Assignment 6 – Surfin' USA Report Template

Nickolas Tran

CSE 13S – Winter 24

Purpose

This assignment focuses on solving the Traveling Salesman Problem (TSP). The TSP involves finding the shortest possible route that visits each city exactly once and returns to the origin city. You will be using graph representation, graph search algorithms, path and cycle handling, optimization and file input to create a project to find out the most efficient way to visit each city using the smallest amount of gas.

Questions

- What benefits do adjacency lists have? What about adjacency matrices?
  Adjacency lists is an efficient use of memory, especially for graphs. Allows for quick iteration over neighbors of a vertex. Suitable for graphs with a large number of vertices but relatively few edges. Adjacency matrices allows for constant time lookup of edge existence. Suitable for dense graphs where most pairs of vertices are connected.

- Which one will you use. Why did we choose that (hint: you use both)
  I use adjacency lists for efficient memory usage and traversal operations, while adjacency matrices are employed for fast edge existence checks.

- If we found a valid path, do we have to keep looking? Why or why not?
  If I find a valid path, I might not need to keep looking. For some problems, finding any valid path might be sufficient, while for others we may need to find the shortest path or all possible paths.

- If we find 2 paths with the same weights, which one do we choose?
  If multiple paths with the same weights are found, the choice of which one to use might depend on the specific requirements of the problem.

- Is the path that is chosen deterministic? Why or why not?
  The path chosen might be deterministic depending on the algorithm and its implementation. If the algorithm always makes the same choices based on the input and its internal state, the chosen path would be deterministic.

- What type of graph does this assignment use? Describe it as best as you can.
  The graph used in this assignment is an undirected, weighted graph, where the objective is to find the optimal Hamiltonian cycle.

- 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?
  Constraints on edge weights could include limits on their range. I could optimize the DFS further by considering these constraints to reduce unnecessary branches of the search tree, leading to improved performance.

Testing

Unit Testing: I will write unit tests for each function in the path.c file to ensure that they behave as expected under different scenarios. This includes testing boundary cases, normal cases, and edge cases.
Integration Testing: I will integrate the path.c module with the existing graph.c and stack.c modules and conduct integration tests to verify that they work together seamlessly.
Input Validation Testing: I will test the functions with various types of input, including valid input, invalid input, and edge cases, to ensure robustness and proper error handling.
Error Handling Testing: I will test errors in the input data and see how the functions handle these errors.

How to Use the Program

The user uses the program to compile and run the code

```
make
```

The user then runs the main program

```
./tsp -d -i maps/<graph file name>
```

Program Design

0.1 Main Data Structures:

1. Graph Structure: The program utilizes a graph data structure to represent relationships between vertices. This structure consists of an adjacency list to efficiently store connections between vertices. Each vertex contains information about its adjacent vertices and edge weights.
2. Path Structure: To manage paths within the graph, the program creates a path data structure. This structure maintains a sequence of vertices representing a path along with its associated distance. It provides functions to add and remove vertices from the path, calculate distances, and perform operations related to path manipulation.
3. Stack Structure: A stack data structure is utilized for various operations within the program, such as maintaining a sequence of vertices during path traversal and exploration. The stack supports standard operations like push, pop, peek, and provides functions to check for emptiness, fullness, and retrieve its size.

0.2 Main Algorithms:

1. Graph Traversal Algorithms: The program implements graph traversal algorithms such as Depth-First Search (DFS) and Breadth-First Search (BFS) to explore the vertices and edges of the graph efficiently. These algorithms are used for tasks like pathfinding, cycle detection, and connected component identification.
2. Pathfinding Algorithm: To find paths between vertices, the program employs algorithms like Dijkstra's algorithm or A* search algorithm. These algorithms determine the shortest path or the path with the lowest cost between two vertices in the graph.
3. Path Manipulation Algorithms: Various algorithms are implemented to manipulate paths within the graph, including adding vertices to a path, removing vertices, calculating distances, and copying paths from one data structure to another.
4. Stack-Based Operations: The stack data structure is used for maintaining the state of traversal and exploration algorithms. Stack-based operations are used for backtracking, storing intermediate results, and managing recursive function calls efficiently.

