Assignment 6 – Surfin' U.S.A

Debi Majumdar

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.

This code is designed to aid Alissa, a budget-conscious college student, in planning an optimized travel route to visit cities mentioned in the Beach Boys' song "Surfin' U.S.A." It employs graph theory and depth-first search algorithms to find the most mileage-efficient path that starts and ends in Santa Cruz, ensuring Alissa visits each city exactly once. By leveraging these algorithms, the code minimizes the total distance traveled, providing a practical solution to the Travelling Salesman Problem for real-world travel scenarios.

Graphs

Graphs are fundamental data structures used to represent relationships between objects. They consist of vertices (nodes) and edges (connections) that define the interactions between these vertices.In this context, graphs serve as a crucial tool for modeling Alissa's travel itinerary and solving the Travelling Salesman Problem. Each city mentioned in the Beach Boys' song "Surfin' U.S.A." is represented as a vertex, while the connections between cities represent possible routes (edges). By utilizing graph theory and algorithms, such as depth-first search (DFS), the code can efficiently explore these interconnected vertices to find the most mileage-efficient route that starts and ends in Santa Cruz, visiting each city exactly once. Graph algorithms provide the framework for optimizing Alissa's journey, ensuring that she minimizes the amount of miles needed for her trip while adhering to the constraints of the problem. Thus, graphs play a vital role in facilitating the planning and optimization of Alissa's travel itinerary.

This assignment primarily deals with undirected graphs. An undirected graph is a type of graph where edges have no inherent directionality, meaning they represent symmetric relationships between vertices. In the context of Alissa's trip planning, the cities she visits and the distances between them form an undirected graph. Each city corresponds to a vertex, and the distances between cities represent the edges. This type of graph suits the problem well as it allows Alissa to travel between any two cities bidirectionally, reflecting the real-world scenario where roads or paths connect cities without one-way restrictions.

Stacks

Stacks are a fundamental data structure characterized by their Last-In, First-Out (LIFO) behavior, akin to a stack of pancakes. In this assignment, stacks are utilized to track Alissa's travel path efficiently. Each city she visits is pushed onto the stack, representing her current location. When she moves to a new city, the previous city is popped off the stack, simulating her backtrack to the previous location. This stack-based approach allows the code to maintain a record of Alissa's path and efficiently explore different routes using depth-first search (DFS) to find the most mileage-efficient itinerary. Additionally, stacks play a crucial role in the implementation of the path data structure, which stores the vertices visited by Alissa and calculates the total distance covered during her journey. Overall, stacks are essential for managing Alissa's travel path and optimizing her itinerary in this assignment.

Adjacency Lists and Adjacency Matrices

Adjacency lists and adjacency matrices are two common representations of graphs, each offering distinct advantages depending on the specific application. Adjacency lists provide several benefits such as efficient memory usage, flexibility, ease of implementation, and fast iteration over neighbors. They are particularly suitable for sparse graphs where memory efficiency is crucial and for scenarios requiring dynamic graph structures. On the other hand, adjacency matrices offer constant-time edge lookup, making them efficient for dense graphs and facilitating certain graph operations like matrix multiplication. In this assignment, both representations are employed to leverage their respective strengths. Adjacency lists are used to efficiently store the graph structure, especially for sparse graphs, while adjacency matrices are utilized for specific operations like edge existence checks and distance calculations, taking advantage of their constant-time lookup and facilitating certain graph algorithms efficiently. The choice between the two representations depends on the specific operation and the characteristics of the graph being analyzed.

In this assignment, we utilize both adjacency lists and adjacency matrices to represent and analyze the graph structure efficiently. While adjacency lists are preferred for their memory efficiency and ability to handle sparse graphs effectively, adjacency matrices offer advantages in certain operations such as edge existence checks and distance calculations due to their constant-time lookup capability. By leveraging both representations, we can optimize memory usage and algorithm performance, ensuring an efficient solution to the Travelling Salesman Problem while effectively managing the complexity of graph operations.

Paths

Once a valid path has been found, there is no need to continue searching for other paths. This is because the objective of the Travelling Salesman Problem is to find the shortest path that visits each vertex exactly once and returns to the starting vertex. Once such a path is identified, it is guaranteed to be the optimal solution, and any further exploration would only yield paths of equal or greater length, which are not desirable.

If multiple paths with the same weights are found, any of them can be chosen as they all represent equally optimal solutions. In this scenario, the choice of which path to select may depend on other factors such as computational efficiency or specific requirements of the problem.

