = 0
p.vertices = stack\_create(capacity)
return p


void path\_free(Path **pp)
if the stack in the path is not null
free memory for the array of items in the stack
free memory for the path


uint32\_t path\_vertices(const Path *p)
Return the amount of verticies in the path
```

```
uint32\_t path\_distance(const Path *p)
Return the weight of the path


void path\_add(Path *p, uint32\_t val, const Graph *g)
if stack\_full(p.vertices)
return
if path\_vertices(p) > 0
stack\_peek(p.vertices, i)
p.total\_weight += get weight of new path
stack\_push(p.vertices, val)


uint32\_t path\_remove(Path *p, const Graph *g)
pop first item
update weight of the path


void path\_copy(Path *dst, const Path *src)
dst weight = src weight
dst vertices = src vertices


void path\_print(const Path *p, FILE *f, const Graph *g)
print all city names in the path
```

Here is my Psudocode for tsp.c

```
function dfs(g, ver, p, smalp):
    Visit the vertex 'ver' in the graph 'g'.
    Add the vertex 'ver' to the path 'p'.

    Iterate over all vertices in the graph 'g':
        If there is an edge from 'ver' to vertex 'i' and 'i' has not been visited:
            Recursively call dfs(g, i, p, smalp) to explore the adjacent vertex 'i'.

        If the path 'p' contains all vertices in the graph:
            If there is a connection from 'ver' back to the starting vertex:
                Add the starting vertex to the path 'p'.

                If the distance of the current path is 0 or less than the distance of the smallest
                    path:
                   Copy the current path 'p' to the smallest path 'smalp'.

                Remove the starting vertex from the path 'p'.

    Remove the vertex 'ver' from the path 'p'.
    Mark the vertex 'ver' as unvisited in the graph 'g'.

function main(argc, argv):
    Initialize input and output files, and flags for direction and help.

    Process command-line options:
        - If '-d' is present, set the direction flag to true.
        - If '-h' is present, set the help flag to true.
        - If '-o' is present, open the specified output file.
        - If '-i' is present, open the specified input file.
```

```
    If the help flag is true:
        Print usage instructions and command-line options.
        Return 0 to exit the program.

    Read the number of vertices from the input file.
    Create a graph with the specified number of vertices and direction.
    Add vertices to the graph with their names.

    Read the number of edges from the input file.
    Add edges to the graph with their start, end, and weight.

    Close the input file.

    Create an active path and a minimum path, both initialized with the number of vertices in the
         graph.

    Perform depth-first search starting from the starting vertex, updating the minimum path found.

    If no path is found in the minimum path:
        Print a message indicating Alissa is lost.

    Otherwise:
        Add the starting vertex to the minimum path.
        Print the starting point and the path taken by Alissa.
        Print the total distance of the path.

    Free memory allocated for the active and minimum paths, the graph, and close the output file.

    Return 0 to indicate successful program execution.
```

Function Descriptions

For each function in your program, you will need to explain your thought process. This means doing the following

- 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.
- For more complicated functions, include pseudocode that describes how the function works
- For more complicated functions, also include a description of your decision making process; why you chose to use any data structures or control flows that you did.

Do not simply use your code to describe this. This section should be readable to a person with little to no code knowledge. **DO NOT JUST PUT THE FUNCTION SIGNATURES HERE. MORE EXPLANATION IS REQUIRED.**

Graph *graph_create(uint32_t vertices, bool directed) This function takes the number of vertices in the graph and a boolean that tells you if the graph is directed or not. It outputs a pointer to the newly pointed graph. This function is used to create a new graph, it allocates memory for the graph, initializes the vertices, and it initializes an array that tracks the visited vertices.

void graph_free(Graph **gp) This function take a graph pointer and does not output anything. This function frees all of the memory used by the graph including, vertex names, the adjacency matrix, the visited array, and the graph structure itself.

uint32_t graph_vertices(const Graph *g) This function takes a pointer to a graph and outputs the number of vertices as a 32 bit number. This function is used to find the number of vertices in a graph.

void graph_add_vertex(Graph *g, const char *name, uint32_t v) This function takes a pointer to a graph, a name for the vertex, and a vertex. This function names a vertex in the graph.

const char* graph_get_vertex_name(const Graph *g, uint32_t v) This function takes a pointer to a graph and a vertex. This function finds and outputs the name of the vertex given.

char **graph_get_names(const Graph *g) This function take a pointer to a graph and outputs an array of names. This function returns all of the names in the graph.

void graph_add_edge(Graph *g, uint32_t start, uint32_t end, uint32_t weight) This function takes a pointer to a graph, the start and end of an edge, and the weight of the edge. This function adds an edge between the two given vertices and if the graph is undirected the edge goes in both directions.

uint32_t graph_get_weight(const Graph *g, uint32_t start, uint32_t end) This function takes a pointer to a graph and the starting and ending vertice of an edge. This function outputs the weight of the given edge.

void graph_visit_vertex(Graph *g, uint32_t v) This function takes a pointer to a graph and a vertex. This function visits the given vertex by, adding it to the visited array and setting visited to true.

void graph_unvisit_vertex(Graph *g, uint32_t v) This function takes a pointer to a graph and a vertex. This function unvisits the given vertex by, removing it from the visited list and setting the visited to false.

bool graph_visited(const Graph *g, uint32_t v) This function takes a pointer to a graph and a vertex. This function outputs whether or not the given vertex has been visited.

Stack *stack_create(uint32_t capacity) This function takes the capacity for the stack and outputs the pointer to that stack. This function creates a new stack by allocating memory, setting its capacity, and initializing the top pointer to 0.

void stack_free(Stack **sp) This function takes a pointer to a stack. This function frees all of the memory being used by the given stack.

bool stack_push(Stack *s, uint32_t val) This function takes a pointer to a stack and a value. This function pushes a value onto the given stack and outputs whether or not it was successful.

bool stack_pop(Stack *s, uint32_t *val) This function takes a pointer to a stack and a empty integer. This function pops the last item on the given stack and stores the value in the empty integer. It also outputs if the function was successful or not.

bool stack_peek(const Stack *s, uint32_t *val) This function takes a pointer to a stack and a empty integer. This function copies the last number in the given stack to the empty integer and outputs whether or not it was successful.

bool stack_empty(const Stack *s) This function takes a pointer to a stack. This function checks and outputs if the given stack is empty or not.

bool stack_full(const Stack *s) This function takes a pointer to a stack. This function checks and outputs if the given stack is full or not.

uint32_t stack_size(const Stack *s) This function takes a pointer to a stack. It then outputs the number of elements in the stack.

void stack_copy(Stack *dst, const Stack *src) This functions takes two stacks and copies one onto the other.

void stack_print(const Stack *s, FILE *f, char *vals[]) This function takes the input file, stack, and city name. This function prints all of the names in the stack.

Path *path_create(uint32_t capacity) This function takes a number and creates a path, making that number the capacity.

void path_free(Path **pp) This function takes a path and frees both it and its vertices.

uint32_t path_vertices(const Path *p) This function takes a path and returns all of the vertices in the path.

uint32_t path_distance(const Path *p) This function takes a path and returns the weight of it.

void path_add(Path *p, uint32_t val, const Graph *g) This function takes a path, graph, and varible. It uses all of these to create a new path on the given graph.

uint32_t path_remove(Path *p, const Graph *g) This function take a path and graph then removes the path and updates the weights.

void path_copy(Path *dst, const Path *src) This function takes two paths and copys one onto the other.

void path_print(const Path *p, FILE *f, const Graph *g) This function takes a path and graph then prints out the path.