Assignment 6 – Surfin' U.S.A. Report

Jackson Dawson

CSE 13S – Winter 24

Purpose

The "tsp" program takes in information describing a graph, and prints out the fastest path to traverse from a given starting index, to every other adjacent vertex, and back again. It can take command line options to specify where you want to provide input from, where you want the output to go, or whether you want to toggle directed/undirected graphs. It requires specifically formatted input, as later outlined in this document. This program utilizes stack, graph, and path data structures, as well as a depth-first-search algorithm in order to effectively and efficiently provide accurate results.

Questions

- What benefits do adjacency lists have? What about adjacency matrices?

  Adjacency lists can be a really effective way to store small graphs. They are simpler to implement and maintain, and arguably to interpret. Adjacency matrices are a bit more complicated, and thus require a bit more complication in their implementation. However, adjacency matrices are more effective for storing large graph information.

- Which one will you use? Why did we choose that? (hint: you use both)

  I'm 99.99% positive I only used an adjacency matrix... uh oh. It works well and intuitively. I suppose I could have used an adjacency list but the matrix maybe gave us a reason to really index into things?

- If we have found a valid path, do we have to keep looking? Why or why not?

  We need to continue looking, as there may be a more efficient path.

- If we find 2 paths with the same weights, which one do we choose?

  As instructed in the assignment PDF, we should use the first one.

- Is the path that is chosen deterministic? Why or why not?

  The path that is chosen is deterministic. In fact, the exact order in which the entire program traverses the graph is deterministic. Each recursive call in my implementation will look at whether any of the other nodes are adjacent nodes, and will explore them in a predictable, sequential order.

- 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?

  The edge weights cannot be negative and they must be integers. I am not entirely sure what to do with this information that is to my advantage, other than knowing that I don't need to double check about the quantity of my "distance" variable.

Testing

I will test TSP using bash scripts to compare my implementation's output with the output of the included binary. I will use each of the provided "maps" to ensure thorough testing. Further, I will ensure that a few duplicate tests are run with the -i, -d, -o, and -h options selected. Unit testing will be administered for the stack, path and graph implementations as well. I will create a C file that includes several key functions from each ADT, create objects, manipulate their data, and attempt to free each object. These tests will be compiled as part of "make tests" in the Makefile, and will be checked for memory leaks using Valgrind. I created a few custom maps as well to really get an understanding of the ideal behavior.

How to Use the Program

Start by compiling the program to make it usable. This is achieved by typing the following commands into the terminal:

```
make clean
make format
make
```

Now the program "tsp" is ready to use. You may run "tsp" with any of the following options:

- **-i:** Sets the file to read from (input file). Requires a filename as an argument. The default file to read from is STDIN.[1]
- **-o:** Sets the file to write to (output file). Requires a filename as an argument. The default file to write to is STDOUT.[1]
- **-d:** Treats all graphs as directed.[1]
- **-h:** Prints a help message to STDOUT.[1]

An example execution may look like the following, where "tsp" will read from STDIN, print to STDOUT, and treat given graphs as directed.

```
./tsp -d
```

"tsp" now expects a graph to be provided in the following format:[1]

```
2
Home
The Beach
3
0 1 1
1 0 2
1 1 0
```

"tsp" will now print the most efficient path according to the provided graph, such as:

```
Alissa starts at:
Home
The Beach
Home
Total Distance: 3
```

To clean up after yourself, do:

```
make clean
```

You're all done!

[1] "2" is the number of vertices, followed by the names of each vertex. "3" is the number of edges in this graph, and the following lines denote [starting vertex] [ending vertex] [respective weight].

