Assignment 4 – Surfin' U.S.A

Aditya Bhaskar

CSE 13S – Spring 2023

Purpose

The purpose of this program is to assist Alissa, a college student, in planning a cost-effective journey to visit various cities mentioned in the Beach Boys' song "Surfin' U.S.A." Alissa aims to explore each city while starting and ending her trip in Santa Cruz, California, showcasing the region's exceptional oceans and beaches. Given her limited budget, the program utilizes graph theory, depth-first search, and a stack to solve the classic Traveling Salesman Problem and determine the most efficient route for Alissa.

This assignment on solving the Traveling Salesman Problem using graph theory and depth-first search provides several valuable programming lessons, including:

- **Problem-solving with algorithms** The assignment emphasizes the importance of using algorithmic thinking to break down complex problems into smaller, more manageable steps. It demonstrates how graph theory and search algorithms can be applied to real-world scenarios.
- **Graph representation:** The assignment introduces different ways to represent graphs, such as adjacency lists and adjacency matrices. Understanding these representations is essential for efficient graph traversal and manipulation.
- **Data Structures:** The assignment highlights the use of various data structures, including stacks, to manage and manipulate data during the program's execution. It emphasizes the importance of selecting the appropriate data structure based on the requirements of the problem.
- **TModular Programming:** The assignment promotes modular programming by dividing the code into separate files and structuring the program with different components (e.g., graph, stack, path). This modular approach enhances code organization, readability, and reusability.
- **Memory Management:** The assignment addresses memory management by dynamically allocating memory for data structures such as the stack and path. It emphasizes the importance of memory efficiency and avoiding wasteful memory allocation.
- **Graph Traversal Algorithms:** The assignment introduces depth-first search (DFS) as a graph traversal algorithm for exploring potential paths. It demonstrates how DFS can be used to solve the Traveling Salesman Problem and find the most efficient route.
- **Complexity Analysis:** The assignment prompts consideration of algorithmic complexity and efficiency. It highlights that the brute-force approach for solving the Traveling Salesman Problem becomes impractical for larger graphs, necessitating the use of optimized algorithms and heuristics.
- **Problem constraints and validation:** The assignment emphasizes the importance of validating the obtained solution against specific problem constraints. It demonstrates how to ensure that the path satisfies the requirements, such as visiting all cities exactly once and returning to the starting point.
- **Code Documentation:** The assignment exemplifies the significance of clear and concise code documentation. It provides explanations and comments throughout the code to enhance readability and facilitate understanding for both developers and potential users.

- **Software Engineering Practices:** By working on a structured assignment, students learn and practice essential software engineering principles such as code organization, encapsulation, code reuse, and adherence to coding standards.

Overall, this assignment provides valuable insights into algorithmic problem-solving, graph theory, data structures, memory management, and software engineering practices, equipping students with essential programming skills applicable to a wide range of real-world scenarios.

How to Use the Program

- **Compilation:**
  - Ensure that you have a C compiler installed on your system.
  - Open your ssh terminal and login to the VM.
  - Navigate to the directory (in this case the VM) where the program files and the Makefile are located.
  - Run the following command to compile the program:

```
make all
```

  - The Makefile will handle the compilation process and generate executable files for all the compiled files.
- **Running the Program:**
  - Once the program is successfully compiled, you can run it with the following command:

```
./tsp -d -i surfin.graph
```

  - You can specify additional command-line options to modify the program's behavior as needed.
- **Command-Line Options:**
  - **-i 'filename':** Sets the input file to read the graph from. Replace ¡filename¿ with the name of the input file. If this option is not provided, the program will read the graph from stdin.
  - **-o 'filename':** Sets the output file to write the results to. Replace ¡filename¿ with the desired name of the output file. If this option is not provided, the program will write the results to stdout.
  - **-d**: Treats all graphs as directed. By default, the program assumes an undirected graph. Specifying this option will consider the edges as directed, meaning only (i,j) will be added and not (j,i).
  - **-h**: Prints a help message explaining the command-line options and their usage.
- **I/O:**
  - The program expects the graph data to be provided in the input file (or stdin if not specified).
  - The graph should be formatted as follows:
    - The first line should contain the number of cities (vertices) in the graph.
    - Each subsequent line should specify an edge in the format: **'source vertex', 'destination vertex', and 'weight'**.
  - The program will calculate the shortest path that visits each city exactly once and returns to the starting city.
  - The output will be written to the specified output file (or stdout if not specified).
  - The output will include the shortest path and the total distance traveled.

