Assignment 6 – Surfin' U.S.A.

Jason Maheru

CSE 13S – Winter 24

Purpose

- The program is designed so that it can use the ADT graphs to find the shortest travel distance on a map. You know the distance from each point to the next.

Questions

- What benefits do adjacency lists have? What about adjacency matrices?
- Which one will you use. Why did we chose that (hint: you use both)?
- If we have found a valid path, do we have to keep looking? Why or why not?
- If we find 2 paths with the same weights, which one do we choose?
- Is the path that is chosen deterministic? Why or why not?
- What type of graph does this assignment use? Describe it as best as you can
- 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?

1. Adjacency lists have many benefits to them. Such as space efficiency for graphs with fewer edges, ease of iteration when it comes to iterating over vertexes in an adjacency list, and will use less memory overall. Adjacency matrices are more space-efficient for graphs with many edges (dense graphs).
2. I will be using both because as stated in the question above it all depends on how edge-weighted that graph is so I will be checking for how edge-weighted the graph is for me to so which adjacney I will be using.
3. If we do find a valid path, I believe that you keep looking for another path because you want to find the shortest path.
4. If we find two paths with the same weight then you want to choose the path with fewer edges.
5. I believe that the deterministic depends if the shortest path is unique or if there are multiple shortest paths. If the path is unique then it is deterministic otherwise, it would be the opposite for the other case.
6. This assignment is based on using undirected weighted graphs. Meaning that the edges in the graph have no direction so you can go from vertex A to vertex B and vice versa. Also, weight is provided which is the distance from one vertex to the next.
7. The edge weights can not be negative and must be finite values. Also if you are traveling back and forth between two cities then the amount traveled between both should be the same.

Testing

- First test would be to use the graphs provided and make sure the output is correct. Test with no inputs, test with a single city, test with two city trips, maximum cities, and maximum edge and negative edge weights. Also, check for error handling.

How to Use the Program

- To run the program it is very straightforward. You first run the Makefile, the compiler, by running the following command `make`. Then you run the program with the following command `./<program-name> <type><graph-file>`.

Program Design

Main Data Structures:

- Graphs: graph data structures are used to represent the transportation network, using adjacency lists and adjacency matrices.
- Edge weight: to represent distances.

Main Algorithms:

- Input parsing: Open, read, and close files that are provided for the program to run.
- DFS: DFS is used for traversing the graph to explore all possible routes or to find specific paths between cities.

Pseudocode

Graph.c

```
Structure:
- Graph
    - uint32_t vertices
    - bool directed
    - bool *visited
    - char **names
    - uint32_t **weights

Functions:
    - graph_create(vertices, directed):
        Create a new graph with the given number of vertices and directedness.
    - graph_free(graph):
        Free the memory allocated for the graph.
    - graph_vertices(graph):
        Return the number of vertices in the graph.
    - graph_add_edge(graph, start, end, weight):
        Add an edge between vertices start and end with the given weight.
    - graph_get_weight(graph, start, end):
        Get the weight of the edge between vertices start and end.
    - graph_visit_vertex(graph, vertex):
        Mark a vertex as visited.
    - graph_unvisit_vertex(graph, vertex):
        Mark a vertex as unvisited.
    - graph_visited(graph, vertex):
        Check if a vertex has been visited.
    - graph_get_names(graph):
        Get the array of vertex names.
    - graph_add_vertex(graph, name, vertex):
        Add a vertex with a given name.
    - graph_get_vertex_name(graph, vertex):
        Get the name of a vertex.
```

Path.c

```
Structure:
- Path
    - uint32_t total_weight
    - Stack *vertices

Functions:
    - path_create(capacity):
        Create a new path with the given capacity.
    - path_free(path):
        Free the memory allocated for the path.
    - path_vertices(path):
        Get the number of vertices in the path.
    - path_distance(path):
        Get the total weight of the path.
    - path_add(path, vertex, graph):
        Add a vertex to the path, updating the total weight.
    - path_remove(path, graph):
        Remove a vertex from the path, updating the total weight.
    - path_copy(dst, src):
        Copy the contents of one path to another.
    - path_print(path, file, graph):
        Print the path, including vertex names and total weight.
```

Stack.c

```
Structure:
- Stack
    - uint32_t capacity
    - uint32_t top
    - uint32_t *items

Functions:
    - stack_create(capacity):
        Create a new stack with the given capacity.
    - stack_free(stack):
        Free the memory allocated for the stack.
    - stack_push(stack, value):
        Push a value onto the stack.
    - stack_pop(stack, value):
        Pop a value from the stack.
    - stack_peek(stack, value):
        Get the value at the top of the stack without removing it.
    - stack_empty(stack):
        Check if the stack is empty.
    - stack_full(stack):
        Check if the stack is full.
    - stack_size(stack):
        Get the number of elements in the stack.
    - stack_copy(dst, src):
        Copy the contents of one stack to another.
    - stack_print(stack, file, values[]):
        Print the contents of the stack, using values from an array.
```

