Assignment 5 - Surfin' U.S.A

Anish Talluri

CSE 13S – Fall 2023

Purpose

Designed to crack the Traveling Salesman Problem, this program employs graph theory, depth-first search, and a stack for efficient route optimization. Tailored for budget-conscious users navigating city routes, it guarantees an optimal journey starting and concluding in Santa Cruz, with each city visited precisely once. Whether you're exploring coastal landscapes or grappling with similar logistical challenges, this tool stands ready to provide a practical and effective solution, ensuring a seamless and cost-conscious experience.

How to Use the Program

To utilize the Traveling Salesman Program (TSP), you first need to compile the source code using the provided Makefile. Ensure all source code files (`tsp.c, graph.c, stack.c, path.c`) and header files (graph.h, stack.h, path.h, vertices.h) are in your working directory. Create a Makefile with the specified compiler and flags. Execute make all to compile the program. Once compiled, run the program with `./tsp [OPTIONS] -i input_file`, where `[OPTIONS]` include `-o output_file` to specify an output file, `-d` to treat graphs as directed, and `-h` for help. For example, `./tsp -i maps/basic.graph` finds the shortest path in an undirected graph, `./tsp -d -i maps/directed.graph` considers directed edges, and `./tsp -i maps/undirected.graph -o output.txt` saves results to a file. The program outputs Alissa's journey and total distance, displaying error messages if issues arise. Experiment with different graphs and options for optimal paths. If errors occur, check the displayed messages for guidance.

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.

Data Structures

Describe your data structures here. Data structures include things like arrays, enums, strings, and structs. You should also mention why you chose the data structures that you did.

The main data structures in the Traveling Salesman Program (TSP) include:

1. Graph Structure: Represents the graph, storing information about vertices, edges, and weights. The adjacency matrix is implemented for efficient graph representation. The adjacency matrix allows quick access to edge weights and facilitates graph traversal

2: Stack Structure: Used to track the path Alissa takes during her trip. Dynamic memory allocation for the array of items within the stack allows efficient management of the path. The stack follows a last-in-first-out (LIFO) ordering, providing a suitable structure for path tracking.

3. Path Structure: Represents a path in the TSP, storing the total weight and a stack of vertices. The path structure is designed to keep track of the distance traveled and the sequence of vertices visited.

4.

Algorithms

For the algorithms section, you want to show psuedocode. Psuedocode should not be able to run on any computer, but should be written in plain english so that someone who knows nothing about computer science can follow along.

Want to show some pseudocode? Use the framed verbatim text shown above.

```
Function main(argc, argv):
    Initialize default program options (input_file = stdin, output_file = stdout,
    directed = false)

    Parse command-line options and update program options accordingly

    If help option present or no input file specified:
        Print help message to stderr
        Exit with success code

    Initialize Graph data structure (graph)
    Read graph data from input file and populate the graph

    If directed option set:
        Mark graph as directed

    Initialize Path data structure (current_path)
    Initialize Path data structure (min_path) to store the best path found
    Set total_weight of min_path to a large value

    Perform DFS to find all possible Hamiltonian cycles starting from START_VERTEX
    For each Hamiltonian cycle found:
        Add the cycle to current_path
        If current_path is valid:
            If total_weight of current_path is less than total_weight of min_path:
                Copy current_path to min_path

    If a valid path found (min_path updated):
        Print vertices of min_path to output file (specified or stdout)
        Print total distance traveled by Alissa
    Else:
        Print "No path found! Alissa is lost!" to outfile

    Free memory allocated for graph and paths
    Exit with success code
```

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.
- For more complicated functions, include psuedocode 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.
- Stack* stack_create(uint32_t capacity): Creates a new stack with the specified initial capacity and returns a pointer to it. This function dynamically allocates memory for the stack and initializes its fields
- void stack_free(Stack** sp): Frees the memory allocated for the given stack. The double pointer allows the function to set the original pointer to NULL after freeing
- bool stack_push(Stack* s, uint32_t val): Pushes a value onto the stack. Returns true if the push is successful and false otherwise
- bool stack_pop(Stack* s, uint32_t* val): Pops the top element from the stack and stores it in the provided pointer. Returns true if the pop is successful and false otherwise
- bool stack_peek(const Stack* s, uint32_t* val): Retrieves the top element from the stack without removing it. Returns true if the peek is successful and false otherwise
- bool stack_empty(const Stack* s): Checks if the stack is empty. Returns true if the stack has no elements and false otherwise
- bool stack_full(const Stack* s): Checks if the stack is full. Returns true if the stack has reached its capacity and false otherwise
- uint32_t stack_size(const Stack* s): Returns the number of elements in the stack. Provides the count of elements
- void stack_copy(Stack* dst, const Stack* src): Copies all items from the source stack to the destination stack, ensuring the destination has sufficient capacity
- void stack_print(const Stack* s, FILE* outfile, char* cities[]): Prints the stack as a list of elements, using city names based on the indices stored in the stack
- Path* path_create(uint32_t capacity): Creates a new path with the specified initial capacity and returns a pointer to it. This function dynamically allocates memory for the path and initializes its fields, including the associated stack
- void path_free(Path** pp): Frees the memory allocated for the given path. The double pointer allows the function to set the original pointer to NULL after freeing
- uint32_t path_vertices(const Path* p): Returns the number of vertices in the path. Provides the count of vertices
- uint32_t path_distance(const Path* p): Returns the total distance covered by the path. Retrieves the total weight of the path
- void path_add(Path* p, uint32_t val, const Graph* g): Adds a vertex to the path, updating the distance based on the adjacency matrix in the graph
- uint32_t path_remove(Path* p, const Graph* g): Removes the most recently added vertex from the path, updating the distance based on the adjacency matrix in the graph. Returns the index of the removed vertex
- void path_copy(Path* dst, const Path* src): Copies all items from the source path to the destination path, ensuring the destination has sufficient capacity
- void path_print(const Path* p, FILE* outfile, const Graph* g): Prints the path as a list of vertices, using city names based on the adjacency matrix in the graph