- The shortest path will be displayed as a sequence of city names, indicating the order in which they should be visited.
- The total distance traveled represents the sum of the weights (travel time) of all edges in the path.

- **Removing Compiled Files:**
  - To remove the compiled files and clean up the directory, run the following command:

```
make clean
```

  - This command will remove the executable files and any object files generated during the compilation process.

Program Design

As the maintainer of this program, it's essential to understand its organization and structure to facilitate troubleshooting and potential enhancements. This section will provide an overview of the main data structures and algorithms employed in the program, using simple language to ensure accessibility to individuals without a computer science background.

- **File Organization:**
  - The program is organized into multiple files, each serving a specific purpose.
  - The main files include graph.c, stack.c, and tsp.c.
  - graph.c contains functions and structures related to graph manipulation.
  - stack.c implements the stack data structure and associated operations.
  - tsp.c contains the main logic for solving the TSP and managing user input and output.
- **Modular Approach:**
  - The program follows a modular design, dividing the functionality into separate files and functions.
  - This approach enhances code organization, readability, and maintainability.
  - Each file focuses on specific data structures or algorithms, making it easier to locate and modify relevant code sections.

Understanding the program's data structures, algorithms, and code organization will help you navigate the codebase effectively. By familiarizing yourself with the graph structure, stack implementation, DFS algorithm, and the TSP solution approach, you can troubleshoot issues, implement enhancements, and ensure the program continues to meet the desired objectives.

Data Structures

- **Graph Structure:**
  - The program utilizes a graph data structure to represent the cities and their connections.
  - Each city is treated as a vertex, and the connections between cities are represented as edges.
  - The graph structure consists of two primary components: vertices (cities) and edges (connections).
  - The vertices are stored as an array of strings, where each string represents the name of a city.
  - The edges are represented using an adjacency matrix, which is a two-dimensional array.
  - The adjacency matrix captures the weights (travel times) between cities.
- **Stack Structure:**
  - A stack is employed to track the path of Alissa's journey through the cities.

- The stack data structure is implemented using an array to store the elements (cities).
- The stack follows the Last-In-First-Out (LIFO) principle, where the last city visited is the first to be removed.

- **Path Structure:**
  - The path structure is used to track the sequence of cities visited and the total distance traveled.
  - It contains two main components: an array to store the city sequence and a variable to hold the total distance.

Algorithms

- **Depth-First Search:**
  - DFS is a graph traversal algorithm employed to search for paths in the graph.
  - It is utilized to explore all possible paths that Alissa can take during her journey.
  - The algorithm operates by starting at a city (vertex) and recursively exploring all adjacent cities until all cities are visited.
  - The stack data structure is instrumental in implementing DFS to keep track of the visited cities.

