Design Draft

Purpose

The purpose of this graph program is to provide a basic implementation of a graph data structure in C, allowing for the creation, manipulation, and traversal of graphs. It supports both directed and undirected graphs and includes features to add vertices and edges, retrieve vertex names, check visited status, and print a human-readable representation of the graph.

Questions

Benefits of Adjacency Lists:

- Adjacency lists are more memory-efficient when the graph is sparse because they only store information about edges that actually exist.
- They allow for efficient traversal of the graph, especially for sparse graphs, as only adjacent vertices are stored for each vertex.
- Adjacency lists are easier to work with when the graph is dynamic and changing frequently because adding or removing edges requires less memory manipulation.

Benefits of Adjacency Matrices:

- Adjacency matrices are more memory-efficient for dense graphs because they represent every possible edge, making lookups for edge existence faster.
- They allow for constant-time lookup of edge existence between any two vertices.
- Adjacency matrices are easier to work with for certain graph algorithms like Floyd-Warshall for all-pairs shortest path, as it involves matrix operations.

Choice of Representation:

- For this assignment, both adjacency lists and adjacency matrices are used.
- Adjacency lists are used to represent the graph for memory efficiency and flexibility in handling sparse graphs.
- Adjacency matrices are used to store edge weights for quick lookup during calculations like Dijkstra's algorithm.

Valid Path Search:

- If a valid path is found, there is no need to keep looking for another path if the goal is to find just one shortest path.
- However, if the goal is to find all possible shortest paths, then the search should continue until all paths are exhausted or a certain condition is met.

Choosing Between Paths with Same Weights:

- If two paths with the same weights are found, either one can be chosen arbitrarily, as they are equivalent in terms of their total weight.

Deterministic Path Choice:

- The path chosen is deterministic in the sense that given the same input and conditions, the algorithm will always choose the same path.
- However, if there are multiple shortest paths with the same weight, the choice between them may not be deterministic unless a specific tie-breaking rule is defined.

Type of Graph in the Assignment:

- The assignment uses an undirected weighted graph.
- It represents cities as vertices and the distances between them as edge weights.

Constraints on Edge Weights:

- Edge weights represent distances between cities, which are typically non-negative.
- For Alissa's case, the edge weights may represent either physical distances or travel times.
- To optimize DFS further based on constraints, we could consider:
    - Pruning branches of the search tree if the current path's weight exceeds a certain threshold, assuming we want to find the shortest path.
    - Avoiding revisiting previously visited vertices if it's not beneficial, using techniques like memoization or marking visited vertices.

Testing

To comprehensively test the graph program, we will:

- Create both directed and undirected graphs.
- Add vertices and edges with various names and weights.
- Retrieve and print vertex names.
- Check and update the visited status of vertices.
- Print the graph to verify the adjacency matrix and vertex information.
- Use tools like Valgrind to ensure there are no memory leaks.

How to Use the Program

To use this graph program:

Compilation:

gcc -o graph graph.c

Execution:

./graph

1. **Functions:**
    - graph_create(vertices, directed): Creates a new graph.
    - graph_free(&graph): Frees the graph.
    - graph_add_vertex(graph, name, vertex): Adds a vertex.
    - graph_get_vertex_name(graph, vertex): Gets the vertex name.
    - graph_add_edge(graph, start, end, weight): Adds an edge.
    - graph_get_weight(graph, start, end): Gets the edge weight.
    - graph_visit_vertex(graph, vertex): Marks a vertex as visited.
    - graph_unvisit_vertex(graph, vertex): Marks a vertex as unvisited.
    - graph_visited(graph, vertex): Checks if a vertex is visited.
    - graph_print(graph): Prints the graph.
2. **Stack Functions:**
    - stack_create(capacity): Creates a new stack.
    - stack_free(&stack): Frees the stack.
    - stack_push(stack, value): Pushes a value onto the stack.
    - stack_pop(stack, &value): Pops a value from the stack.
    - stack_peek(stack, &value): Peeks at the top value of the stack.
    - stack_empty(stack): Checks if the stack is empty.
    - stack_full(stack): Checks if the stack is full.
    - stack_size(stack): Returns the size of the stack.
    - stack_copy(dst, src): Copies one stack to another.
    - stack_print(stack, outfile, cities): Prints the stack.
3. **Path Functions:**
    - path_create(capacity): Creates a new path.
    - path_free(&path): Frees the path.
    - path_vertices(path): Returns the number of vertices in the path.
    - path_distance(path): Returns the total distance of the path.
    - path_add(path, value, graph): Adds a vertex to the path.
    - path_remove(path, graph): Removes a vertex from the path.
    - path_copy(dst, src): Copies one path to another.
    - path_print(path, outfile, graph): Prints the path.

