**Data Structures and Algorithm Assignment**

**Problem Statement:**

1. Create an ADT for an undirected, weighted graph, represented by an adjacency matrix. Use whole numbers for the weights.
2. Write a simple application that simulates an airline's graph of ticket prices. The program should read in vertices and prices and keep an array that maps vertices to the names of the airports. For each route that the airline does not fly, store Zero. Once the graph is set up, the program should accept queries consisting of the names of two airports and either report a cost or report that no flight is available.
3. The (Depth-First Search DFS) or (Breadth First Search BFS) algorithm can be used to traverse all the vertices of a graph

**Solution Provided:**

**Language Used:** Java

**Version:** JDK11

**Code Structure:**

1. **Graph Class**

Graph class contains the main logic of the application

*Global Variables:*

|  |  |  |  |
| --- | --- | --- | --- |
| **S.no** | **Variable Name** | **Data Type** | **Description** |
| 01 | vertexList | List<String> | Used to store the vertices (airports) |
| 02 | adjList | List<List<Edge>> | Used to store the edge details between two vertices |
| 03 | vertexToIndexMap | Map<String, Integer> | Used to store key-value pairs of vertex name and index it is stored in the vertexList |

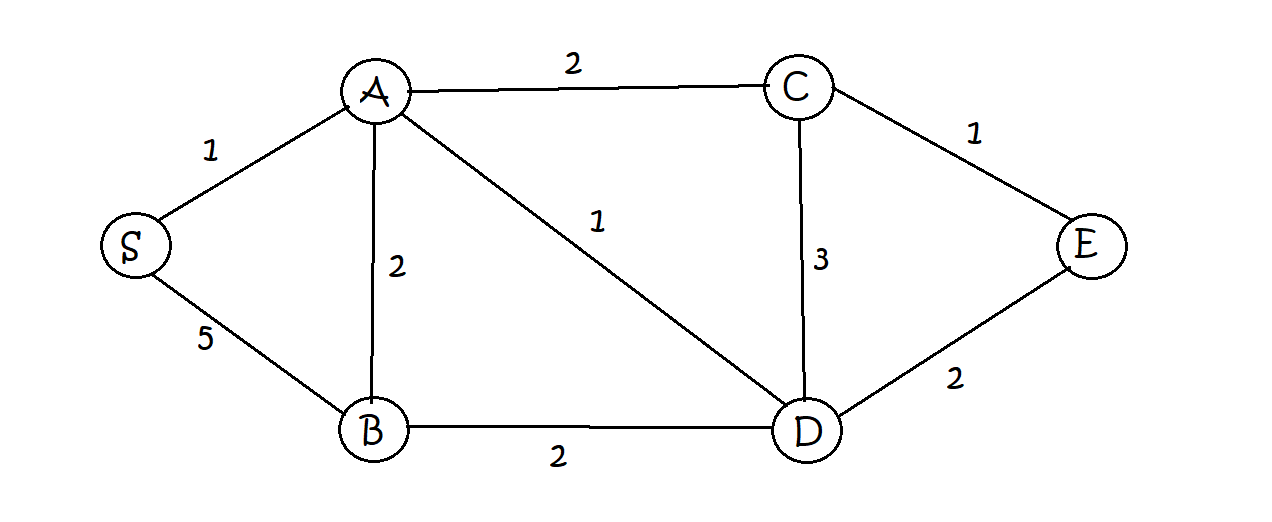
*Methods:*

|  |  |
| --- | --- |
| **Methods** | **Details** |
| addVertex(String s) | **Description:**  Adds a new vertex to the graph. It checks if the vertex already exists, and if not, it adds the vertex to the vertex list, updates the vertex-to-index map, and expands the adjacency matrix by adding a new row and column with initial weights of 0.  **Time Complexity:**  O(V). The method requires iterating over the vertex list and updating the adjacency matrix, both of which have a maximum of V elements.  **Space Complexity:**  O(V^2). The space complexity is dominated by the size of the adjacency matrix, which has V rows and V columns. |
| removeVertex(String s) | **Description:**  Removes a vertex from the graph. It checks if the vertex exists, and if so, it removes the corresponding row and column from the adjacency matrix, updates the vertex list and vertex-to-index map, and adjusts the indices in the vertex-to-index map for the remaining vertices.  **Time Complexity:**  O(V). The method involves removing a row and column from the adjacency matrix, iterating over the vertex list, and updating the vertex-to-index map, and adjusting the indices in the vertex-to-index map.  **Space Complexity:**  O(1). The space complexity remains constant as it does not depend on the number of vertices. |
| addEdge(String v, String w, int x) | **Description:**  Adds an edge between two vertices with a specified weight. It checks if the vertices exist and updates the corresponding entry in the adjacency matrix.  **Time Complexity:**  O(1). The method performs constant-time operations of retrieving indices and updating matrix entries.  **Space Complexity:**  O(1). The space complexity remains constant as it does not depend on the number of vertices. |
| removeEdge(String v, String w) | **Description:**  Removes an edge between two vertices. It checks if the vertices exist and updates the corresponding entry in the adjacency matrix to 0.  **Time Complexity:**  O(1). The method performs constant-time operations of retrieving indices and updating matrix entries.  **Space Complexity:**  O(1). The space complexity remains constant as it does not depend on the number of vertices. |