- **Traveling Salesman Problem:**
  - The program aims to solve the TSP, which involves finding the shortest path that visits every city once and returns to the starting city.
  - To solve the TSP efficiently, the program utilizes a combination of graph theory and DFS.
  - It employs a backtracking approach that explores all possible paths and keeps track of the shortest path found so far.
  - The algorithm ensures that Alissa visits each city exactly once, checks if the path is feasible, and identifies the most fuel-efficient path.

```
// Pseudocode for Graph Data Structure

struct Graph:
    vertices: uint32_t
    directed: bool
    visited: bool[]
    names: char*[]
    weights: uint32_t**


function graph_create(vertices, directed):
    g = allocate memory for Graph
    g.vertices = vertices
    g.directed = directed
    g.visited = allocate memory for bool array of size vertices, initialized with false
    g.names = allocate memory for char* array of size vertices
    g.weights = allocate memory for uint32_t* array of size vertices

    for i from 0 to vertices:
        g.names[i] = NULL
        g.weights[i] = allocate memory for uint32_t array of size vertices, initialized with 0

    return g
```

```
procedure graph_free(gp):
    g = *gp
    for i from 0 to g.vertices:
        if g.names[i] is not NULL:
            free g.names[i]
        free g.weights[i]

    free g.names
    free g.weights
    free g.visited
    free g
    *gp = NULL


function graph_vertices(g):
    return g.vertices


procedure graph_add_vertex(g, name, v):
    if g.names[v] is not NULL:
        free g.names[v]
    g.names[v] = duplicate string name


function graph_get_vertex_name(g, v):
    return g.names[v]


function graph_get_names(g):
    return g.names


procedure graph_add_edge(g, start, end, weight):
    g.weights[start][end] = weight
    if not g.directed:
        g.weights[end][start] = weight


function graph_get_weight(g, start, end):
    return g.weights[start][end]


procedure graph_visit_vertex(g, v):
    g.visited[v] = true


procedure graph_unvisit_vertex(g, v):
    g.visited[v] = false


function graph_visited(g, v):
    return g.visited[v]
```

```
procedure graph_print(g):
    for i from 0 to g.vertices:
        print("Vertex", i, ":", g.names[i])

    print("Adjacency Matrix:")
    for i from 0 to g.vertices:
        for j from 0 to g.vertices:
            print(g.weights[i][j], end=" ")
        print()
```

```
// pseudocode for Stack Data Structure

# Create a new stack with the given capacity
def stack_create(capacity):
    stack = {
        'capacity': capacity,
        'top': 0,
        'items': [None] * capacity
    }
    return stack

# Free the memory allocated for the stack
def stack_free(stack):
    stack['items'] = None
    stack = None

# Push a value onto the stack
def stack_push(stack, val):
    if stack['top'] == stack['capacity']:
        return False
    stack['items'][stack['top']] = val
    stack['top'] += 1
    return True

# Pop the top value from the stack
def stack_pop(stack):
    if stack['top'] == 0:
        return False, None
    stack['top'] -= 1
    val = stack['items'][stack['top']]
    return True, val

# Peek the top value from the stack
def stack_peek(stack):
    if stack['top'] == 0:
        return False, None
    val = stack['items'][stack['top'] - 1]
    return True, val

# Check if the stack is empty
def stack_empty(stack):
```

```
    return stack['top'] == 0

# Check if the stack is full
def stack_full(stack):
    return stack['top'] == stack['capacity']

# Get the size of the stack
def stack_size(stack):
    return stack['top']

# Copy the contents of the source stack to the destination stack
def stack_copy(dst, src):
    if dst['capacity'] >= src['top']:
        dst['top'] = src['top']
        for i in range(src['top']):
            dst['items'][i] = src['items'][i]

# Print the elements of the stack
def stack_print(stack, file, vals):
    for i in range(stack['top']):
        file.write(vals[stack['items'][i]] + '\n')
```

```
# Pseudocode for Path Data Structure

# Create a new path
def path_create():
    path = {
        'length': 0,
        'vertices': []
    }
    return path

# Free the memory allocated for the path
def path_free(path):
    path['vertices'] = None
    path = None

# Add a vertex to the end of the path
def path_add_vertex(path, vertex):
    path['vertices'].append(vertex)
    path['length'] += 1

# Remove the last vertex from the path
def path_remove_vertex(path):
    if path['length'] > 0:
        path['vertices'].pop()
        path['length'] -= 1

# Get the length of the path
def path_length(path):
    return path['length']

# Get the vertex at a specific index in the path
```

```
def path_get_vertex(path, index):
    return path['vertices'][index]

# Check if the path contains a specific vertex
def path_contains_vertex(path, vertex):
    return vertex in path['vertices']

# Check if the path is empty
def path_empty(path):
    return path['length'] == 0

# Print the vertices in the path
def path_print(path, file, vertex_names):
    for vertex in path['vertices']:
        file.write(vertex_names[vertex] + '\n')
```

```
# Pseudocode for Depth-First Search (DFS)

# Recursive DFS function
def dfs(graph, start_vertex, visited):
    # Mark the current vertex as visited
    visited[start_vertex] = True

    # Process the current vertex
    process_vertex(start_vertex)

    # Traverse all adjacent vertices
    for neighbor in graph.get_neighbors(start_vertex):
        if not visited[neighbor]:
            dfs(graph, neighbor, visited)

# Perform DFS traversal on the graph
def dfs_traversal(graph, start_vertex):
    # Create a visited array to track visited vertices
    visited = [False] * graph.num_vertices()

    # Call the recursive DFS function
    dfs(graph, start_vertex, visited)

# Process the vertex (custom operation)
def process_vertex(vertex):
    print("Processing vertex:", vertex)
    # Perform any custom operations on the vertex
```