The path chosen is deterministic in the sense that given the same input and algorithm, it will always produce the same output. However, the exact path chosen may vary depending on the specific implementation of the algorithm, the order of vertices processed, or other factors such as randomization if employed. Therefore, while the selection process itself is deterministic, the outcome may not be unique across different runs.

Constraints and Optimization

The edge weights in the context of Alissa's trip represent the distances between different cities or locations. These distances are constrained by factors such as road conditions, traffic, and fuel efficiency. For Alissa, who is on a budget, minimizing the total distance traveled is crucial. Additionally, considering her limited fuel capacity, optimizing the route to prioritize shorter distances between locations can help minimize fuel consumption and overall cost.

To optimize our depth-first search (DFS) further based on these constraints, we can incorporate heuristics or strategies that prioritize visiting neighboring cities with shorter distances first. This can be achieved by sorting the adjacent vertices based on their distances from the current city before exploring them. Additionally, we can implement pruning techniques to avoid exploring paths that are unlikely to lead to a shorter overall distance. For example, if the current path's distance exceeds the shortest path found so far, we can backtrack early without exploring further branches, thus saving computational resources. By integrating these optimizations into our DFS algorithm, we can efficiently find the most mileage-efficient route for Alissa's trip.

Testing

List what you will do to test your code. Make sure this is comprehensive.

To ensure the reliability of our code, we'll conduct comprehensive testing covering functionality, performance, and edge cases. Unit testing will assess individual functions within the graph, stack, and path ADTs, covering various scenarios and error handling. Integration testing will validate interactions between ADTs, ensuring seamless integration and consistent behavior.

Functional testing will focus on solving the Travelling Salesman Problem with different input graphs, verifying correctness and performance. File I/O testing will ensure proper parsing and writing of graph data. Command-line options testing will validate option handling, including edge cases.

Error handling testing will cover error conditions, while performance testing will measure execution time and memory usage. Edge case testing will assess extreme scenarios, and validation testing will compare results with known solutions. Through this approach, we aim to deliver reliable and efficient code for the Travelling Salesman Problem.

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 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[1][**?**][**?**].

To use our program, start by compiling the source code provided using the Makefile provided in the repository.Ensure you have the necessary dependencies installed, such as a C compiler and standard libraries.

```
make all
```

Once compiled, the program can be executed from the command line with the following options:

- `-i <input_file>`: Specifies the input file containing the graph data. This flag is optional, and if not provided, the program will default to reading from stdin.
- `-o <output_file>`: Specifies the output file where the results will be written. This flag is also optional, and if omitted, the program will default to writing to stdout.
- `-d`: Treats all graphs as directed. By default, the program assumes an undirected graph, meaning that edges are added bidirectionally. Using this flag ensures that only directed edges are added.
- `-h`: Prints a help message to the standard output, providing information on how to use the program and the available options.

Once the program is compiled and executed with the desired options, it will read the input graph, solve the Travelling Salesman Problem, and output the optimal path to the specified output file or stdout. If any errors occur during execution, error messages will be displayed to stderr.

For example, to run the program with an input file named "graph.txt" and direct the output to a file named "output.txt" while treating graphs as directed, you would use the following command:

[1]This is my footnote.

```
./tsp -i graph.txt -o output.txt -d
```

Alternatively, if you want to use stdin for input and stdout for output, you can simply run:

```
./tsp
```

This flexibility allows users to tailor the program's behavior according to their specific requirements and preferences.

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.

0.0.1 Maintenance Guide

When maintaining our program, it's important to understand the organization of the codebase and the key data structures and algorithms used. The functions mentioned here were provided in the assignment pdf as well as their implementations by Jess Srinivas and Ben Grant. Here's an overview:

0.0.2 Data Structures

The main data structures used in our program are:

- **Graph**: Represents the graph data structure and contains information about vertices, edges, and weights. Implemented using adjacency lists to efficiently store and retrieve graph data.

  Functions:

- `graph_create(uint32_t vertices, bool directed)`: This function will help create a new graph struct and return a pointer to it. This will initialize all items in the visited array to be false.
- `graph_free(Graph **gp)`: This function will help free all the memory used by the graph. It will take in a double pointer to that gp pointer is set to and set it to null.
- `graph_vertices(const Graph *g)`: This function finds the number of vertices in a graph.
- `graph_add_vertex(Graph *g, const char *name, uint32_t v)`: This function will give the city at vertex v the name it was passed to. It will then make a copy of the name and store it in the graph object.
- `graph_get_vertex_name(const Graph *g, uint32_t v)`: This function gets the name of a vertex.
- `graph_get_names(const Graph *g)`: This function will only return the names of the graph it is at. It will not return the names of the graph vertex.
- `graph_add_edge(Graph *g, uint32_t start, uint32_t end, uint32_t weight)`: This function will add edge an edge between the start and the end with weight. The weight will be used according to the adjacent matrix of the graph.
- `graph_get_weight(const Graph *g, uint32_t start, uint32_t end)`: This function will help look up the weight of the edge between the start and end. Then it will return that weight.
- `graph_visit_vertex(Graph *g, uint32_t v)`:This function will help add the vertex "v" to the list of visited vertices. It will return true if the vertex is visited in graph g and false if it isn't visited.

