Lab 4 Design Document

1. Class Designs

1.1 Graph Class

Description: This is my Graph class, which represents the weighted nondirected graph that is created
Constructor*:* My constructor for my Graph class will create an adjacency list of 500,000 to store all the adjacent nodes to each node in the graph. It also initialized a vertex dictionary in order to store all the vertices that are inserted in the Graph
Destructor*:* My Destructor will go through each index of the adjacency list and delete each linked list and also delete the vertex dictionary
Private Member Variables:

- *LinkedList **adjacency_list*: an array of LinkedList pointers, which will store the adjacent vertices for each vertex
- *VertexDictionary *vertex_dictionary:*a pointer to a VertexDictionary object that sores all the vertices added to the graph

Member Functions:

1. *void loadFile(fstream& fin):* This function will read the first four numbers of a text file and then call the insert function to insert an edge into the graph
2. *void insert(int a, int b, double d, double s):* This function will insert an edge into the graph, it will first use the vertex dictionary to check if that vertex already exists or not. If not, then it will create a new vertex and put it in the dictionary. It will then check the list of adjacent vertices to one vertex to see whether or not the edge already exists. If so, it will just update the distance and speed_limit values. If not, then it will create a new edge and add it to the linked list of both vertices (since we are working with a non-directed graph).
3. *void printAdjVertices(int a):* This function will first check if the vertex exists. If it does then it will go through the linked list of adjacent nodes to the vertex chosen and print those out one by one.
4. *void deleteVertex(int a):* This function will first check if the vertex exists. If so, then it will call the deleteVertices function to delete the entire linked list of adjacent nodes to the delete vertex as those edges will no longer exist. The function will return an array that stores all of the adjacent nodes that was deleted. Then I use that array to go through the other linked lists of the adjacent nodes and traverse the linked list until I see the deleted vertex and delete it. (have to delete the other linked lists because the graph is non-directed)
5. *void updateAdjustmentFactor(int a, int b, int A):* This function will go to the linked list of both vertices and traverse the list to find the other vertex and change their adjustment factor.
6. *void findPath(int a, int b):* This is the Dijkstra's algorithm. First I initialize a distance array with the max number of a double variable and then initialize the source vertex with a distance of 0. After that I add the adjacent vertices of the source vertex to a priority queue, and then compare, updating the distance according and pop them into the queue. When the queue is empty, it signals that the algorithm is over and the smallest distance is found. During the loops, if I see that one of the paths traverses back to the destination vertex the user inputted, then I will print out that vertex. At the end, I reinitialize the distance and visited arrays with their initial values so they can be used again if findPath or findLowest is called.
7. *void updateFile(fstream& fin):* Just like the loadFile function, it will read the first three numbers of a text file and then call the updateAdjustmentFactor command
8. *void findLowest(int a, int b):* This is also the Dijkstra's algorithm too. Except it does not print the intersection numbers of the vertices that are in the path of vertex a and b. However, it performs the entire algorithm and then from the distance array, it finds the destination vertex that was inputted by the user and prints out their lowest path weight from a to b. At the end, I reinitialize the distance and visited arrays with their initial values so they can be used again if findPath or findLowest is called.

1.2 LinkedListClass

Description: This is my linkedlist class where I use to store all the adjacent vertices to a vertex in the graph
Constructor: My constructor sets the head vertex variable to 0 since the linked list is currently empty right now
Destructor*:* My destructor will traverse through the entire linked list and delete all the vertices one by one until everything is deleted and memory is all reallocated.
Private Member Variables:

- *Vertex *head_vertex:* This is a pointer variable that will point to the head of the linked list. This allows me to have access to the entire list by just having a head variable

Member Functions:

1. *void addNewEdge(int vertex, double d, double s, int A):* This function creates a new vertex and traverses through the entire linked list until it gets to the end and inserts the new vertex at the very end.
2. *bool checkEdge(int intersection_num, double d, double s)/(int intersection_num):* I have two versions of this function. The first one will check whether an edge between two vertices already exist. If they do exist then the distance and

speed of that edge will just be updated. In the second version of the function, it only takes in the intersection_num as a parameter and it will return true or false whether the edge exists or not.

