Assignment 5 - Surfin' U.S.A.

Aiden Trager

CSE 13S – Fall 2023

Purpose

The purpose of this code is to assist Alissa, a college student on a budget, in planning a trip to visit various cities mentioned in the Beach Boys' song "Surfin' U.S.A." The goal is to find the most mileage-efficient route that starts and ends in Santa Cruz, visiting each city exactly once. The code utilizes graph theory, depth-first search, and a stack to implement a solution to the Travelling Salesman Problem. This optimization helps Alissa minimize the amount of miles needed for her journey. The code provides an efficient solution to a real-world travel problem using graph algorithms.

How to Use the Program

In order to run the program first make the program and then run it. These are what the commands should look like in your terminal:

```
make all
```

Followed by:

```
./tsp *Flags may be needed*
```

The flags necessary can be described by the output of the -h flag, which gives:

Usage: ./tsp [options]

Options:

-d Specify the graph to be directed.

-i input_file Specify the input file path containing the cities and edges of a graph.

-o output_file Specify the output file path to print to.

-h Print out a help message describing the purpose of the program and the command-line options it accepts, exiting the program afterwards.

If no flags are used then the program will read from stdin once you type ./tsp.

Program Design

Data Structures

Main Data Structures: The program utilizes three main data structures - Graph, Stack, and Path.

1. Graph:
    - Purpose: Represents the map of cities and the connections between them.
    - Components:
        - `vertices`: Number of cities in the graph.

- `directed`: Indicates whether the graph is directed or undirected.
- `visited`: Array to track visited vertices during graph traversal.
- `names`: Array of strings holding names of vertices.
- `weights`: Adjacency matrix storing edge weights between vertices.

- Functions:
  - `graph_create(uint32_t vertices, bool directed)`: Creates a new graph.
  - `graph_free(Graph **gp)`: Frees memory used by the graph.
  - `graph_vertices(const Graph *g)`: Finds the number of vertices in a graph.
  - `graph_add_vertex(Graph *g, const char *name, uint32_t v)`: Adds a vertex with a given name.
  - `graph_get_vertex_name(const Graph *g, uint32_t v)`: Gets the name of a vertex.
  - `graph_add_edge(Graph *g, uint32_t start, uint32_t end, uint32_t weight)`: Adds an edge between vertices.
  - `graph_get_weight(const Graph *g, uint32_t start, uint32_t end)`: Gets the weight of an edge.
  - `graph_visit_vertex(Graph *g, uint32_t v)`: Marks a vertex as visited.
  - `graph_unvisit_vertex(Graph *g, uint32_t v)`: Marks a vertex as unvisited.
  - `graph_visited(Graph *g, uint32_t v)`: Checks if a vertex is visited.
  - `graph_print(const Graph *g)`: Prints a human-readable representation of the graph.

2. Stack:

- Purpose: Represents a stack used to track Alissa's path during the TSP solution.
- Components:
  - `capacity`: Maximum capacity of the stack.
  - `top`: Index pointing to the top of the stack.
  - `items`: Array to store stack elements.
- Functions:
  - `stack_create(uint32_t capacity)`: Creates a new stack.
  - `stack_free(Stack **sp)`: Frees memory used by the stack.
  - `stack_push(Stack *s, uint32_t val)`: Adds an element to the top of the stack.
  - `stack_pop(Stack *s, uint32_t *val)`: Removes the top element from the stack.
  - `stack_peek(const Stack *s, uint32_t *val)`: Gets the top element without removing it.
  - `stack_empty(const Stack *s)`: Checks if the stack is empty.
  - `stack_full(const Stack *s)`: Checks if the stack is full.
  - `stack_size(const Stack *s)`: Returns the number of elements in the stack.
  - `stack_copy(Stack *dst, const Stack *src)`: Copies elements from one stack to another.
  - `stack_print(const Stack* s, FILE *outfile, char *cities[])`: Prints the stack elements.

3. Path:

- Purpose: Represents a path in the TSP solution, including the total weight and vertices.
- Components:
  - `total_weight`: Total weight of the path.
  - `vertices`: Stack representing the path.
- Functions:

- `path_create(uint32_t capacity)`: Creates a new path.
- `path_free(Path **pp)`: Frees memory used by the path.
- `path_vertices(const Path *p)`: Finds the number of vertices in a path.
- `path_distance(const Path *p)`: Finds the distance covered by a path.
- `path_add(Path *p, uint32_t val, const Graph *g)`: Adds a vertex to the path.
- `path_remove(Path *p, const Graph *g)`: Removes the most recently added vertex from the path.
- `path_copy(Path *dst, const Path *src)`: Copies a path from source to destination.
- `path_print(const Path *p, FILE *outfile, const Graph *g)`: Prints the path.

Algorithms

The main algorithm is the depth first search algorithm. It explores a graph by visiting as far as possible along each branch before backtracking. In the context of the TSP, it is used to find all possible paths that Alissa can take to visit each city exactly once.

This is the psudocode given in the assigment:

```
function dfs(node n, graph g):
    mark n as visited
    for every edge e from n:
        if e is not visited:
            dfs(e, g)
    mark n as unvisited
```

Results

The code works correctly finding the route that Alissa should take if she wants to go the least distance and still go to every city. This assignment was definitely the hardest for me so far. For some reason it was very challenging for me to conceptualize and apply the depth first search to this problem. After attending a multitude of tutoring sessions and using two late days I now feel that I have a good grip on the concepts present in this assignment. I have implemented the depth first search algorithm as it was intended and my code will find the best route.