Tsp.c

```
#include graph.h
#include path.h
#include stack.h
#include vertices.h

#define HELPMESSAGE ...

function dfs(vertex, graph, current_path, best_path):
    Mark the vertex as visited
    Add the vertex to the current path
    For each adjacent vertex:
        If the vertex is not visited:
            Recursively call dfs on the adjacent vertex
        If the current path contains all vertices and forms a cycle:
            Add the last vertex to complete the cycle
            If the current path is shorter than the best path:
                Copy the current path to the best path
            Remove the last vertex from the current path
    Mark the vertex as unvisited
    Return the best path found

function main(argc, argv):
    Initialize input and output files
    Parse command-line arguments
    Read number of vertices from input file
    Create a new graph with the specified number of vertices
    Read vertex names from input file and add them to the graph
    Read edges from input file and add them to the graph
    Create empty path and best path
    Perform DFS to find the shortest path
    Print the best path to the output file
    Close input and output files
    Free memory allocated for paths and graph
    Return 0
```

Function Descriptions

graph.c

```
Graph *graph_create(uint32_t vertices, bool directed)
    - Create a new graph struct and return a pointer

void graph_free(Graph **gp)
    - here you want to free all memory used by the graph

uint32_t graph_vertices(const Graph *g)
    - find the number of vertices in a graph

void graph_add_vertex(Graph *g, const char *name, uint32_t v)
    - makes a copy of name and stores it in the graph object

const char* graph_get_vertex_name(const Graph *g, uint32_t v)
    - gets the name of the city with vertex v from the array of city names and return type is const

char **graph_get_names(const Graph *g)
    - gets the names of the every city in an array and returns a double pointer

void graph_add_edge(Graph *g, uint32_t start, uint32_t end, uint32_t weight)
    - Adds an edge between start and end with weight weight to the adjacency matrix of the graph.
```

```
uint32_t graph_get_weight(const Graph *g, uint32_t start, uint32_t end)
   - Looks up the weight of the edge between start and end and returns it.

void graph_visit_vertex(Graph *g, uint32_t v)
   - Adds the vertex v to the list of visited vertices.

void graph_unvisit_vertex(Graph *g, uint32_t v)
   - Removes the vertex v from the list of visited vertices.

bool graph_visited(const Graph *g, uint32_t v)
   - Returns true if vertex v is visited in graph g, false otherwise.
```

stack.c

```
(BASIC STACK IMPLEMENTATION FROM B4)
Stack *stack_create(uint32_t capacity);

void stack_free(Stack **sp);

bool stack_push(Stack *s, uint32_t val);

bool stack_pop(Stack *s, uint32_t *val);

bool stack_peek(const Stack *s, uint32_t *val);

bool stack_empty(const Stack *s);

bool stack_full(const Stack *s);

uint32_t stack_size(const Stack *s);

void stack_copy(Stack *dst, const Stack *src);

void stack_print(const Stack *s, FILE *f, char *vals[]);
```

path.c

```
Path *path_create(uint32_t capacity)
   - Creates a path data structure, containing a Stack and a weight of zero.

void path_free(Path **pp)
   - Frees a path, and all its associated memory.

uint32_t path_vertices(const Path *p)
   - Finds the number of vertices in a path.

uint32_t path_distance(const Path *p)
   - Finds the distance covered by a path.

void path_add(Path *p, uint32_t val, const Graph *g)
   - Adds vertex val from graph g to the path. This function must also update the distance and
       length of the path. When adding a vertex to an empty path, the distance should remain zero.
        Otherwise, you must look up the distance from the most recent vertex to the new one and
       add that to the total weight. The distance can be non-zero only when there are at least two
        cities in the path.

uint32_t path_remove(Path *p, const Graph *g)
   - Removes the most recently added vertex from the path. This function must also update the
       distance and length of the path based on the adjacency matrix in the graph pointed to by g.
        When removing the last vertex from a path, the distance should become zero. The distance
       can be non-zero only when there are at least two cities in the path. Since the return value
        of this function was not specified in the original version of this assignment, we will not
        require you to return anything specific. However, youll probably find that returning the
       index of the removed vertex is the most useful for your DFS implementation.

void path_copy(Path *dst, const Path *src)
   - Copies a path from src to dst.

void path_print(const Path *p, FILE *outfile, const Graph *g)
   - Prints the path stored, using the vertex names from g, to the file outfile. See the Section 8
        for the exact form of the print statement. This function should only print the names of
       the vertices. The rest of the output is produced by tsp.c.
```

tsp.c

```
Help Message

Path *dfs(uint32_t v, Graph *g, Path *p, Path *c)
    - This function is a depth-first search (DFS) algorithm implemented.
    It takes four parameters:
    v: The current vertex being visited.
    g: A pointer to the graph structure.
    p: A pointer to the current path being explored.
    c: A pointer to the current best path found so far.
    - Returns the current best path c.

int main(int argc, char **argv)
    - This is the main entry point of the program.
    - It starts by setting default input and output files to stdin and stdout, respectively.
    - It parses command-line arguments using getopt() to specify input and output files and whether
         the graph should be directed (-i, -o, -d options).
    - It prints the final path to the output file.
    - Finally, it closes input and output files, frees memory allocated for paths and the graph,
        and returns 0 to indicate successful execution.
```

References