**Homework4**

**Input to the algorithm:**

* A **TXT file** **Flight\_Data** is given as input to the code in the following format:

[i] **Source** [ii] **Destination** [iii] **Arrival Time** [iv] **Departure Time** [v] **Flight Type**

* **Arrival and Departure time is in the 24 hrs format.**
* **Source and Destination are taken according to the airport abbreviations.**

*For Example: LAX for Los Angeles and BOS for Boston*

* **Capacity is taken according to the specified flight type in the code.**

*For Example: 185 capacity for A321 aircraft*

* **Python Code is coded and executed on the pycharm 2019.2.5 (community edition) platform by placing the Graph.py and Flight\_Data.txt in the same project folder.**
* **Installation of NumPy library\*\* needs to be done on the pycharm platform for the correct execution of the code.**
* **TXT file must be placed at the same place as the source code for the proper extraction of the specified data from the TXT file.**
* **TXT file path must be specified in the input\_text\_file\_path column at the bottom of the code.**
* **Similarly, temp file path, which is the same as the path of the source code must be specified in the temp\_file\_path at the bottom of the code.**

\*\* from collections import defaultdict

    import numpy as np

**Pseudo Code:**

1. Import NumPy Library
2. Define a Graph class
3. Define a function which gives the residual graph
4. Define a function which does breadth first search with source, sink and parent path(to store the residual path) as the input.
5. def BreadthFirstSearch(selfdirect, s, t, path\_parent):

# specify all the nodes as not visited

visited = [False] \* (selfdirect.ROW)

# Create a queue for BreadthFirstSearch

queue = []

# specify the source node as visited and enqueue it

queue.append(s)

visited[s] = True

# Standard BreadthFirstSearch Loop

while queue:

# Dequeue a vertex from queue and print the output

u = queue.pop(0)

# Get all adjacent vertices of the dequeued vertex u

# If a adjacent has not been visited, then specify it

# visited and enqueue it

for ind, val in enumerate(selfdirect.graph[u]):

if visited[ind] == False and val > 0:

queue.append(ind)

visited[ind] = True

path\_parent[ind] = u

# If we reach sink in BreadthFirstSearch starting from source, then return

# true, else false

return True if visited[t] else False

# Returns tne maximum flow from s to t in the given graph

1. Define a ford Fulkerson function which uses the above function breadth first search to compute the maximum flow.
2. def FordFulkerson(selfdirect, source, sink):

# This array is filled by BreadthFirstSearch and to store path

path\_parent = [-1] \* (selfdirect.ROW)

max\_flow = 0 # There is no flow initially

# Keep augmenting the flow while there is path from source to sink

while selfdirect.BreadthFirstSearch(source, sink, path\_parent):

# Find the least residual capacity of the edges along the

# path filled by BreadthFirstSearch. Or we can say find the maximum flow

# through the path found.

path\_flow = float("Inf")

s = sink

while (s != source):

path\_flow = min(path\_flow, selfdirect.graph[path\_parent[s]][s])

s = path\_parent[s]

# Add path flow to overall flow

max\_flow += path\_flow

# update residual capacities of the edges and reverse edges

# along the path

v = sink

while (v != source):

u = path\_parent[v]

selfdirect.graph[u][v] -= path\_flow

selfdirect.graph[v][u] += path\_flow

v = path\_parent[v]

return max\_flow

1. Define a function to change the time zone of the Phoenix, Denver, Atlanta, Washington, Boston and Chicago airports to Los Angeles airport.
2. Specify the airports and aircraft capacities and read the input text file/input data to compute the max flow per hour by calling the Ford Fulkerson function in the main function.
3. Print the max flow for the day.

**Time Complexity:**

The above algorithm is one of the implementations of Ford Fulkerson algorithm, commonly known as Edmonds-Karp algorithm. This algorithm uses breadth first search in the ford Fulkerson algorithm to pick the path with minimum number of edges. Vertices/Nodes are represented as V and edges are represented as E. So the final time complexity of the algorithm is:

**O(VE2)**

**Output:**

Maximum Passengers that can travel from LAX to JFK on 6th January 2020: **5829**