- int main(int argc, char* argv[]): The main function that orchestrates the execution of the Traveling Salesman Program. Parses command-line options, reads graph data, performs DFS, and prints the best path. Returns an exit code
- void dfs(Path* current, Path* shortest, Graph* g, uint32_t v, FILE* ofile): This function performs a Depth-First Search (DFS) traversal on a graph, tracking the shortest path between vertices. It explores all possible paths from a given vertex ('v') within the graph ('g') and stores the shortest path found in the 'shortest' path object while updating the 'current' path during exploration

Error Handling

Error handling in this code primarily revolves around validating file input operations, ensuring file handles are successfully opened, and verifying the correctness of user-provided arguments. Upon encountering file-related issues, the code effectively identifies and reports errors to stderr, ensuring users are notified of any unexpected behavior. However, the error handling strategy could be bolstered by implementing more granular error messages that pinpoint specific issues encountered during file operations or command-line argument parsing. Incorporating additional checks to handle exceptional cases within the graph manipulation or path-finding routines would further fortify the program's resilience and improve its ability to gracefully handle unforeseen scenarios. As a result, augmenting the error handling mechanisms would contribute significantly to the code's robustness and user-friendliness

Testing

Testing was primarily conducted using a dedicated 'maps' folder containing various graph files. These files were used as inputs for the program, allowing for comprehensive testing of different graph structures and edge cases. The program's outputs were then compared against an executable reference file, serving as a benchmark to verify the accuracy and correctness of the generated results. This rigorous testing methodology, encompassing diverse graph configurations and scenarios, ensured thorough validation of the program's functionality. By employing these sets of graph inputs and systematically comparing the produced outputs with the reference files, the program's reliability and consistency were extensively verified, contributing to its robustness and accuracy.

Results

While the code effectively finds and prints the shortest path, there is a lack of detailed error handling for scenarios where no valid path is discovered. More descriptive error messages and comprehensive commenting, although utilized for debugging with `printf` statements, would significantly enhance the code's readability and debugging potential for future revisions.

References

-Ben/Jess/Prof. Veenstra for the code provided for multiple functions in the stack.c, path.c, and graph.c
-Ben/Jess/Prof. Veenstra for the code given for file opening, file scanning, and pseudocode for the DFS
-Prathik for providing the structure and understanding on how to make the DFS function

References

```
atalluri@13s:~/cse13s/asgn5$ ./tsp -d -i maps/basic.graph
Alissa starts at:
Home
The Beach
Home
Total Distance: 3
atalluri@13s:~/cse13s/asgn5$
```

```
atalluri@13s:~/resources/asgn5$ ./tsp -d -i maps/basic.graph
Alissa starts at:
Home
The Beach
Home
Total Distance: 3
atalluri@13s:~/resources/asgn5$
```

Figure 1: My output vs. Reference file output for basic.graph

```
atalluri@13s:~/cse13s/asgn5$ ./tsp -d -i maps/bayarea.graph
No path found! Alissa is lost!
atalluri@13s:~/cse13s/asgn5$
```

```
atalluri@13s:~/resources/asgn5$ ./tsp -d -i maps/bayarea.graph
No path found! Alissa is lost!
atalluri@13s:~/resources/asgn5$
```

Figure 2: My output vs. Reference file output for bayarea.graph but with directed flag on

```
atalluri@13s:~/cse13s/asgn5$ ./tsp -i maps/bayarea.graph
Alissa starts at:
Santa Cruz
Half Moon Bay
San Mateo
Daly City
San Francisco
Oakland
Walnut Creek
Dublin
Hayward
San Jose
Santa Cruz
Total Distance: 203
atalluri@13s:~/cse13s/asgn5$
```

```
atalluri@13s:~/resources/asgn5$ ./tsp -i maps/bayarea.graph
Alissa starts at:
Santa Cruz
Half Moon Bay
San Mateo
Daly City
San Francisco
Oakland
Walnut Creek
Dublin
Hayward
San Jose
Santa Cruz
Total Distance: 203
atalluri@13s:~/resources/asgn5$
```

Figure 3: My output vs. Reference file output for bayarea.graph with directed flag off

```
atalluri@13s:~/cse13s/asgn5$ ./tsp -d -i maps/surfin.graph
Alissa starts at:
Santa Cruz
Ventura County Line
Pacific Palisades
Manhattan
Redondo Beach, L.A.
Haggerty's
Sunset
Doheny
Trestles
San Onofre
Swami's
Del Mar
La Jolla
Santa Cruz
Total Distance: 965
atalluri@13s:~/cse13s/asgn5$
```

```
atalluri@13s:~/resources/asgn5$ ./tsp -d -i maps/surfin.graph
Alissa starts at:
Santa Cruz
Ventura County Line
Pacific Palisades
Manhattan
Redondo Beach, L.A.
Haggerty's
Sunset
Doheny
Trestles
San Onofre
Swami's
Del Mar
La Jolla
Santa Cruz
Total Distance: 965
atalluri@13s:~/resources/asgn5$
```

Figure 4: My output vs. Reference file output for surfin.graph