- `graph_unvisit_vertex(Graph *g, uint32_t v)`: This function marks a vertex as unvisited.
- `graph_visited(Graph *g, uint32_t v)`: This function checks if a vertex is visited.
- `graph_print(const Graph *g)`: This function will help print the human-readable representation of a graph
- **Stack**: Represents a stack data structure used for tracking the path taken by Alissa during her trip. Implemented using a dynamic array to allow for efficient push and pop operations.

  Functions:
- `stack_create(uint32_t capacity)`: This function creates a stack and also dynamically allocates space for it. This function will return a pointer.
- `stack_free(Stack **sp)`: This function helps free all the space used in a stack and also sets the pointers to NULL.
- `stack_push(Stack *s, uint32_t val)`: This function will be used to help add a value to the top of the stack. This boolean will return true if it was successful and false if the stack was full and did not have enough room to push.
- `stack_pop(Stack *s, uint32_t *val)`: This function will be used to help remove the value positioned at the top of the stack. This boolean will return true upon success and false if the stack was empty.
- `stack_peek(const Stack *s, uint32_t *val)`:This function will be used to check if the value at the top of the stack is stored at the address it is pointed to. This boolean will return true upon success and false if the stack was empty.
- `stack_empty(const Stack *s)`: This function will be used to help determine when a stack is empty, meaning that the stack has no elements in it. This boolean will return true if the stack is empty and false if it is not.
- `stack_full(const Stack *s)`: This function will be used to help determine when a stack is full, meaning that the stack has no more room to add any more elements. This boolean will return true if the stack is full and false if it is not.
- `stack_size(const Stack *s)`: This function will help overwrite the dst with all the items from src, essentially copying them over.
- `stack_copy(Stack *dst, const Stack *src)`: This function will print out the stack as a list of elements given a list of vertex names. This function will be useful in seeing where Alissa went.
- `stack_print(const Stack* s, FILE *outfile, char *cities[])`: This function will print out the stack as a list of elements given a list of vertex names. This function will be useful in seeing where Alissa went.
- **Path**: Represents a path data structure that includes the total weight and a stack of vertices visited. Used to track the shortest path taken by Alissa while considering mileage efficiency.

  Functions:
- `path_create(uint32_t capacity)`: This function will help create a path that contains a stack and a weight of zero. It will be useful in determining the best path for Alissa to take.
- `path_free(Path **pp)`: This function will help free a path and the memory connected to it. That way we can help Alissa find more or different paths.
- `path_vertices(const Path *p)`:This function will help find the distance covered by a path. This will be useful in helping us determine which path has the best distance for Alissa.

- `path_distance(const Path *p)`: This function will help add a vertex from the graph to the path. It will be helpful in updating the distance between two cities to help us determine Alissa's best route.
- `path_add(Path *p, uint32_t val, const Graph *g)`: This function will help remove the most recently added vertex from the path. When it removes the last vertex, the distance between the two cities will be zero. Otherwise, it will update the distance and the length of the adjacency path of the matrix.
- `path_remove(Path *p, const Graph *g)`:This function will be used to copy a path from src to dst. This will be helpful later on to help implement the dst function later on.
- `path_copy(Path *dst, const Path *src)`: This function will be used to print the path stored to the outfile. It will print the names of the vertices.
- `path_print(const Path *p, FILE *outfile, const Graph *g)`: This function will be used to print the path stored to the outfile. It will print the names of the vertices.

0.0.3 Algorithms

The main algorithms used in our program are:

- **Depth-First Search (DFS)**: Used to traverse the graph and find all possible paths that Alissa can take during her trip. Implemented recursively to explore all possible paths efficiently.
- **Travelling Salesman Problem (TSP) Solution**: Utilizes DFS to find the optimal path for Alissa that minimizes mileage while visiting each city exactly once. Involves backtracking to explore all possible paths and selecting the shortest one.