Function Descriptions

- **graph_create(vertices, directed):**
  - Inputs: vertices (number of vertices in the graph), directed (a boolean indicating whether the graph is directed or not)
  - Output: Returns a pointer to the newly created graph object
  - Purpose: Creates a new graph structure and initializes its properties, such as the number of vertices, directedness, visited array, names array, and weights matrix.

- **graph_free(gp):**
  - Input: gp (a double pointer to the graph object)
  - Output: None
  - Purpose: Frees the memory occupied by the graph object and all its associated arrays and matrices. Sets the pointer to NULL to avoid any potential dangling references.
- **graph_vertices(g):**
  - Input: g (the graph object)
  - Output: Number of vertices in the graph
  - Purpose: Retrieves the number of vertices present in the graph.
- **graph_add_vertex(g, name, v):**
  - Inputs: g (the graph object), name (the name of the city to be associated with the vertex), v (the vertex number)
  - Output: None
  - Purpose: Assigns the given name to the city associated with the specified vertex in the graph. Performs memory management to store a copy of the name in the graph.
- **graph_get_vertex_name(g, v):**
  - Inputs: g (the graph object), v (the vertex number)
  - Output: The name of the city associated with the specified vertex in the graph
  - Purpose: Retrieves the name of the city corresponding to the given vertex in the graph.
- **graph_get_names(g):**
  - Input: g (the graph object)
  - Output: An array of city names stored in the graph
  - Purpose: Retrieves the array of city names present in the graph. Note that it returns the actual array, not a copy.
- **graph_add_edge(g, start, end, weight):**
  - Inputs: g (the graph object), start (the starting vertex of the edge), end (the ending vertex of the edge), weight (the weight/cost of the edge)
  - Output: None
  - Purpose: Adds an edge between the specified vertices in the graph. If the graph is undirected, the edge is added in both directions.
- **graph_get_weight(g, start, end):**
  - Inputs: g (the graph object), start (the starting vertex of the edge), end (the ending vertex of the edge)
  - Output: The weight of the edge between the specified vertices in the graph
  - Purpose: Retrieves the weight of the edge connecting the given vertices in the graph.
- **graph_visit_vertex(g, v):**
  - Inputs: g (the graph object), v (the vertex to mark as visited)
  - Output: None
  - Purpose: Marks the specified vertex as visited in the graph.

- **graph_unvisit_vertex(g, v):**
  - Inputs: g (the graph object), v (the vertex to mark as unvisited)
  - Output: None
  - Purpose: Marks the specified vertex as unvisited in the graph.
- **graph_visited(g, v):**
  - Inputs: g (the graph object), v (the vertex to mark as visited)
  - Output: returns true or false based on the fact that the vertex has been visited or not.
  - Purpose: returns if vertex has been visited or not
- **graph_print(g):**
  - Inputs: g (the graph object)
  - Output: None
  - Purpose: Prints the graph
- **path_create(capacity):**
  - Inputs: capacity (the capacity of the path's stack)
  - Output: Path* (pointer to the created Path struct)
  - Purpose: Creates a new Path object and initializes its properties. It allocates memory for the struct and creates an empty stack for storing vertices.
- **path_free(path **pp):**
  - Inputs: pp (double pointer to Path)
  - Output: None
  - Purpose: Frees the memory allocated for the Path object and its associated stack. It sets the double pointer pp to NULL to prevent dangling references.
- **path_vertices(path *p):**
  - Inputs: p (pointer to Path)
  - Output: **uint32_t** (number of vertices in the path)
  - Purpose: Returns the number of vertices in the path. It simply calls **stack_size** on the path's stack and returns the result.
- **path_distance(path *p):**
  - Inputs: p (pointer to Path)
  - Output: **uint32_t** (total weight/distance of the path)
  - Purpose: Returns the total weight/distance of the path. It retrieves the total weight from the Path struct and returns it.
- **path_add(path *p, val, *g):**
  - Inputs: p (pointer to Path), val (vertex value to add), g (pointer to Graph)
  - Output: None
  - Purpose: Adds a vertex to the path. It calculates the weight of the edge between the last vertex in the path and the new vertex, updates the total weight of the path, and pushes the new vertex onto the stack.
- **path_remove(path *p, *g):**

