Theeraj
21045628

Project 4 -Design Document

05/12/2023

Classes:

In my code, I have 7 classes- Edge, Vertex, resizableArray, Graph, rsAI, PriorityQueue, and illegal_exception that work together to form a graph data structure with several functionalities. The **Graph** class serves as the core structure, maintaining a resizable array of vertices (**resizableArray**) that contains information about individual vertices (**Vertex**) and their associated edges (**Edge**).
The **PriorityQueue** class facilitates Dijkstra's algorithm for finding the shortest path in the graph. The **rsAI** class is a simple resizable array designed for integers. These classes work collaboratively to provide operations such as inserting or updating edges between vertices, printing neighbors, deleting vertices, handling traffic adjustments, updating the graph based on a file, and finding and printing the shortest path or lowest cost between two vertices. My **Graph** class is the central class that controls all these functionalities by utilizing the other classes when necessary, providing a comprehensive set of tools for manipulating and analyzing the underlying graph structure. I also have the **illegal_exception** class that works to handle exceptions.

1) Graph

- Member Variables:
- resizableArray* vertices: This is a pointer to a resizable array, an instance of the resizableArray class. It dynamically manages an array of Vertex objects, serving as the main data structure to store vertices in the graph.
- const double SUB_INFINITY: A constant representing positive infinity, utilized for initializing distances in various algorithms within the graph.
- Member Functions:
- **The Constructor** - Graph(int initCapacity = 500000): Initializes a Graph object with an optional initial capacity (initCapacity) for the vertices array. It creates an empty graph structure, setting the default initial capacity to 500,000. Return Type: None. Parameters: initCapacity, an optional parameter specifying the initial capacity of the resizable array.
- **The Destructor** - ~Graph(): Frees the memory allocated for the vertices array. Prevents memory leaks by deleting the resizable array when a Graph object is no longer needed. Return type and Parameters are None.
- **Vertex& getVertexById(int id)**: Function: Retrieves a reference to a Vertex object based on the provided vertex ID (id).
  Return Type: Reference to a Vertex object.
  Parameters: int id: The ID of the vertex to retrieve.
- **bool edgeExists(int from, int to):** Function: Checks if there exists an edge between two vertices specified from and to.
  Return Type: Boolean indicating whether an edge exists (true) or not (false).
  Parameters: int from: ID of the source vertex. int to: ID of the destination vertex.
- **void addEdge(Vertex& vertexA, Vertex& vertexB, double d, double s):** Function: Adds an edge between two vertices (vertexA and vertexB) with a given distance (d) and speed limit (s).
  Return Type: Void.
  Parameters: Vertex& vertexA: Reference to the first vertex. Vertex& vertexB: Reference to the second vertex. double d: Distance of the edge. double s: Speed limit of the edge.
- **void insert(int a, int b, double d, double s):**
  Description: This function is responsible for inserting or updating an edge between vertices with IDs a and b in the graph. If an edge between these vertices already exists, it updates the distance d and speed limit s. If not, it creates a new edge with the provided parameters.
  Parameters: a (int): ID of the first vertex. b (int): ID of the second vertex. d (double): Distance between the vertices. s (double): Speed limit for the edge.
  Return Type: void.
- **void print(int a)**:
  Description: This function prints the adjacent vertices of the vertex with ID a. If the vertex does not exist or has no edges, it prints an empty line.
  Parameters: a (int): ID of the vertex to print.
  Return Type: void.
- **void deleteVertex(int a):**
  Description: This function deletes the vertex with ID a from the graph, along with all edges connected to it. If the vertex is successfully deleted, it prints "success"; otherwise, it prints "failure."
  Parameters: a (int): ID of the vertex to delete.
  Return Type: void.

- **bool traffic(int a, int b, double A):**

Description: This function updates the adjustment factor (adjF) for the edge between vertices with IDs a and b to the provided value A. It returns true if the update is successful, and false otherwise.
Parameters: a (int): ID of the first vertex. b (int): ID of the second vertex. A (double): New adjustment factor.
Return Type: bool.

