Project 4: Design Document

Rupakeerthana Vemulapalli

20778222

What class(es) did you design? What are the member variables and member functions for each of these classes?

```
//Edge Properties
struct Edge {
    double edge_distance;
};
```

```
struct Vertex{
    City* first_node; //
    Vertex* next_vertex;
};
```

```
struct City {
    string city_name;
    double shortest_distance;
    int degree;  //signified t

    Edge distance_from_src;
    string parent;
    bool min_extracted;
    City* next_city;
};
```

Class/Member variables/member functions:

```
class undirectedGraph{
    private:
    Vertex* vertex_head;
    Vertex* vertex_tail;
    int no_of_nodes;
    int no_of_edges;
    int no_of_min_extracted;

    public:
    undirectedGraph();
    ~undirectedGraph();

    void insert_city(string name);
    void setd(string name1, string name2, double d);
    void setd_helper(string name1, string name2, double d);
    void search_city(string name);

    void print_degree(string name); /*Number of directed edges from th
    void graph_nodes();
    void graph_edges();

    void print_distance(string name1, string name2); /*Cities must be
    void print_shortest_distance(string name1, string name2);
    void generate_shortest_distance(string name1, string name2);
    void set_min_extracted(Vertex *min_vertex);
    bool check_if_names_exist(string name1, string name2);
    Vertex* findmin();
    void relax(City* u, City* v);
    void modifyCity(City* v);
    //void evaluate_no_of_min_extracted();
    void initialization_shortest_distance(string name1, string name2);
    void print_path_shortest_distance(string name1, string name2);

    void clear();
    Vertex* find_city(string name);
};
```

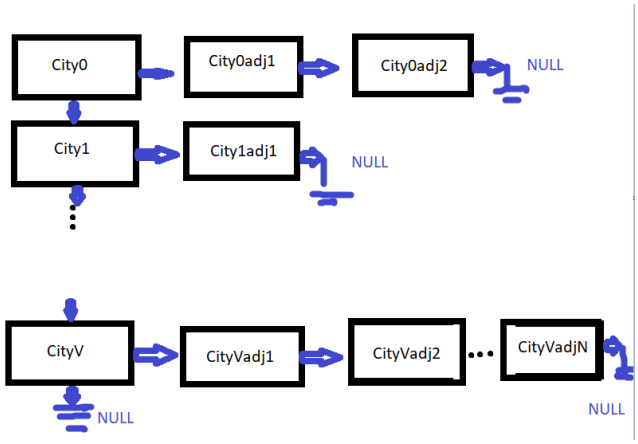
For each class, what are your design decisions regarding constructors/destructors?

```
undirectedGraph::undirectedGraph(){
    vertex_head = NULL;
    vertex_tail = NULL;
    no_of_nodes = 0;
    no_of_edges = 0;
    no_of_min_extracted = 0;
}
```

Describe your implementation.

As evident in the schematic below, my implementation of the algorithm started out with storing the cities as an adjacency matrix. Here, I stored the city's (vertices) are the major nodes in the linked list which are connected to each other. Likewise, all the adjacent nodes

corresponding to the vertex itself are also connected in a linked list. In my code, I also have a pointer- “vertex_head” and “vertex_tail” for the vertex linked list to make searching and adding an element easier (and better in terms of performance. Each node contains the properties outlined in the struct above. I decided to use the adjacency matrix due to reduced storage space at the cost of improved performance.



Insert – i

For insertion, I simply added the city as a major vertex in the linked list by adding the city as a next pointer to the tail, and changing the tail pointer. The error condition is checked initially before the insertion iterating through all the elements of the linked list, checking to see if there is a duplication in city name.

Set distance -setd

To set the edge distances, first there is a validation loop too check if both the cities exist by their city names. Then, if the distance is set between city 0 and city 1, we know that since it’s an undirected graph, the relationship between 0 and 1, and 1 and 0 must be recorded. And so, we would iterate through the adjacency list of city 0 and add city 1, and set the edge distance accordingly. We would essentially do the same for city 1, adding city 0 and setting the same edge distance. Also, when setting the distance, I update the degree of the vertex itself, for future use.

Search – s

To search for the city, simply iterate through each of the vertices in the major cities linked list. Return accordingly.

Degree – degree

To return the degree/the number of adjacent nodes to that city, I simply navigate to the city through a major loop iterating through all the vertices. Then, at the city of concern, I print out the degree which is stored as a property of that node.

Graph nodes – graph_nodes

To return the number of cities in the graph, I simply have a private member variable defined. And every time a new valid city gets declared in the initialization lop, the graph nodes variable is incremented.

Graph edges- graph_edges

Similar to the graph nodes, the graph edges is also a private variable gets incremented every time a new edge distance is set.

Printing the distance between 2 cities – d

First checks whether City2 is adjacent to City1. If it is not adjacent, it returns an error. If it is adjacent, then we know what printing the distance is valid. To print the distance, the edge distance stored as a part of the struct of the node is printed.

Shortest_d

To print the shortest distance I essentially use Dijkstra's algorithm. I first initialize the distance to infinity which is the largest possible int number (2147483647). Also, the parents of each of the nodes are set to NULL. Essentially, each node consists of a property called min_extracted, which is a Boolean that is set/not set based on whether the condition is true for that node. In doing so, I avoid the use of the queue (but still need to iterate through all the nodes to extract the min), although the general algorithm remains the same. Then I essentially execute Line 8-12 as described in class to store the shortest distance in each node from the source. If for some reason, the element is not reachable, so still NULL by the end, we know that this is an error and the code prints a failure.

Print Path

After finding the shortest path, I essentially have a loop to go from the destination following the "parent" back to the src node and storing this in an array. The array is then printed in reverse order. Error conditions are similar to the previous.

Clear

To clear all the nodes, I essentially point each of the vertices in the linked list to NULL.

If It is expected that your implementation has asymptotic upper/tight bound, you should also describe how you have achieved this (or better) runtime in your implementation.

Function	Calculated Running time
i	O(1)
Setd	O(V+E)
s	O(V)
degree	O(V)
Graph_nodes	O(1)
Graph_edges	O(1)
d	O(V+E)
Shortest_d	O(V^2)
Print_path	O(V*E)
clear	O(V)