- Inputs: p (pointer to Path), g (pointer to Graph)
- Output: **uint32_t** (value of the removed vertex)
- Purpose: Removes the last vertex from the path and returns its value. It calculates the weight of the edge between the new last vertex and the removed vertex, updates the total weight of the path, and pops the last vertex from the stack.

- **path_copy(*dst, *src):**
  - Inputs: dst (pointer to the destination Path), src (pointer to the source Path)
  - Output: None
  - Purpose: Copies the contents of the source path to the destination path. It copies the total weight and calls **stack_copy** to copy the vertices from the source stack to the destination stack.

- **path_print(*p, *outfile, *g):**
  - Inputs: p (pointer to Path), outfile (pointer to the output file), g (pointer to Graph)
  - Output: None
  - Purpose: Prints the vertices in the path, along with the total distance, to the specified output file. It iterates through the stack of vertices, retrieves the vertex names from the graph, and prints them to the file.

- **stack_create(uint32_t capacity):**
  - Inputs: capacity - the maximum capacity of the stack.
  - Output: Returns a pointer to the ne.wly created stack.
  - Purpose: Creates a stack data structure with the specified capacity. It allocates memory for the stack and initializes its fields.

- **stack_push(Stack *s, uint32_t val):**
  - Inputs: s - a pointer to the stack, val - the value to be pushed onto the stack.
  - Output: Returns true if the push operation is successful, false if the stack is already full.
  - Pushes a value onto the stack.

- **stack_free(Stack **sp):**
  - Inputs: sp - a double pointer to the stack pointer that needs to be freed.
  - Output: None
  - Purpose: Frees the memory allocated for the stack and its items.

- **path_distance(path *p):**
  - Inputs: p (pointer to Path)
  - Output: **uint32_t** (total weight/distance of the path)
  - Purpose: Returns the total weight/distance of the path. It retrieves the total weight from the Path struct and returns it.

- **stack_pop(Stack *s, uint32_t *val):**
  - Inputs: s - a pointer to the stack, val - a pointer to a variable to store the popped value.
  - Output: Returns true if the pop operation is successful, false if the stack is already empty.
  - Purpose: Pops the top value from the stack and stores it in the provided variable.

- **stack_peek(const Stack *s, uint32_t *val):**

  - Inputs: s - a pointer to the stack, val - a pointer to a variable to store the peeked value.
  - Output: Returns true if the peek operation is successful, false if the stack is empty.
  - Purpose: Retrieves the top value from the stack without removing it and stores it in the provided variable.

- **stack_empty(const Stack *s)**
  - Inputs: s - a pointer to the stack.
  - Output: Returns true if the stack is empty, false otherwise.
  - Purpose: Checks if the stack is empty.

- **stack_full(const Stack *s):**
  - Inputs: s - a pointer to the stack.
  - Output: Returns true if the stack is full, false otherwise.
  - Purpose: Checks if the stack is full.

- **stack_size(const Stack *s):**
  - Inputs: s - a pointer to the stack.
  - Output: Returns the size of the stack.
  - Purpose: Returns the number of elements currently stored in the stack.

- **stack_copy(Stack *dst, const Stack *src):**
  - dst - a pointer to the destination stack, src - a pointer to the source stack.
  - Output: None
  - Purpose: Copies the contents of the source stack to the destination stack.

- **stack_print(const Stack *s, FILE *f, char *vals[]):**
  - s - a pointer to the stack, f - a pointer to the file where the stack elements will be printed, vals[] - an array of strings representing the values stored in the stack.
  - Output: None
  - Purpose: Prints the elements of the stack to a file, using the corresponding string values from vals[].