- **void path(int a, int b):**
  Description: This function finds and prints the shortest path from the vertex with ID a to the vertex with ID b using Dijkstra's algorithm. If no path exists, it prints "failure."
  Parameters: a (int): ID of the source vertex. b (int): ID of the destination vertex.
  Return Type: void.
- **bool update(const std::string& filename):**
  Description: This function reads traffic update data from a file specified by filename. It updates the adjustment factors for corresponding edges in the graph. If any update is successful, it prints "success"; otherwise, it prints "failure."
  Parameters: filename (const std::string&): Name of the file containing traffic updates.
  Return Type: bool.
- **void lowest(int sourceVertexId, int destinationVertexId):**
  Description: This function finds and prints the lowest weighted path (minimum time) from the vertex with ID sourceVertexId to the vertex with ID destinationVertexId using Dijkstra's algorithm. If no path exists, it prints "failure."
  Parameters: sourceVertexId (int): ID of the source vertex. destinationVertexId (int): ID of the destination vertex.
  Return Type: void.

2) PriorityQueue

- Member Variables:
- **distances (double*):** Pointer to an array of distances.
- **vertices (int*):** Pointer to an array of vertex IDs.
- **size (int):** Tracks the current number of elements in the priority queue.
- **capacity (int):** Indicates the current capacity of the priority queue.
- Member Functions:
- **PriorityQueue(int capacity):** Constructor initializes the priority queue with the specified capacity.
- **~PriorityQueue():** Destructor deletes the dynamically allocated arrays.
- **bool empty() const:** Checks if the priority queue is empty. Return type is bool and parameter is None
- **void push(int vertex, double distance):** Adds a vertex with its distance to the priority queue. Return type is void, parameters are vertex and distance.
- **int top() const:** Returns the vertex with the minimum distance. Return type is int, No parameters
- **void pop():** Removes the vertex with the minimum distance. Return type is void, no parameters.
- **heapifyUp(int i):** Parameter is i - The index of the element that needs to be moved up in the heap. Return Type is void
  Description: This function is used in the context of a min-heap. It ensures that the element at index i maintains the heap property by moving it up the heap until the parent-child relationship is restored.
- **heapifyDown(int i):** Parameter is i - The index of the element that needs to be moved down in the heap. Return Type is void
  Description: This function is used in the context of a min-heap. It ensures that the element at index i maintains the heap property by moving it down the heap until the parent-child relationship is restored.
- **swap(int i, int j)**: Parameters are i, j - The indices of the elements to be swapped. Return Type: void
  Description: Swaps the elements at indices i and j in the heap.
- **parent(int i) const:** Parameter is i - The index of the element for which the parent index is needed. Return Type is int.
  Description: Returns the index of the parent of the element at index i in the heap.
- **leftChild(int i) const:** Parameter is i - The index of the element for which the left child index is needed. Return Type is int.
  Description: Returns the index of the left child of the element at index i in the heap.
- **rightChild(int i) const:** Parameter is i - The index of the element for which the right child index is needed. Return Type is int.
  Description: Returns the index of the right child of the element at index i in the heap.

3) Edge

- Member Variables:
- **destination (int)**: Represents the ID of the destination vertex.
- **distance (double):** Stores the distance of the edge.
- **speedLimit (double):** Indicates the speed limit of the edge.
- **adjF (double):** Adjustment factor for traffic conditions.

- Member Functions:
- **Edge():** Default constructor initializes members to default values.
- **Edge(int dest, double dist, double speed):** Parameterized constructor sets members based on provided arguments.
- **Constructor Design**: The default constructor sets default values for the edge properties. The parameterized constructor allows for the initialization of edge properties with specific values.
- **Destructor Design:** No specific destructor is defined since there is no dynamic memory allocation in the class.

4) Vertex

- Member Variables:
- **id (int):** Represents the ID of the vertex.
- **edges (Edge*):** Pointer to an array of edges connected to the vertex.
- **numEdges (int):** Tracks the number of edges connected to the vertex.
- Member Functions:
- **Vertex():** Default constructor initializes members to default values.
- **Vertex(int vertexId):** Parameterized constructor sets the vertex ID and initializes other members.
- **~Vertex():** Destructor deletes the dynamically allocated array of edges.
- **Constructor Design:** The default constructor initializes the vertex with default values. The parameterized constructor sets the vertex ID and initializes other members.
- **Destructor Design:** The destructor is responsible for freeing the memory allocated for the array of edges.

5) resizableArray

- Member Variables:
- **array (Vertex*):** Pointer to an array of vertices.
- **size (int):** Tracks the current number of vertices in the array.
- **currentCapacity (int):** Indicates the current capacity of the array.
- Member Functions:
- **resizableArray(int initCap):** Constructor initializes the array with the specified initial capacity.
- **~resizableArray():** Destructor deletes the dynamically allocated array.
- **int getSize():** Returns the current size of the array.
- **int getCapacity():** Returns the current capacity of the array.
- **Vertex& getAt(int indx):** Returns the vertex at the specified index.
- **bool push(Vertex val):** Adds a vertex to the array.
- **void resize(int newSize):** Resizes the array to the specified size.
- **Constructor Design:** The constructor initializes the array with a specified initial capacity.
- **Destructor Design:** The destructor is responsible for freeing the memory allocated for the array.