Program Design

The program consists of the following data structures and functions:

- **Data Structures:**
    - Graph: A structure containing the number of vertices, directed/undirected flag, visited array, names array, and adjacency matrix.

- **Main Algorithms:**
    - Creation and initialization of the graph.
    - Memory management for graph data.
    - Addition and retrieval of vertices and edges.
    - Checking and updating visited status.
    - Printing the graph for visualization.

Pseudocode

```
function graph_create(vertices, directed)
    allocate memory for Graph
    initialize vertices and directed
    initialize visited array to false
    initialize names array
    initialize adjacency matrix with zeros
    return Graph

function graph_free(graph)
    if graph is NULL, return
    free visited array
    free each name in names array
    free names array
    free each row in adjacency matrix
    free adjacency matrix
    free graph
    set graph to NULL

function graph_add_vertex(graph, name, vertex)
    if name already exists at vertex, free it
    copy name to names array at vertex

function graph_get_vertex_name(graph, vertex)
    return name from names array at vertex

function graph_add_edge(graph, start, end, weight)
    set weight in adjacency matrix from start to end
    if graph is undirected, set weight from end to start

function graph_get_weight(graph, start, end)
    return weight from adjacency matrix

function graph_visit_vertex(graph, vertex)
    set visited array at vertex to true
```

```
function graph_unvisit_vertex(graph, vertex)
    set visited array at vertex to false

function graph_visited(graph, vertex)
    return visited status from visited array

function graph_print(graph)
    print graph type and number of vertices
    for each vertex
        print vertex name and visited status
        for each connected vertex
            print connected vertex and weight
```

Function Descriptions

1. **graph_create**
    - **Inputs:** uint32_t vertices, bool directed
    - **Outputs:** Graph*
    - **Purpose:** Initializes and returns a new graph structure.
    - **Pseudocode:** See above.
2. **graph_free**
    - **Inputs:** Graph **gp
    - **Outputs:** None
    - **Purpose:** Frees all memory used by the graph and sets the pointer to NULL.
    - **Pseudocode:** See above.
3. **graph_vertices**
    - **Inputs:** const Graph *g
    - **Outputs:** uint32_t
    - **Purpose:** Returns the number of vertices in the graph.
    - **Pseudocode:** Simply return the vertices count.
4. **graph_add_vertex**
    - **Inputs:** Graph *g, const char *name, uint32_t v
    - **Outputs:** None
    - **Purpose:** Adds a name to a vertex.
    - **Pseudocode:** See above.
5. **graph_get_vertex_name**
    - **Inputs:** const Graph *g, uint32_t v
    - **Outputs:** const char*
    - **Purpose:** Returns the name of a vertex.
    - **Pseudocode:** See above.
6. **graph_get_names**
    - **Inputs:** const Graph *g
    - **Outputs:** char**

