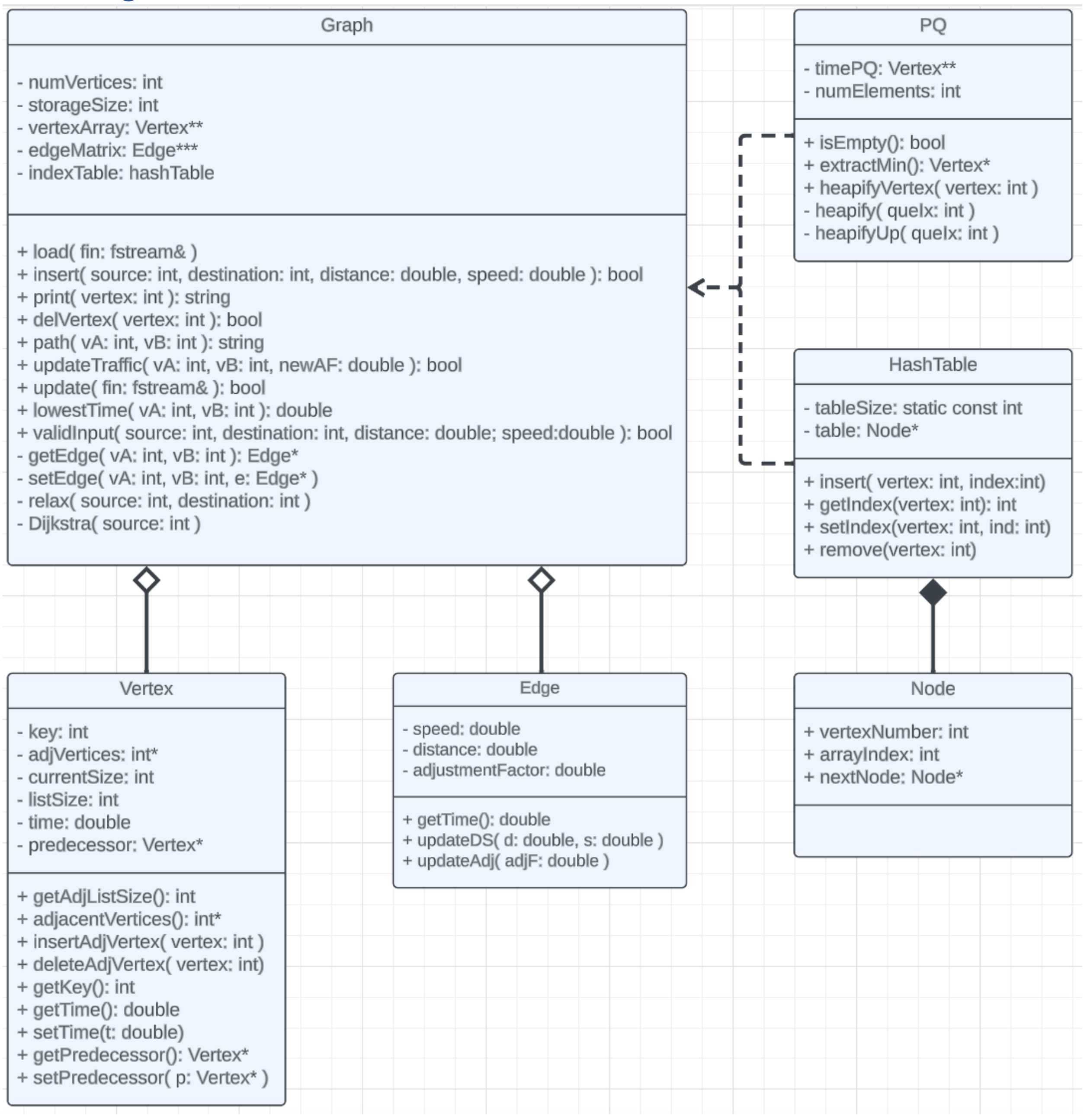
Emergency Response Graph (Project 4)

20992999 k25chen

Overview

An object of the Graph class is used to represent the undirected, weighed graph of this project. Each vertex is represented by a Vertex object and is stored in a vertex array, and a hash map is used to store each vertex with their array index value. The edge between two vertex is represented by an Edge class, with edges being stored in a 2D array. Vertex array index of each vertex is used to index into the 2D array when finding the edge between two vertices. Dijkstra's algorithm is implemented to realize various functionalities of the graph.