6) rsAI

- Member Variables:
- **data (int*):** Pointer to an array of integers.
- **size (int):** Tracks the current number of elements in the array.
- **capacity (int):** Indicates the current capacity of the array.
- Member Functions:
- **rsAI():** Constructor initializes the array with a default capacity.
- **~rsAI():** Destructor deletes the dynamically allocated array.
- **void push_back(int value):** Adds an integer to the array.
- **int* get_data() const:** Returns a pointer to the array.
- **int get_size() const:** Returns the current size of the array.
- **Constructor Design:** The constructor initializes the array with a default capacity.
- **Destructor Design:** The destructor is responsible for freeing the memory allocated for the array.

7) illegal_exception

The **illegal_exception** class is a simple exception class that inherits from **std::exception**. Its primary purpose is to throw an exception with a predefined message indicating an "illegal argument." The **what** function provides the error message when the exception is caught.

[1] **Runtime Analysis**

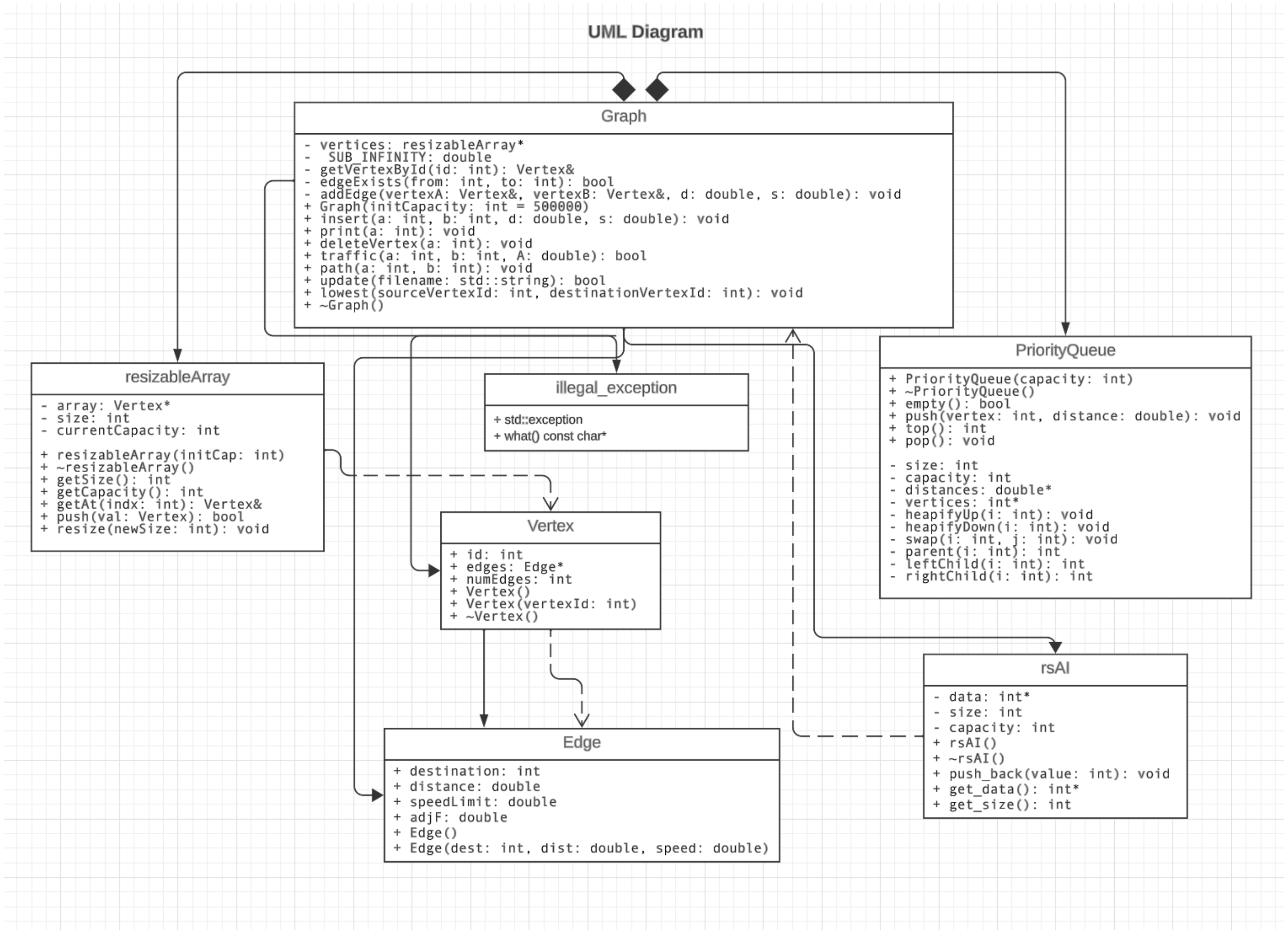
Dijkstra’s Algorithm runtime – O(|E|log(|V|))