Understanding how these data structures and algorithms interact with each other is essential for maintaining and debugging the program effectively. Additionally, familiarity with the specific implementation details and any dependencies used will aid in troubleshooting and making improvements to the codebase.

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.

Listing 1: My pseudocode for tsp.c:

```
Include necessary header files

Define function prototypes:
- Path *solve_tsp(Graph *graph)
- int next_permutation(uint32_t *array, uint32_t length)

Define main function(int argc, char *argv[])
    Initialize variables:
    - directed = 0
    - input_file = NULL
    - output_file = NULL

    Parse command line options using getopt():
    - '-d': Set directed to 1
    - '-i': Set input_file to optarg
    - '-o': Set output_file to optarg
    - '-h': Print help message and exit if encountered

    Open input file for reading if provided, otherwise use stdin
    Read number of vertices from input file
    Create a graph 'graph' with the specified number of vertices and directedness
    Read vertex names and add them to the graph
    Read number of edges from input file
    Read edge information and add edges to the graph

    Close the input file

    Solve the TSP problem using the solve_tsp() function, storing the result in tsp_path

    Print the TSP solution:
    - If output_file is provided, open it for writing, else use stdout
    - If no path is found, print "No path found! Alissa is lost!"
    - Otherwise, print the path and total distance

    Free memory allocated for tsp_path and graph

Define function solve_tsp(Graph *graph)
    Initialize variables:
    - num_vertices = graph_vertices(graph)
    - current_path = path_create(num_vertices + 1)
    - best_path = path_create(num_vertices + 1)

    Add start vertex to current_path
    Perform depth-first search (DFS) to find the best path starting from the start vertex
    Free memory allocated for current_path
    Return best_path

Define function dfs_tsp(current_vertex, graph, current_path, best_path)
    Mark current_vertex as visited in graph
    Check if all vertices are visited:
    - If yes and there is an edge back to start vertex:
        - Add start vertex to current_path
        - Update best_path if current_path is shorter
        - Remove start vertex from current_path
    - If not, recursively visit unvisited neighbors

    Mark current_vertex as unvisited in graph

Define function next_permutation(array, length)
    Implement algorithm to generate the next lexicographically greater permutation of the array
```

Function Descriptions

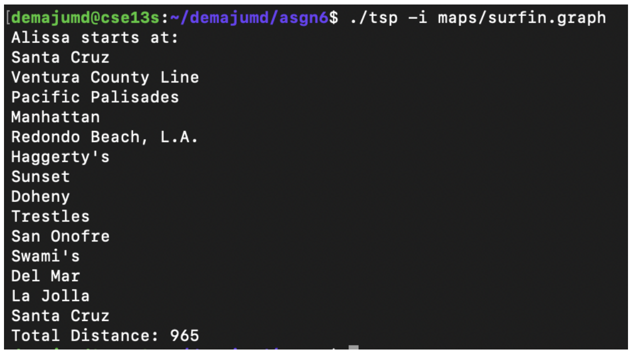
The function descriptions for the functions provided in the assignment by the Professor and tutors are mentioned in the Program Design section.

Results

Here are some of the results from running my program: In the first image of my program output, notice that a path is printed along with the total distance that it would take to travel the path. The path that is output is the most optimal path (in terms of distance) given the vertices/locations and edges/connections. Also, in every path that is printed by my program, every location is visited once (excluding the start/end location) and Alissa starts and ends her journey at the same place.

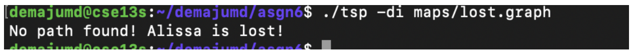
In the second image of my program output, notice that there is no path printed. Instead, my tsp.c program tells the user that Alissa is lost. This is because there was no path found that visits every location once and begins and ends at the starting location.

```
scan-build make
```



```
demajumd@cse13s:~/demajumd/asgn6$ ./tsp -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
```

Figure 1: Output of running tsp with surfin.graph when graph is undirected



```
demajumd@cse13s:~/demajumd/asgn6$ ./tsp -di maps/lost.graph
No path found! Alissa is lost!
```

Figure 2: Output of running tsp with lost.graph when graph is directed

When I run scan-build make on my command line, it results in "No bugs found." which is a good sign. That means that it did not find any bugs. However, there are possible bugs that could bypass this. `valgrind`

When I run valgrind on my command line with different combinations of the ./tsp, every valgrind output results in the same number of allocs and frees under HEAP SUMMARY as it says that: All heap blocks were freed – no leaks are possible and that: ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0). It was a successful run of valgrind.

References

[1] Wikipedia contributors. C (programming language) — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/C_(programming_language), 2023. [Online; accessed 20-April-2023].