## The data structure model

### Graph representation

* 1. We have selected “graph” as the data structure to represent given connections between the airports and flights since it is the optimal data structure to represent this kind of non-hierarchical interconnection between nodes. The graph is implemented using an adjacency matrix, with rows relating to the cargo flights and columns relating to the airports. A connection between a flight and an airport will be denoted as “1” in the relevant entry in the matrix. All the other entries will be set to “0” (i.e.: No connection) as default value. There are mainly two ways to represent matrix in python, using nested list or “numpy”. Since we are not allowed to use “numpy”, here a 2-D list is used (list of lists) to implement the adjacency matrix. Row and column indices will be used as flight and airport ids respectively. The variable name of the adjacency matrix in the code is “*adj\_matrix*”. Alternatively an “adjacency list” also could be used to represent the graph which will be more efficient in terms of memory usage. However the use of adjacency matrix makes the implementation and the time complexity of each operation effective and optimal.

1. There are two other lists (“*flights\_list*” and “*airports\_list*”) used to keep the lists and ids of the flights and the airports. These 2 lists are mainly used to lookup for the ids of the flights and airports. However using dictionaries would be more efficient (O(1) lookup time complexity) rather than using lists(O(n) lookup time complexity). Since dictionaries aren’t allowed, we have used lists here.

#### Sample Input:

Indigo666 / Chennai / New Delhi

Indigo777 / Calcutta / New Delhi

Spicejet222 / Ahmedabad / Nagpur / Mumbai

AirIndia111 / Ahmedabad / New Delhi

Vistara555 / Vishakhapatnam / Hyderabad

#### Adjacency Matrix

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Flight ID | Chennai | New Delhi | Calcutta | Ahmedabad | Nagpur | Mumbai | Vishakhapatnam | Hyderabad |
| Airport ID |  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| Indigo666 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| Indigo777 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| Spicejet222 | 2 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| AirIndia111 | 3 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| Vistara555 | 4 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |

#### Graph

Indigo666

Spicejet222

Vistara555

AirIndia111

Indigo777

#### Flight and Airport Lists

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Airport | Chennai | New Delhi | Calcutta | Ahmedabad | Nagpur | Mumbai | Vishakhapatnam | Hyderabad |
| ID | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| Flight | Indigo666 | Indigo777 | Spicejet222 | AirIndia111 | Vistara555 |
| ID | 0 | 1 | 2 | 3 | 4 |

## Details of each operations

### def readAirportFlightfile(self, inputfile):

Here we are populating the 2 lists and the adjacency matrix (mentioned above) while reading the input file. The adjacency matrix cannot be created on the go while reading the input file line by line since we don’t know how many flights and airports (which will be dimensions of the matrix) will be there until all the input is read. Hence, we populate temporary 2-D list named “graph\_data” which will hold the connections between flights and airports until the adjacency matrix is created.

Time complexity of the “readAirportFlightfile” function would be O(n3) since while populating adjacency matrix we are using 2 nested loops(O(n2)) (outer loop for each flight and inner loop for each connected airport of a flight) and a list index search (O(n)) inside the inner loop.

### def showAll(self):

Here we print the content in the two lists (“*flights\_list*” and “*airports\_list*”) to the output file. The time complexity is O(n).

### def displayHubAirport(self):

Time complexity: O(n2)

### def displayConnectedAirports(self, flight):

Check the row related to the given flight in the matrix and print the airports with connections marked as “1”.

Time complexity: O(n)

### def displayDirectFlight(self, airport\_a, airport\_b):

Both airports should be connected to the same flight for them to be connected. So, there should be “1”s in both columns related to the two airports, in a same row.

Time complexity: O(n)

### def findServiceAvailable(self, airport\_a, airport\_b):

In this function we have used recursive approach to find the route from airport\_a to airport\_b. Two lists (“visited\_airports” and “path”) have been used keep track of the visited airports and the route while traversing the graph.

Algorithm:

Case 1: Source airport is already visited. If True return None.

Case 2: Direct flight exists between airport\_a to airport\_b. If True return “True” and prepend the airport\_b and connected flight to the “path” list.

Case 3: No direct flight exists between airport\_a to airport\_b. Then first mark the “airport\_a” as visited in “visited\_airports” list and get all the directly connected airports of “airport\_a” and call the function recursively with each of those airports as “airport\_a” while keeping the “airport\_b” the same through all the recursion calls.

Time complexity would be O(n3) since 2 nested for loops (O(n2)) are used to do recursion call for each directly connected airport of source airport (airport\_a) and maximum no: of recursion calls would be n (no: of airports) (Same airport won’t be visited again)

### Helper functions

#### 7.1 def displayConnectedFlights(self, airport):

Check the particular column related to the given airport in the matrix and print the flights with connections marked as “1”.

Time complexity: O(n)

#### 7.2 def get\_list\_item\_id(self, list\_name, item\_name):

Return the index of the item in the given list

Time complexity: O(n)

#### 7.3 def write\_to\_file(self, message):

This is the helper function that writes a given string into the output file "outputPS15.txt"

#### 7.4 def print\_matrix(self, matrix):

Helper function to print the content of a given matrix. Only used for debugging purpose.