Pseudocode

for graph.c

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>
#include <inttypes.h>

#include "graph.h"

Graph Structure:
    uint32_t   vertices
    bool       directed
    bool       *visited
    char       **names
    uint32_t   **weights

{insert following functions in function descriptions}
```

for stack.c

```
#include <inttypes.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>

#include "stack.h"

Stack Structure:
    items: Array of uint32_t to store the stack elements
    capacity: Maximum capacity of the stack
    top: Index of the top element in the stack, initialized to -1

{insert following functions in function descriptions}
```

for path.c

```
#include "path.h"
#include "stack.h"
#include "graph.h"

#include <stdio.h>
#include <stdbool.h>
#include <inttypes.h>
#include <stdlib.h>

Structure Path:
    uint32_t      total_weight
        Stack         *vertices


{insert following functions in function descriptions}
```

for tsp.c

```
#include "graph.h"
#include "path.h"
#include "stack.h"
#include "vertices.h"

#include <inttypes.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

function DFS(graph, currentVertex, endVertex, path, shortestPath):
    graph.visitVertex(currentVertex)
    path.addVertex(currentVertex)

    if currentVertex == endVertex:x
        if graph.getWeight(currentVertex, 0) != 0:
            // Add the starting vertex to the path
            path.addVertex(0)

            if shortestPath.distance == 0 OR path.distance < shortestPath.distance:
                shortestPath.copy(path)

        path.removeLastVertex()

    for each adjacentVertex of currentVertex:
        if graph.getWeight(currentVertex, adjacentVertex) != 0 AND !graph.isVisited(adjacentVertex):
            DFS(graph, adjacentVertex, endVertex, path, shortestPath)

    graph.unvisitVertex(currentVertex)
    path.removeLastVertex()

function main():
    DFS(graph, 0, numberOfVertices, path, result)
    if result.distance != 0:
    else:
```

Function Descriptions

For function Graph *graph_create(uint32_t, bool directed):

- Input: "vertices" and "directed"
- Output: Returns a new graph object
- Purpose: Creates a new graph with the specific number of vertices and directness
- Pseudocode:

```
function graph_create(vertices, directed):
    create a new graph object g
    set g.vertices to vertices
    set g.directed to directed
    allocate memory for g.visited array of size vertices
    allocate memory for g.names array of size vertices
    allocate memory for g.weights 2D array of size vertices x vertices
    return g
```

For function void graph_free(Graph **gp):

- Input: "gp"
- Output: None
- Purpose: Frees the memory allocated for the graph object
- Pseudocode:

```
function graph_free(gp):
    if gp is NULL or *gp is NULL:
        return
    free memory for visited array in *gp
    free memory for names array in *gp
    free memory for weights 2D array in *gp
    free memory for *gp
    set *gp to NULL
```

For function uint32_t graph_vertices(const Graph *g):

- Input: "g"
- Output: Returns the number of vertices in the graph
- Purpose: Finds the number of vertices in a graph
- Pseudocode:

```
function graph_vertices(g):
    if g is NULL:
        return 0
    return g.vertices
```

For function void graph_add_vertex(Graph *g, const char *name, uint_32_t v):

- Input: "g" and "name" and "v"
- Output: None

- Purpose: Creates a copy and stores the name in the graph object
- Pseudocode:

```
function graph_add_vertex(g, name, v):
    if g is NULL or v >= g.vertices or name is NULL:
        return
    set g.names[v] to a copy of name
```

For function const char* graph_get_vertex_name(const Graph *g, uint32_t v):

- Input: "g" and "v"
- Output: Returns the name of the specified vertex
- Purpose: Gets the name of the city of the array and returns the stored in the Graph.
- Pseudocode:

```
function graph_get_vertex_name(g, v):
    if g is NULL or v >= g.vertices:
        return NULL
    return g.names[v]
```

For function char **graph_get_names(const Graph *g):

- Input: "g"
- Output: Returns an array of vertex names
- Purpose: Gets the names of the cities in the array
- Pseudocode:

```
function graph_get_names(g):
    if g is NULL:
        return NULL
    return g.names
```

For function void graph_add_edge(Graph *g, uint32_t start, uint32_t end, uint32_t weight):

- Input: "g" and "start" and "end" and "weight"
- Output: None
- Purpose: Adds and edge with weight to the adjacency matrix
- Pseudocode:

```
function graph_add_edge(g, start, end, weight):
    if g is NULL or start >= g.vertices or end >= g.vertices:
        return
    set weight in g.weights[start][end] to weight
    if g is not directed:
        set weight in g.weights[end][start] to weight
```

For function uint32_t graph_get_weight(const Graph *g, uint32_t start, uint32_t end):

- Input: "g" and "start" and "end"
- Output: Returns the weight of the edge between the specified vertices

- Purpose: Loooks up the weight of the edge
- Pseudocode:

```
function graph_get_weight(g, start, end):
    if g is NULL or start >= g.vertices or end >= g.vertices:
        return 0
    return weight in g.weights[start][end]
```

For function void graph_visit_vertex(Graph *g, uint32_t v):

- Input: "g" and "v"
- Output: None
- Purpose: Adds the vertex to the list of vertices
- Pseudocode:

```
function graph_visit_vertex(g, v):
    if g is NULL or v >= g.vertices:
        return
    set g.visited[v] to true
```

For function void graph_unvisit_vertex(Graph *g, uint32_t v):

- Input: "g" and "v"
- Output: None
- Purpose: Removes the vertex from the list of visited vertices
- Pseudocode:

```
function graph_unvisit_vertex(g, v):
    if g is NULL or v >= g.vertices:
        return
    set g.visited[v] to false
```

For function bool graph_visited(const Graph *g, uint32_t v):

- Input: "g" and "v"
- Output: Returns a boolean value indicating whether the vertex is visited or not.
- Purpose: Returns true if v is visited and g if false
- Pseudocode:

```
function graph_visited(g, v):
    if g is NULL or v >= g.vertices:
        return false
    return g.visited[v]
```

For function void graph_print(const Graph *g):

- Input: "g"
- Output: None
- Purpose: Prints a representation of the map

- Pseudocode:

```
function graph_print(g):
    if g is NULL:
        return
    print "Graph with " + g.vertices + " vertices:"
    print "Directed: " + g.directed
    print "Vertices:"
    for each vertex v in g:
        print v + ": " + g.names[v]
    print "Edges:"
    for each edge (u, v) in g:
        if g.weights[u][v] is not 0:
            print g.names[u] + " -> " + g.names[v] + ": " + g.weights[u][v]
```

For function Stack *stack_create(uint32_t capacity):

- Input: Capacity of the stack
- Output: Pointer to a new created stack
- Purpose: Creates a stack and allocates space for it
- Pseudocode:

```
Function stack_create(capacity):
    Initialize a new stack
    Allocate memory for the stack structure
    Allocate memory for the items array with the given capacity
    Set the capacity of the stack
    Set top to -1
    Return the stack
```

For function void stack_free(Stack **sp):

- Input: Pointer to a pointer to the stack
- Output: None
- Purpose: Frees all the space used by the stack
- Pseudocode:

```
Function stack_free(stack):
    Deallocate memory for the items array
    Deallocate memory for the stack structure
    Set stack to NULL
```

For function bool stack_push(Stack *s, uint32_t val):

- Input: Pointer to the stack, value to push onto the stack
- Output: True if push is successful, false otherwise
- Purpose: Adds val to the top of the stack
- Pseudocode:

```
Function stack_push(stack, value):
    If stack is NULL or stack is full:
        Return false
    Increment top
    Set items[top] to value
    Return true
```

For function bool stack_pop(Stack *s, uint32_t *val):

- Input: Pointer to the stack, pointer to store the popped value
- Output: True if pop is successful, false otherwise
- Purpose: Sets the integer pointed by val to the last item and removes the last item on the stack
- Pseudocode:

```
Function stack_pop(stack, value):
    If stack is NULL or stack is empty:
        Return false
    Set value to items[top]
    Decrement top
    Return true
```

For function bool stack_peek(const Stack *s, uint32_t *val);

- Input: Pointer to constant stack, pointer to store peeked value
- Output: True if peek is successful, false otherwise
- Purpose: Sets the integer to the last item on the stack but it does not change the stack
- Pseudocode:

```
Function stack_peek(stack, value):
    If stack is NULL or stack is empty:
        Return false
    Set value to items[top]
    Return true
```

For function bool stack_empty(const Stack *s):

- Input: Pointer to constant stack
- Output: True if stack is empty, false otherwise
- Purpose: Returns true if stack is empty
- Pseudocode:

```
Function stack_empty(stack):
    Return whether top is -1
```

For function bool stack_full(const Stack *s):

- Input: Pointer to constant stack
- Output: True if stack is full, false otherwise
- Purpose: Returns true if stack is full

- Pseudocode:

```
Function stack_full(stack):
    Return whether top is equal to capacity - 1
```

For function uint32_t stack_size(const Stack *s):

- Input: Pointer to constant stack
- Output: The number of elements in the stack
- Purpose: Returns the numbers in the stack
- Pseudocode:

```
Function stack_size(stack):
    Return top + 1
```

For function void stack_copy(Stack *dst, const stack *src):

- Input: Pointers to the destination and source stack
- Output: None
- Purpose: Overwrites `dst` with all the items from `src`
- Pseudocode:

```
Function stack_copy(destination, source):
    If destination or source is NULL:
        Return
    Copy capacity and top from source to destination
    For each index i from 0 to top:
        Copy items[i] from source to destination
```

For function void stack_print(const Stack *s, FILE *outfile, char *cities[]):

- Input: Pointer to a constant stack , file pointer for output, array of value labels
- Output: None
- Purpose: Print out the stack as a list of elements
- Pseudocode:

```
Function stack_print(stack, file, values):
    If stack, file, or values is NULL:
        Return
    Write "Stack contents:" to the file
    For each index i from top to 0:
        Write "[i] values[items[i]]" to the file
```

For function Path *path_create(uint32_t capacity):

- Input: "capacity"
- Output: Returns a pointer to the newly created "Path" structure, or "NULL" if memory allocation fails
- Purpose: Creates a path containing a Stack

- Pseudocode:

```
Function path_create(capacity):
    Create a new Path structure
    Initialize vertices array with size capacity
    Set size to 0
    Set distance to 0
    Return the new Path
```

For function void path_free(Path **pp):

- Input: "pp"
- Output: Frees the memory allocated for the "Path" structure and sets the pointer to "NULL"
- Purpose: Frees a path and its associated memory
- Pseudocode:

```
Function path_free(pp):
    Free memory allocated for vertices array in Path
    Free memory allocated for Path structure
    Set pp to NULL
```

For function uint32_t path_vertices(const Path *p):

- Input: "p"
- Output: Returns the number of vertices in the path
- Purpose: Finds the number of vertices in the path
- Pseudocode:

```
Function path_vertices(p):
    Return p.size
```

For function uint32_t path_distance(const Path *p):

- Input: "p"
- Output: Returns the total distance of the path, which is the sum of weights of edges between consecutive vertices
- Purpose: Finds the distance covered by the path
- Pseudocode:

```
Function path_distance(p):
    Return p.distance
```

For function void path_add(Path *p, uint32_t val, const Graph *g):

- Input: "p" and "val" and "g"
- Output: Adds the vertex with value "val" to the path "p"
- Purpose: Adds vertex `val` from graph `g` to the path
- Pseudocode:

```
Function path_add(p, val, g):
    Add val to the end of vertices array in Path
    Increase size by 1
    If size > 1:
        Get the weight of the edge from the last vertex to val in graph g
        Add the weight to the total distance of the path
```

For function uint32_t path_remove(Path *p, const Graph *g):

- Input: "p" and "g"
- Output: Removes the last vertex from the path "p" and returns its value
- Purpose: Removes the most recent added vertex from the path
- Pseudocode:

```
Function path_remove(p, g):
    If size == 0:
        Return 0
    Remove the last vertex from vertices array in Path
    Decrease size by 1
    If size > 0:
        Get the weight of the edge from the new last vertex to the vertex that was removed in
            graph g
        Subtract the weight from the total distance of the path
    Return the removed vertex
```

For function void path_copy(Path *dst, const Path *src):

- Input: "dst" and "src"
- Output: Copies the contents of the source path "src" to the destination path "dst"
- Purpose: Copies a path from `src` to `dst`
- Pseudocode:

```
Function path_copy(dst, src):
    Copy the vertices array from src to dst
    Copy size and distance from src to dst
```

For function void path_print(const Path *p, FILE *outfile, const Graph *g):

- Input: "p" and "outfile" and "g"
- Output: Prints the vertices of the path "p" and the total distance of the path to the file stream "f"
- Purpose: Prints the path stored
- Pseudocode:

```
Function path_print(p, f, g):
    For each vertex v in vertices array of Path p:
        Print the name of vertex v using graph_get_vertex_name function from graph g
        If it's not the last vertex, print "->"
    Print the total distance of the path
```