    - **Purpose:** Returns an array of vertex names.
    - **Pseudocode:** Simply return the names array.
7. **graph_add_edge**
    - **Inputs:** Graph *g, uint32_t start, uint32_t end, uint32_t weight
    - **Outputs:** None
    - **Purpose:** Adds an edge with a specified weight.
    - **Pseudocode:** See above.
8. **graph_get_weight**
    - **Inputs:** const Graph *g, uint32_t start, uint32_t end
    - **Outputs:** uint32_t
    - **Purpose:** Returns the weight of an edge.
    - **Pseudocode:** See above.
9. **graph_visit_vertex**
    - **Inputs:** Graph *g, uint32_t v
    - **Outputs:** None
    - **Purpose:** Marks a vertex as visited.
    - **Pseudocode:** See above.
10. **graph_unvisit_vertex**
    - **Inputs:** Graph *g, uint32_t v
    - **Outputs:** None
    - **Purpose:** Marks a vertex as unvisited.
    - **Pseudocode:** See above.
11. **graph_visited**
    - **Inputs:** const Graph *g, uint32_t v
    - **Outputs:** bool
    - **Purpose:** Checks if a vertex has been visited.
    - **Pseudocode:** See above.
12. **graph_print**
    - **Inputs:** const Graph *g
    - **Outputs:** None
    - **Purpose:** Prints a human-readable representation of the graph.
    - **Pseudocode:** See above.
13. **stack_create**
    - **Inputs:** uint32_t capacity
    - **Outputs:** Stack*
    - **Purpose:** Creates and initializes a new stack with given capacity.
    - **Pseudocode:** See above.
14. **stack_free**
    - **Inputs:** Stack **sp
    - **Outputs:** None
    - **Purpose:** Frees the memory allocated for the stack and sets the pointer to NULL.
    - **Pseudocode:** See above.
15. **stack_push**
    - **Inputs:** Stack *s, uint32_t val

    - **Outputs:** bool
    - **Purpose:** Pushes a value onto the stack.
    - **Pseudocode:** See above.
16. **stack_pop**
    - **Inputs:** Stack *s, uint32_t *val
    - **Outputs:** bool
    - **Purpose:** Pops a value from the stack.
    - **Pseudocode:** See above.
17. **stack_peek**
    - **Inputs:** const Stack *s, uint32_t *val
    - **Outputs:** bool
    - **Purpose:** Peeks at the top value of the stack.
    - **Pseudocode:** See above.
18. **stack_empty**
    - **Inputs:** const Stack *s
    - **Outputs:** bool
    - **Purpose:** Checks if the stack is empty.
    - **Pseudocode:** See above.
19. **stack_full**
    - **Inputs:** const Stack *s
    - **Outputs:** bool
    - **Purpose:** Checks if the stack is full.
    - **Pseudocode:** See above.
20. **stack_size**
    - **Inputs:** const Stack *s
    - **Outputs:** uint32_t
    - **Purpose:** Returns the number of elements in the stack.
    - **Pseudocode:** See above.
21. **stack_copy**
    - **Inputs:** Stack *dst, const Stack *src
    - **Outputs:** None
    - **Purpose:** Copies the contents of one stack to another.
    - **Pseudocode:** See above.
22. **stack_print**
    - **Inputs:** const Stack *s, FILE *outfile, char *cities[]
    - **Outputs:** None
    - **Purpose:** Prints the stack's elements as city names.
    - **Pseudocode:** See above.
23. **path_create**
    - **Inputs:** uint32_t capacity
    - **Outputs:** Path*
    - **Purpose:** Creates and initializes a new path with a given capacity.
    - **Pseudocode:** See above.
24. **path_free**

- **Inputs:** Path **pp
- **Outputs:** None
- **Purpose:** Frees the memory allocated for the path and sets the pointer to NULL.
- **Pseudocode:** See above.

25. **path_vertices**
    - **Inputs:** const Path *p
    - **Outputs:** uint32_t
    - **Purpose:** Returns the number of vertices in the path.
    - **Pseudocode:** See above.
26. **path_distance**
    - **Inputs:** const Path *p
    - **Outputs:** uint32_t
    - **Purpose:** Returns the total distance of the path.
    - **Pseudocode:** See above.
27. **path_add**
    - **Inputs:** Path *p, `uint32