Govind Pillai
gopillai@ucsc.edu
4/29/2021

CSE 13s Spring 2021
Assignment 4: The Circumnavigations of Denver Long
Design Document

PURPOSE:

The purpose of this lab is to find the hamiltonian path between a given set of points (locations). This is done by using a Graph struct (a matrix) to hold the "weight" of an edge in a path. For example, the length of the edge between vertices i and j, would be stored in the matrix at [i][j]. A path struct (Stack) is used to keep track of the many different paths tested. Finally, DFS (Depth-First Search) is implemented in order to find the shortest path.

DEFINITIONS:

Vertex: A point or location on the graph

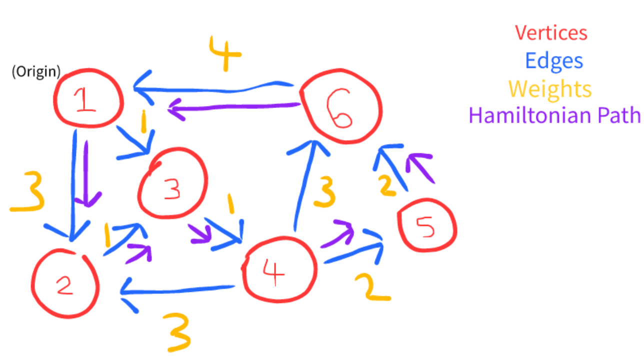
Edge: The path between two vertices on the graph

Hamiltonian Path: A hamiltonian path is a path between vertices on a graph in which every vertex is visited exactly once and the last vertex has an edge with the origin.

Graph Struct: A matrix that carries different values called "weights". The indices represent vertices and the value at two given indices represent the length of the edge between the two indices (or vertices).

Path struct: A stack that carries the visited vertices on the path so far. The Path struct also holds the total length of the path so far.

DIAGRAM:



GRAPH (based on above diagram)**:**

	1	2	3	4	5	6
1	0	3	1	0	0	0
2	0	0	1	0	0	0
3	0	0	0	1	0	0
4	0	0	0	0	2	3
5	0	0	0	0	0	2
6	4	0	0	0	0	0

PSEUDOCODE:

Graph.c:

```
graph_create(# of vertices, is_undirected) {
        Allocate space for graph
        set g→ vertices to # of vertices
        Set g→ undirected to is_undirected
        for all elements in visited array {
                Set to false;
        }
        For all elements in matrix {
                set indices to given weights;    (if undirected is true, set its reflection to the same weights)
        }
}

graph_delete(graph) {
        Free allocated space for graph
}

graph_vertices(graph) {
        Return g→ vertices
}
graph_add_edge(graph, vertex i, vertex j, weight k) {
        set value at matrix[i][j] to k
}
```

```
graph_has_edge(graph, vertex i, vertex j) {
        return if matrix[i][j] is non-zero
}
graph_edge_weight(graph, vertex i, vertex j) {
        return value at matrix[i][j]
}
graph_visited(graph, vertex v) {
        return value at visited[v]
}
graph_mark_visited(graph, vertex v) {
        set visited[v] to true
}
graph_mark_unvisited(graph, vertex v) {
        set visited[v] to false
}
```

Stack.c:

```
stack_create(capacity) {
        Allocate space for stack
        Set g→ top to 0
        set g→ capacity to capacity
        Allocate space for items
}
stack_delete() {
        free up space allocated for stack
        free up space allocated for items
}
stack_empty(stack) {
        Return true  if top is equal to 0, false otherwise
}
stack_full(stack) {
        Return true if top is equal to capacity, false otherwise
}
stack_size(stack) {
        Return top;
}
stack_push(stack, value x) {
        set items[top] equal to x;
        Increment top;
}
stack_peek(stack, *x) {
        set *x to value at items[top-1];
}
```

```
stack_pop(Stack, *x) {
        Decrement top;
        Set *x to value at items[top];

}
stack_copy(stack copy, stack source) {
        For all elements in source→ items {
                set copy→ items[i] to source→ items[i];
        }
}
```

Path.c:

```
path_create() {
        set vertices to stack using stack_create;
        Set length to 0;
}
path_delete() {
        call stack_delete();
}
path_push_vertex(Path, vertex v, Graph) {
        if(graph_has_edge(v, stack_peek(Path)) {
                Increment length by graph_edge(v, stack_peek(Path));
        }
        stack_push(Path, v);
}
path_pop_vertex(Path,  *v, Graph) {
        if(graph_has_edge(v, stack_peek(Path)) {
                Decrement length by graph_edge(v, stack_peek(Path));
        }
        stack_pop(Path, v);
}
path_vertices(Path) {
        stack_size(Path);
}
path_copy(Path copy, Path source) {
        stack_copy(copy, source);
        copy→ length = source→ length;
}
```

DFS pseudocode:

```
DFS(Graph, Vertex v, Current Path, Shortest Path) {
    set vertex v to visited;
    for(all vertices (j)) {
        If graph_has_edge(v,j) and j is not visited {
            path_push_vertex(Path, Vertex j);
            DFS(Graph, Vertex j, Current Path, Shortest Path);
        }
    }
    Set vertex v to unvisited;
    if current path length is shorter than shortest path length {
        Set shortest path to current_path;
    }
}
```