| displayAdj() | **Description:**  Displays the adjacency matrix representing the graph. It iterates over the adjacency matrix to print the vertices and their corresponding weights.  **Time Complexity:**  O(V^2). The method iterates over the [V x V] adjacency matrix.  **Space Complexity**:  O(1). The space complexity remains constant as it does not depend on the number of vertices. |
| displayEdges() | **Description:**  Displays all valid edges in the graph. It iterates over the adjacency matrix to find non-zero weights and prints the corresponding vertices.  **Time Complexity:**  O(V^2). The method iterates over the [V x V] adjacency matrix.  **Space Complexity:**  O(1). The space complexity remains constant as it does not depend on the number of vertices. |
| BFS(String v) | **Description:**  The BFS method performs a breadth-first search traversal starting from a given vertex. It visits all vertices reachable from the starting vertex in a level-by-level manner.  **Time Complexity:**  O(V^2). The nested loops in the method iterate over the adjacency matrix, which has a size of V^2. For each vertex, the method examines all other vertices to determine adjacency.  **Space Complexity:**  O(V). It uses a boolean array isVisited to track visited vertices, which requires O(V) space. Additionally, a queue is used to store vertices during traversal, which can hold at most V elements at a given time. |
| DFS(String v) | **Description:**  The DFS method performs a depth-first search traversal starting from a given vertex. It explores as far as possible along each branch before backtracking.  **Time Complexity:**  O(V^2). Like the BFS method, the nested loops in DFS iterate over the adjacency matrix, which has a size of V^2. The method visits all connected vertices.  **Space Complexity:**  O(V). It uses a boolean array isVisited to track visited vertices, requiring O(V) space. Additionally, a stack is used to store vertices during traversal, which can hold at most V elements at a given time. |
| flightAvailable(String s, String d) | **Description:**  The flightAvailable method checks if there is a flight available between two vertices, either directly or through a path of connected vertices, by performing a depth-first search.  **Time Complexity:**  O(V^2). It uses a depth-first search approach to traverse the graph, which involves visiting all vertices and checking adjacency in the adjacency matrix.  **Space Complexity:**  O(V). It uses a boolean array isVisited to track visited vertices, requiring O(V) space. Additionally, during the depth-first search, a recursive call stack is utilized, which can reach a maximum depth of V, resulting in O(V) space usage. |

1. **GraphMain Class**

GraphMain class has the main method which controls the application.

**Graph 1:**

****

**Build Graph Input:**

1

S

1

A

1

B

1

C

1

D

1

E

3

S A 1

3

S B 5

3

A B 2

3

A C 2

3

A D 1

3

B D 2

3

C D 3

3

C E 1

3

D E 2

**Post Build:**

|  |  |
| --- | --- |
| **Input** | **Output** |
| 5 | S A B C D E  S 0 1 5 0 0 0  A 1 0 2 2 1 0  B 5 2 0 0 2 0  C 0 2 0 0 3 1  D 0 1 2 3 0 2  E 0 0 0 1 2 0 |
| 6 | S->A  S->B  A->S  A->B  A->C  A->D  B->S  B->A  B->D  C->A  C->D  C->E  D->A  D->B  D->C  D->E  E->C  E->D |
| 7  B | BFS : [B, S, A, D, C, E] |
| 8  B | DFS : [B, S, A, C, D, E] |
| 9  S E | Flight price is: 9 |
| 2  A |  |
| 9  E S | Flight price is: 11 |
| 4  D E |  |
| 9  S E | Flight price is: 11 |
| 4  C E |  |
| 9  S E | No flight available from S to E |