Program Design

Main Data Structures:

- Stack ADT
  - Implemented to manage vertices in the Path ADT. It uses an array to allow push and pop operations, enabling last-in, first-out (LIFO) access to elements. This structure is essential for tracking the sequence of vertices traversed in a path, especially when adding or removing vertices, as it supports operations like stack_push, stack_pop, stack_peek, and stack_empty. The Stack is used within the Path structure to hold vertex identifiers as the path progresses through the graph.
- Graph ADT
  - A complex data structure used to represent weighted, directed or undirected graphs. It consists of a set of vertices (nodes) connected by edges with associated weights, representing the distance between nodes. This structure uses an adjacency matrix to store edge weights, a boolean array to track visited vertices, and an array of strings for vertex names. Key operations include graph_add_edge for adding edges, graph_get_weight for retrieving edge weights, graph_add_vertex for adding vertex names, and methods for visiting and unvisiting vertices.
- Path ADT
  - Encapsulates a route through the graph, tracking the total weight (distance) and the sequence of vertices that comprise the path. It leverages the Stack ADT to manage the sequence of vertices, enabling dynamic addition and removal as the path evolves. The Path structure includes operations for creating and freeing paths, adding and removing vertices (with automatic weight adjustment based on the graph's edge weights), and copying paths.

Main Algorithms:

- TSP Implementation
  - The TSP program utilizes a Depth-First-Search (DFS) algorithm to traverse the provided graph, ensuring that every possible path from the starting vertex is attempted. The process is generally: get input from denoted datastream, create a graph representing the graph described in the input, traverse the graph and return the shortest path
- Depth-First Search (DFS)
  - This algorithm is broken up into two functions; the first allocates space for recording the best path, the current path, and the recorded minimum path distance (weight). Then it calls the actual DFS function. This function marks its given vertex as visited, adds the vertex to the path, and continues to add its adjacent vertices to the path as long as it can by calling itself. If the path contains all of the vertices, it attempts to return to the start and claim the new path as the best path. If it doesn't, it will backtrack one node at a time and look for new paths to go down.

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.

The program follows a structured flow:

1. Interpret selected command line options, adjusting for varied input/output procedures and/or toggling (un)directed graph mechanics.
2. Take in input from the user, either via input file or STDIN.
3. Represent graph using adjacency matrix.

4. Traverse graph, recording and updating the fastest path:
   (a) Initialize two paths: currentPath and bestPath, with bestPath intended to store the shortest path found.
   (b) Sets an initial minimum path weight to the maximum unsigned 32-bit integer value to compare against paths found.
   (c) Marks all vertices as unvisited before starting the search.
   (d) Performs a DFS from a starting vertex to find the shortest path that visits all vertices exactly once and returns to the starting point.
   (e) Utilizes a recursive helper function tsp_dfs_visit to explore paths, mark vertices as visited, add vertices to the current path, and backtrack when necessary.
   (f) Checks if a complete tour is possible when all vertices have been visited, and updates the bestPath and minPathWeight if a better path is found.
   (g) Returns best path
5. Prints the path to the correct output file
6. Free the graph and the returned path

Function Descriptions

- **Function:** `get_graph_from_stream`
  - **Inputs:** A file stream (`FILE*`) from which to read the graph data, and a boolean indicating whether the graph is directed.
  - **Outputs:** A pointer to a `Graph` structure populated with vertices and edges as read from the stream.
  - **Purpose:** Reads graph data from a given stream and constructs a graph by adding vertices and edges as specified in the data.
- **Function:** `tsp_dfs`
  - **Inputs:** A pointer to a `Graph` structure and a starting vertex (`uint32_t`) for the DFS algorithm.
  - **Outputs:** A pointer to a `Path` structure representing the best path found, or `NULL` if no path is found.
  - **Purpose:** Implements a depth-first search to find the shortest path that visits all vertices in a graph starting from a given vertex.
- **Function:** `tsp_dfs_visit`
  - **Inputs:** A pointer to the `Graph` structure, the current vertex being visited (`uint32_t`), the starting vertex (`uint32_t`), pointers to the current and best path (`Path*`), and a pointer to track the minimum path weight (`uint32_t*`).
  - **Outputs:** None directly, but modifies the best path and minimum path weight as it explores the graph.
  - **Purpose:** Recursively explores all possible paths starting from a given vertex to find the shortest possible path that visits all vertices, updating the best path and minimum path weight accordingly.
- **Function:** `hasIsolation`
  - **Inputs:** A pointer to a `Graph` structure.
  - **Outputs:** An integer indicating whether the graph has isolated vertices (returns `true` if isolation is found, otherwise `false`).

- **Purpose:** Checks the graph for isolated vertices, which would make it impossible to find a path that visits all vertices.