UML Diagram


Graph
- numVertices: int
- storageSize: int
- vertexArray: Vertex**
- edgeMatrix: Edge***
- indexTable: hashTable
+ load(fin: fstream&)
+ insert(source: int, destination: int, distance: double, speed: double): bool
+ print(vertex: int): string
+ delVertex(vertex: int): bool
+ path(vA: int, vB: int): string
+ updateTraffic(vA: int, vB: int, newAF: double): bool
+ update(fin: fstream&): bool
+ lowestTime(vA: int, vB: int): double
+ validInput(source: int, destination: int, distance: double; speed:double): bool
- getEdge(vA: int, vB: int): Edge*
- setEdge(vA: int, vB: int, e: Edge*)
- relax(source: int, destination: int)
- Dijkstra(source: int)
PQ
- timePQ: Vertex**
- numElements: int
+ isEmpty(): bool
+ extractMin(): Vertex*
+ heapifyVertex(vertex: int)
- heapify(queIx: int)
- heapifyUp(queIx: int)
HashTable
- tableSize: static const int
- table: Node*
+ insert(vertex: int, index:int)
+ getIndex(vertex: int): int
+ setIndex(vertex: int, ind: int)
+ remove(vertex: int)
Vertex
- key: int
- adjVertices: int*
- currentSize: int
- listSize: int
- time: double
- predecessor: Vertex*
+ getAdjListSize(): int
+ adjacentVertices(): int*
+ insertAdjVertex(vertex: int)
+ deleteAdjVertex(vertex: int)
+ getKey(): int
+ getTime(): double
+ setTime(t: double)
+ getPredecessor(): Vertex*
+ setPredecessor(p: Vertex*)
Edge
- speed: double
- distance: double
- adjustmentFactor: double
+ getTime(): double
+ updateDS(d: double, s: double)
+ updateAdj(adjF: double)
Node
+ vertexNumber: int
+ arrayIndex: int
+ nextNode: Node*

Class Functions

Node Class

Each object stores the vertex (the key) and the array index. Since collision in the hash table is resolved using chaining, each Node also has a pointer to the next node in the table slot.

HashTable Class

The hash table is used in Graph to store vertex-index pairs. Since valid vertices range from 1-500000, the hash function is simply the division method (key % m). The table is represented by an array of Node pointers, where each pointer points to the first Node in a slot.

Function	Behavior
insert(int vertex, int index)	Dynamically create a new Node and insert the key-value pair vertex-index into the hash table.
getIndex(int vertex) returns: int	Return the index associated with the given vertex in the hash table. If the vertex was not stored in the table, return -1.
remove(int vertex)	Delete the Node with the given vertex, remove it from the hash table by resetting pointers.

The class constructor initializes every slot in the table to null pointer. The class destructor iterates through each slot, delete the dynamically allocated Nodes.

PQ Class

A min-heap created from an array of Vertex pointers is used to implement the priority queue ADT, with respect to the time attribute of each Vertex. This class is used in the Dijkstra algorithm in Graph.

Function	Behavior
heapify(int index)	Move the Vertex at index down the heap until it satisfies the min-heap property.
heapifyUp(int index)	Move the Vertex at index up the heap until it satisfies the min-heap property.
extractMin() returns: Vertex*	Return the root vertex in the heap. Move the last Vertex to the root and call heapify on index 0 to restore min-heap. Decrement vertex count.
heapifyVertex(int vertex)	This function is called in Graph when the time value of Vertex "vertex" has been modified. Since time of Vertices will only decrease after initialization in this project, call heapifyUp() on the index of the given vertex.
PQ(Vertex** arr, int num)	Initialize the heap array to size num, the number of vertices, and copy each element in the given array into the heap. Build a min-heap.
~PQ()	Delete the heap array.

Edge Class

Each object represent an undirected, weighed edge between two vertices. The weight, time, is determined by its three attributes: time = distance/(speed*adjustmentFactor)

Function	Behavior
getTime() returns: double	If the adjustment factor is non-zero, return the time according to formula. Otherwise, return the numeric limit infinity.

updateDS(double d, double s)	Set the distance and speed value to d and s, respectively.
updateAdj(double A)	Set the adjustment factor attribute to A.

Empty constructor and destructors.