The PATH and LOWEST commands make use of my Dijkstra’s algorithm implementation. I made a PriorityQueue class for the Dijkstra’s algorithm. The core of Dijkstra's algorithm involves the use of a priority queue to efficiently select the vertex with the minimum distance at each step. As I explain below, priority queue operations are crucial for meeting the required time complexity.

I use a binary heap-based priority queue. The push operation, which inserts an element, has a time complexity of O(log|V|). This is because it needs to maintain the heap property by traversing the height of the heap, which is logarithmic in the number of elements. Similarly, the pop operation, which extracts the minimum element, also has a time complexity of O(log|V|). The relaxation process occurs when I consider updating the distances to adjacent vertices. This process happens once for each edge in the graph during the execution of Dijkstra's algorithm. I have a loop that iterates over each edge once. For each edge, I perform constant-time operations, including calculations and potential updates to the priority queue.

Overall, the initialization of distances and parents take O(|V|) time. For each vertex, I perform at least one pop operation, and for each edge, I might perform a push operation. Since there are E edges, and each push operation is O(log|V|), the overall time complexity of these operations is O(E*log|V|).

Combining all these observations, the total time complexity of Dijkstra's algorithm in my implementation is dominated by the priority queue operations, resulting in a runtime of **O(|E|*log|V|)**. This analysis is based on the fact that the number of edges |E| is at least |V|-1, or |E|>= |V| which holds true in connected graphs.


UML Diagram
Graph
- vertices: resizableArray*
- SUB_INFINITY: double
- getVertexById(id: int): Vertex&
- edgeExists(from: int, to: int): bool
- addEdge(vertexA: Vertex&, vertexB: Vertex&, d: double, s: double): void
+ Graph(initCapacity: int = 500000)
+ insert(a: int, b: int, d: double, s: double): void
+ print(a: int): void
+ deleteVertex(a: int): void
+ traffic(a: int, b: int, A: double): bool
+ path(a: int, b: int): void
+ update(filename: std::string): bool
+ lowest(sourceVertexId: int, destinationVertexId: int): void
+ ~Graph()
resizableArray
- array: Vertex*
- size: int
- currentCapacity: int
+ resizableArray(initCap: int)
+ ~resizableArray()
+ getSize(): int
+ getCapacity(): int
+ getAt(indx: int): Vertex&
+ push(val: Vertex): bool
+ resize(newSize: int): void
illegal_exception
+ std::exception
+ what() const char*
Vertex
+ id: int
+ edges: Edge*
+ numEdges: int
+ Vertex()
+ Vertex(vertexId: int)
+ ~Vertex()
Edge
+ destination: int
+ distance: double
+ speedLimit: double
+ adjF: double
+ Edge()
+ Edge(dest: int, dist: double, speed: double)
PriorityQueue
+ PriorityQueue(capacity: int)
+ ~PriorityQueue()
+ empty(): bool
+ push(vertex: int, distance: double): void
+ top(): int
+ pop(): void
- size: int
- capacity: int
- distances: double*
- vertices: int*
- heapifyUp(i: int): void
- heapifyDown(i: int): void
- swap(i: int, j: int): void
- parent(i: int): int
- leftChild(i: int): int
- rightChild(i: int): int
rsAI
- data: int*
- size: int
- capacity: int
+ rsAI()
+ ~rsAI()
+ push_back(value: int): void
+ get_data(): int*
+ get_size(): int

Citations and References:

[1] ChatGPT, For the Dijkstra’s algorithm, calculate the runtime complexities. OpenAI [Online]. https://chat.openai.com/ (December 5, 2023)