- **Function:** `main`
  - **Inputs:** Command-line arguments including options for input and output file names, and whether the graph is directed.
  - **Outputs:** The function returns an integer indicating the success or failure of the program's execution.
  - **Purpose:** Acts as the entry point for the program, handling command-line arguments, constructing the graph from input, finding the shortest path using TSP DFS, and outputting the results.

- **Function:** `path_create`
  - **Inputs:** Capacity (the maximum number of vertices the path can contain).
  - **Outputs:** A pointer to a newly created Path structure.
  - **Purpose:** Initializes a Path structure with a specified capacity, setting its total weight to 0 and creating an empty stack for vertices.

- **Function:** `path_free`
  - **Inputs:** A double pointer to the Path structure.
  - **Outputs:** None (frees allocated memory).
  - **Purpose:** Cleans up and deallocates memory used by a Path structure, ensuring no memory leaks.

- **Function:** `path_add`
  - **Inputs:** A Path structure to which a vertex is added, the vertex's value, and a reference to the Graph structure for weight calculation.
  - **Outputs:** None (modifies the Path structure by adding a vertex and updating the total weight).
  - **Purpose:** Adds a new vertex to the path, calculating the weight from the last vertex to this new one using the graph's weight data.

- **Function:** `path_remove`
  - **Inputs:** A Path structure from which the last vertex is removed, and a reference to the Graph structure for weight adjustment.
  - **Outputs:** The value of the removed vertex.
  - **Purpose:** Removes the most recently added vertex from the path, adjusting the total weight accordingly.

- **Function:** `path_vertices`
  - **Inputs:** A Path structure.
  - **Outputs:** The number of vertices in the path.
  - **Purpose:** Returns the count of vertices currently in the path.

- **Function:** `path_distance`
  - **Inputs:** A Path structure.
  - **Outputs:** The total weight of the path.
  - **Purpose:** Provides the total weight (or distance) of the path.

- **Function:** `path_copy`
  - **Inputs:** Destination and source Path structures.
  - **Outputs:** None (copies the content of the source Path into the destination Path).
  - **Purpose:** Duplicates the vertices and total weight from one path to another.
- **Function:** `path_print`
  - **Inputs:** A Path structure, a file stream for output, and a Graph structure.
  - **Outputs:** None (prints the path with vertices and weights to the specified file stream).
  - **Purpose:** Displays the sequence of vertices in the path along with the weights of the edges between them, formatted for readability.
- **Function:** `graph_create`
  - **Inputs:** The number of vertices and a boolean indicating whether the graph is directed.
  - **Outputs:** A pointer to a newly created Graph structure.
  - **Purpose:** Initializes a Graph structure, allocating memory for vertex names, an adjacency matrix for weights, and a visited flag array.
- **Function:** `graph_free`
  - **Inputs:** A double pointer to the Graph structure.
  - **Outputs:** None (frees allocated memory).
  - **Purpose:** Deallocates all memory associated with a Graph structure, including its adjacency matrix, vertex names, and visited flags.
- **Function:** `graph_add_edge`
  - **Inputs:** A Graph structure, start vertex, end vertex, and the weight of the edge to be added.
  - **Outputs:** None (modifies the Graph structure by adding or updating an edge's weight).
  - **Purpose:** Adds or updates an edge in the graph with a specified weight, adjusting the adjacency matrix accordingly.
- **Function:** `graph_get_weight`
  - **Inputs:** A Graph structure and two vertices (start and end).
  - **Outputs:** The weight of the edge between the specified vertices.
  - **Purpose:** Retrieves the weight of an edge between two vertices.
- **Function:** `graph_vertices`
  - **Inputs:** A Graph structure.
  - **Outputs:** The number of vertices in the graph.
  - **Purpose:** Returns the total number of vertices in the graph.
- **Function:** `graph_visit_vertex`
  - **Inputs:** A Graph structure and a vertex index.
  - **Outputs:** None (marks a vertex as visited).
  - **Purpose:** Flags a vertex as visited in the graph's visited array.
- **Function:** `graph_unvisit_vertex`
  - **Inputs:** A Graph structure and a vertex index.

  - **Outputs:** None (marks a vertex as not visited).
  - **Purpose:** Resets a vertex's visited status, allowing it to be visited again in future traversals.