Vertex Class
Each Vertex object stores their key (1-500000), an integer array of adjacent vertices, and a time (double) value and a predecessor Vertex pointer that are both updated in the Dijkstra algorithm.

Function	Behavior
getAdjListSize() returns: int	Return the size of the adjacent vertices array.
adjacentVertices() returns: int*	Dynamically create a new integer array identical to the adjacent vertices array, and return the pointer. The caller of this function is responsible for deleting the allocated memory.
insertAdjVertex(int v)	Insert the input vertex into the adjacency array, dynamically resize if array is full.
deleteAdjVertex(int v)	Delete the given vertex from the adjacency array.
Vertex(int k)	Constructor initialize key to k and allocate an empty array of initial size.
~Vertex()	Destructor deletes the allocated array.

There are also the standard get and set functions for key, time, and predecessor attributes.

Graph Class
It contains an array of Vertex pointers that stores all the vertex in the graph, and a 2D array of Edge pointers that store the edge between vertices. Index of the vertices are stored in a hash table.

Function	Behavior
Graph()	Initialize all attributes to zero or null pointer.
~Graph()	Iterate through the vertex array and edge 2D array, deleting each element.
getEdge(int a, int b) returns: Edge*	Since this graph will be undirected, only one edge need to be stored between two vertices. Edge between two vertex a, b of index A, B will be stored at index [min(A, B)][max(A, B)] in the 2D array. Return the Edge pointer stored at that index.
setEdge(int a, int b, Edge* e)	Store the Edge pointer e at the correct index (same as in getEdge()).
validInput(int s, int de, double di, double s) returns: bool	Helper function that checks if the input vertex, distance, and speed are valid. Return false if any of the inputs are not valid.
load(fstream& fin)	Answers “LOAD” command. Repeatedly call the insert() function using fin as input, until the end of fin.
insert(int s, int de, double di, double s) returns: bool insert(int s, int de, double di, double s)	Answers “INSERT” command. If any of the s or d vertex is not yet stored (indexTable.getIndex returns -1), create new Vertex objects and store them. If the edge between s and d is a

returns: bool	null pointer in the 2D array, create a new Edge object of distance di and speed s and store it. Otherwise, call updateDS() on the edge and update the values. Dynamically resize arrays if capacity is near full after insertion.
print(int vertex) returns: string	Answers “PRINT” command. Find the Vertex using the hash table and return its adjacency list as a string.
delVertex(int vertex) returns: bool	Answers “DELETE” command. Loop through the adjacency list and delete every edge, then delete the vertex from array and index table. Return false if graph empty/not stored.
path(int a, int b) returns: string	Answers “PATH” command. Call Dijkstra() algorithm on a, then at b trace the predecessor pointer all the way to a, return a concatenated string of Vertex keys encountered.
updateTraffic(int a, int b, double newA) returns: bool	Answers “TRAFFIC” command. Call getEdge() to get the edge between a, b, and update the adjustment factor attribute. Return true if valid inputs and edge exists.
update(fstream& fin) returns: bool	Answers “UPDATE” command. Call updateTraffic() using fin as input until the end of fin.
lowestTime(int a, int b) returns: double	Answers “LOWEST” command. Call Dijkstra() algorithm on a and return the time value on b.
relax(int s, int d)	If time at d is larger than the time it takes to go to s and travel edge s to d, update time at d to the shorter time and set its predecessor to s.
Dijkstra(int source)	Perform the Dijkstra algorithm on the source vertex. It first initialize all time at vertices to infinity and predecessor to null pointer. Starting with source being time zero, create a min-time priority queue of all vertices. Then, it extracts the minimum from the queue one at a time until the queue is empty, and each time calling relax() on all of minimum vertex’s adjacency vertices, re-adjusting the priority queue using heapifyVertex() after every adjacent vertex’s time value was updated.

Runtime Requirements

Dijkstra algorithm first has an initialization loop running V, the number of vertices, times, and a second loop that again runs V times. In the second loop, extractMin() and heapifyVertex() are used. Since they are essentially heapify functions of a min-heap, they have running time logV. In the worst case, every relax() changes the time value and we have to update the priority queue for every edge. We extractMin() once for every vertex. Thus the worst case running time is O(ElogV + VlogV) = O((E+V)logV), where E is the number of edges and V is the number of vertices.