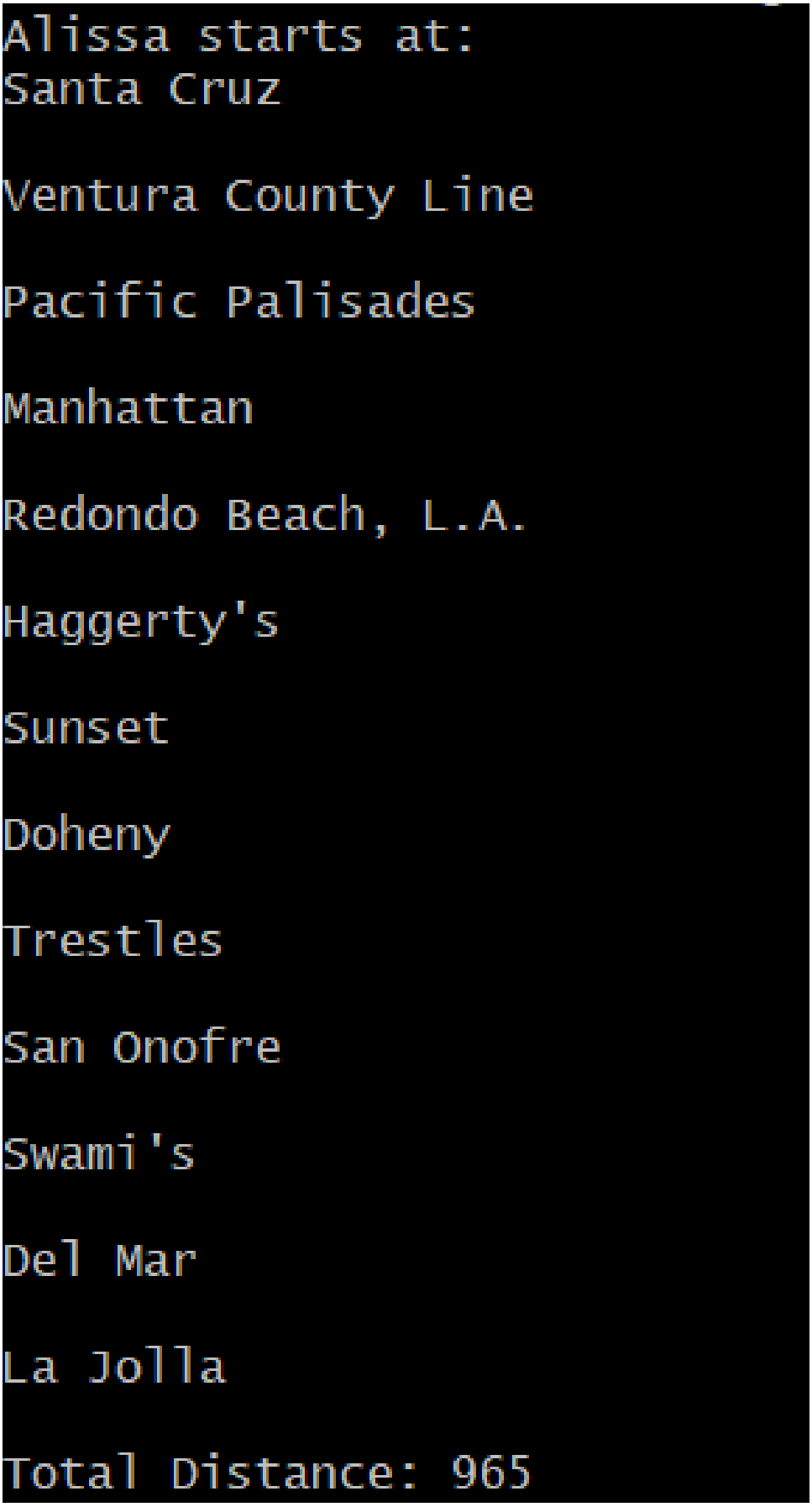

Figure 1: Screenshot of the program running.