- **Function:** `graph_visited`
  - **Inputs:** A Graph structure and a vertex index.
  - **Outputs:** A boolean indicating whether the vertex has been visited.
  - **Purpose:** Sets a given vertex to visited.

- **Function:** `graph_get_names`
  - **Inputs:** A pointer to the Graph structure.
  - **Outputs:** An array of strings containing the names of the vertices in the graph.
  - **Purpose:** Retrieves an array of vertex names, allowing for easy access to vertex identifiers based on their position within the graph.

- **Function:** `graph_add_vertex`
  - **Inputs:** A pointer to the Graph structure, a string representing the name of the vertex, and the vertex's numerical identifier.
  - **Outputs:** None (modifies the Graph structure by associating a name with a vertex identifier).
  - **Purpose:** Associates a human-readable name with a specific vertex in the graph.

- **Function:** `graph_get_vertex_name`
  - **Inputs:** A pointer to the Graph structure and a vertex's numerical identifier.
  - **Outputs:** The name associated with the given vertex identifier.
  - **Purpose:** Retrieves the name of a vertex based on its numerical identifier.

- **Function:** `graph_print`
  - **Inputs:** A pointer to the Graph structure.
  - **Outputs:** None (prints the graph structure to the standard output).
  - **Purpose:** Displays the entire graph structure, including vertex names and the weights of edges between vertices.

- **Function:** `stack_create`
  - **Inputs:** Capacity (the maximum number of elements the stack can hold).
  - **Outputs:** A pointer to a newly created Stack structure.
  - **Purpose:** Initializes and allocates memory for a Stack structure with a specified capacity.

- **Function:** `stack_free`
  - **Inputs:** A double pointer to the Stack structure.
  - **Outputs:** None (frees the allocated Stack structure).
  - **Purpose:** Deallocates memory for the Stack structure, ensuring efficient resource management and preventing memory leaks.

- **Function:** `stack_push`
  - **Inputs:** A pointer to the Stack structure and the value to be pushed onto the stack.
  - **Outputs:** A boolean indicating success or failure of the operation.
  - **Purpose:** Adds a new element to the top of the stack.

- **Function:** `stack_pop`
  - **Inputs:** A pointer to the Stack structure and a pointer to store the value popped from the stack.
  - **Outputs:** A boolean indicating success or failure of the operation.
  - **Purpose:** Removes the top element from the stack and retrieves its value.
- **Function:** `stack_peek`
  - **Inputs:** A pointer to the Stack structure and a pointer to store the value of the top element without removing it.
  - **Outputs:** A boolean indicating success or failure of the operation.
  - **Purpose:** Retrieves the value of the top element without removing it from the stack, allowing inspection of the most recent element without altering the stack's state.
- **Function:** `stack_empty`
  - **Inputs:** A pointer to the Stack structure.
  - **Outputs:** A boolean indicating whether the stack is empty.
  - **Purpose:** Checks if the stack has no elements.
- **Function:** `stack_full`
  - **Inputs:** A pointer to the Stack structure.
  - **Outputs:** A boolean indicating whether the stack is full.
  - **Purpose:** Determines if the stack has reached its capacity.
- **Function:** `stack_size`
  - **Inputs:** A pointer to the Stack structure.
  - **Outputs:** The number of elements currently in the stack.
  - **Purpose:** Provides the current size of the stack.
- **Function:** `stack_copy`
  - **Inputs:** Destination Stack structure pointer and Source Stack structure pointer.
  - **Outputs:** None (copies the content of the source stack to the destination stack).
  - **Purpose:** Duplicates the elements from the source stack to the destination stack.
- **Function:** `stack_print`
  - **Inputs:** A pointer to the Stack structure, a file pointer for output, and an array of strings representing the values in the stack.
  - **Outputs:** None (prints the stack's elements to the specified file).
  - **Purpose:** Outputs a visual representation of the stack's current state, including its elements, to a file or standard output.

References

[1] Jess Srinivas and Ben Grant. Surfin' u.s.a. CSE 13S Winter 2024 Course Assignment, University of California, Santa Cruz, 2024. Based on "The Perambulations of Denver Long" by Professor Darrell Long.