3. *void printVertices():*This function will traverse through the entire linked list and print out their insersection numbers
4. *VertexDictionary* deleteVertices:*This Function will delete all of the adjacent vertices in a linked list but also put those vertices in an array and return it. This way we know which adjacent vertex linked lists we need to visit in order to continue deleting the vertex.
5. *void deleteVertex(int intersection):* This will traverse through the entire linked list until you find the delete vertex. Then it will delete the current vertex and reconnect the other vertices
6. *bool updateAdjFactor(int intersection, int A):* This will traverse through the entire linked list until you find the vertex that needs to update their AdjFactor and it will call another function to update the A
7. *void addToQueue(PriorityQueue *priority_queue):* This function traverses through the linked list and inserts each vertex in a priority_queue for Dijkstra's algorithm
8. *int findVertex(int intersection):* This function will traverse through the entire linked list and find the vertex that you are looking for but return the traveltime of that vertex.
9. *Vertex* get_head:*This is a getter function for the head of the linked list

1.3 VertexDictionary Class

Description: My VertexDictionary class is a dynamic array class of Vertex objects that stores and keeps track of all the vertices that are inserted in the graph.
Constructor*:* My conductor will initialize a dynamic array and then set the max capacity to 10. Capacity will always increase by 10 every time I see her.
Destructor *:* In my destructor, I traverse through the entire array and delete each vertex and delete the entire array at the end.
Private Member Variables:

- *int size:* This variable stores the current size of the array and the array will increase when the size is equal to the capacity
- *int capacity:*This variable stores the total capacity of the array and it will increase as the array becomes full
- *Vertex **vertex_array:* This is a dynamic array of Vertex pointers

Member Functions:

1. *bool checkVertex(int intersection):* This function just traverses through the dictionary to see if the vertex is in the graph or not.
2. *void addVertex(vertex *vertex):* This function adds a new point into the dynamic array. If the array is full, the capacity is increased and the points are inserted into the array with the bigger capacity.
3. *void swapVertices (int index, int parent_index):* This function is used for the priority queue. It will switch the two vertices as given in the parameters in order to maintain the characteristics of a priority queue.
4. *int get_size(), int getIntersection(int i), double get_traveltime(int i), Vertex* get_vertex(int i):* getter functions for private member variables
5. *void set_vertex(int i, Vertex *vertex), void set_traveltime(int i, double t):* setter functions for private member variables

1.4 Vertex Class

Description: This is my vertex calss. This is where I store all of the information of a vertex, including my intersection number, distance, adjustment factor, speed limit, etc.
Constructor: I have three different constructors. The first one does not have any parameters at all. This is for when you need a new vertex object, but you don't have enough information to add to it. The second one has an intersection parameter and it is when you don't/need the distance, and speed limit information. The last constructor has the intersection number, distance, speed limit and the adjustment factor. This is used when the insert function is called and a vertex is created with all of the information.
Destructor *:* My destructor sets the next_intersection pointer to nullptr. This disconnects the different vertices from the linked list.
Private Member Variables:

- *int intersection_num, source_num, adjustment_factor, double distance, speed_limit, travel_time, min_distance:* important variables and information given to a vertex
- *Vertex *next_intersection:* a pointer variable that will point to another Vertex object. This allows the Vertices to be able to connect to one another, forming a linked list.

Member Functions:

1. *int get_intersection(), Vertex* get_next(), double get_traveltime():* getter functions for private member variables
2. *void set_next(Vertex *new_next_vertex), void set_distance(double d), void set_speed(double s), void set_intersection(int intersection), set_traveltime(int t):* setter fuctions for private member variables
3. *bool updateAdjFactor(int A):* This function will update the adjustment factor with the new A input if the new A input is different. If A is equal to zero, then the travel time will become infinity.

1.5 PriorityQueue Class

Description: This is my priority queue class which will store all of the adjacent vertices that I pass by and find its minimum travel time for the Dijkstra's Algorithm

Constructor*:* My constructor will create a new dynamic array variable that will store the priority queue.

Destructor *:* My destructor will delete the dynamic array in order to free up the memory used for it .

Private Member Variables:

- *int size:* variable to keep track of the size of the dynamic array
- *VertexDictionary *priority_queue:* pointer variable that points to the dynamic array

Member Functions:

1. *void insertInPQ(Vertex *vertex):* This function will add a new vertex into the priority queue. It will also check the travel time of the new vertex and swap the vertices accordingly to preserve the structure.
2. *void heapify(int i):* This function is called to heapify the priority queue again after the root vertex was taken away. It will recursively call the heapify function to swap with other vertices until the min heap structure is restored again.
3. *Vertex* extractMin():* This function is called to extract the min vertex in the priority queue. After removing the root vertex, it will heapify the priority queue again
4. *int get_size():*getter function for the size of the array

2. Expected Runtime

2.1 Dijkstra's Algorithm

Inserting a vertex into the priority queue, will take O(log|V|) time because the priority queue will need to restore its min heap structure every time a new vertex is inserted. Extracting the min vertex will also the same amount of time as it will need to be restored again as well. The loop in the algorithm that will traverse through all the adjacent vertices will take O(|E|) amount of times in the worst case, which is if all the edged are visited only once.

Combining these two runtimes, you will see that in the worst case, the runtime will be O(|E|log|V|).

3. UML Design