RUTGERS

DATA STRUCTURES AND ALGORITHMS

PROJECT REPORT

ON

FLIGHT MANAGEMENT SYSTEM

BY

KARTIK RATTAN

&

MAREESH K ISSAR

23rd APRIL 2019

TABLE OF CONTENTS

ABSTRACT

Our enthusiasm for pursuing the Flight Management System problem stems from the application of graph theory to current and future challenges in the field of science and computer technology. This research topic is strongly aligned with our area of interest and this is our primary motivation to pursue this problem.

For our analysis, we are considering a hypothetical world where the cost associated with each travel, flying from a source airport to a destination airport, depends only on the number of stoppage points or hops. Thus, to minimize the cost, we need to use graph-based algorithms to find the shortest path between two nodes of the graph. We evaluate shortest path for directed and weighted graphs using three algorithms- Dijkstra's, Greedy Best First search and A* search. We conclude that given an optimum heuristic function, A* search out performs both Dijkstra's and Greedy Best First search.

INTRODUCTION

In today's highly connected world, understanding interactions amongst various connected entities is of prime importance. A graph is a collection of vertices (nodes) that are connected by edges. The edges represent the relationship between connected vertices. Fig.1 represents various types of edges that are found in graphs.

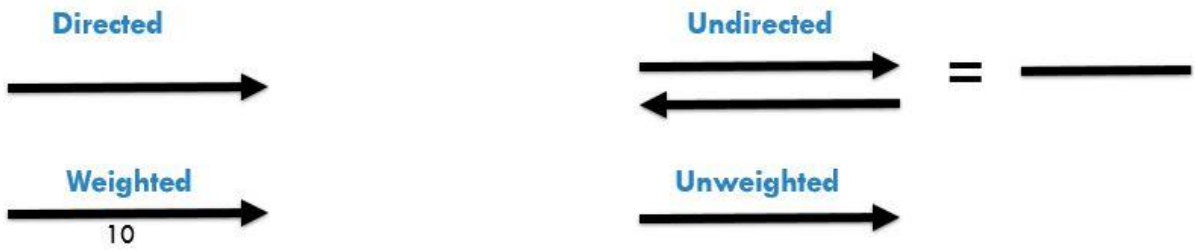

Fig.1 Types of edges.

Graph processing algorithms are powerful tools to understand a network and can be utilised to evaluate multiple models where vertices represent molecules, people, and transportation systems. The edges might represent chemical bond present between those molecules, relationship between people in a network and routes between transportation centres. The graph algorithms we plan to study function independently of the representation of vertices/edges and help us in answering some of the fundamental questions related to them. For instance, is there a path from a given source to a destination? Is there a cycle in a graph?

In our study, we aim to discuss the existence of a shortest path and the cost of finding such a path in a fastest possible manner. We have used an adjacency list to represent them in our program since most of the graphs in real world are sparse. For all the graphs used in experimentation and analysis, nodes represent Airports, edges represent the connection between them, and the weights of these edges represents the distance between them.

Fig.2 Symbolic representation of nodes and edges of a graph

ALGORITHMS EMPLOYED

1) Breadth First Search (BFS)

The breadth first search algorithm traverses the graph in layers. It traverses all the vertices closest to the source, evaluates them and then visits the second closest set of vertices. That is, first it traverses all the 1-hop neighbours of the source node, then traverses all its 2-hop neighbours and until it reaches the destination or exhausts all the vertices.

The Pseudo code for BFS is:

BFS (vertex s)
Mark vertex s as visited. Put s into a queue q.
Repeat until q is empty {
    Remove the least recently added vertex w
    Visit all the unmarked adjacent vertices (b) of w. Mark them as visited and put them on q}

It gives us the shortest path from the source node to any another node for an unweighted graph. That is, it gives us the least number of hops counts from a source to the destination vertex.

2) Depth First Search (DFS)

The depth first search algorithm traverses the graph by exhausting the neighbours present in all the layers for a vertex and repeats this process for all the adjacent neighbours of the source. A key property of this algorithm is that it does not necessarily return the shortest path.

The Pseudo code for DFS is:

DFS (vertex s)
    Mark the vertex s as visited.
    Recursively visit all the unmarked vertices (b) that are adjacent to the vertex s.

The algorithms studied above were discussed in the context of unweighted graphs. Unweighted edges essentially neglect the effort spent in traversing between two nodes. Therefore, adding weights to the edges helps us in quantitatively deciding the path (shortest path) to select in case there are multiple options from source to destination, having equal number of vertices between them.

Thus, we now analyze the algorithms that find the shortest path from source to destination for weighted and directed graphs.

3) Dijkstra's Algorithm

Dijkstra's algorithm computes the shortest path tree from the source vertex to all the vertices in the graph based on their distance from source and edge weights (non-negative).

The Pseudo code for Dijkstra is:

Dijkstra (vertex s)

Consider the vertices in increasing order of distance from source vertex s
(Non-tree vertex with lowest cost value)
Add vertex to the tree and relax all edges pointing from that vertex

It evaluates all the connected vertices and then selects only the edge with the smallest cost. The cost function used while selecting the edge is given by:

Current cost of node = cost of reaching the previous node + edge weight

A major drawback of this algorithm (as shown in Fig.3) is that it may start searching for the shortest path in the wrong direction. Since the search is based only on the weight of an edge, it may traverse along the vertices which might not lead to the destination but has the least cost (as shown by path with edge weight of 5 at the deciding node). The destination node in this case can be reached faster if the algorithm selects the path with edge cost of 8 at the deciding node.

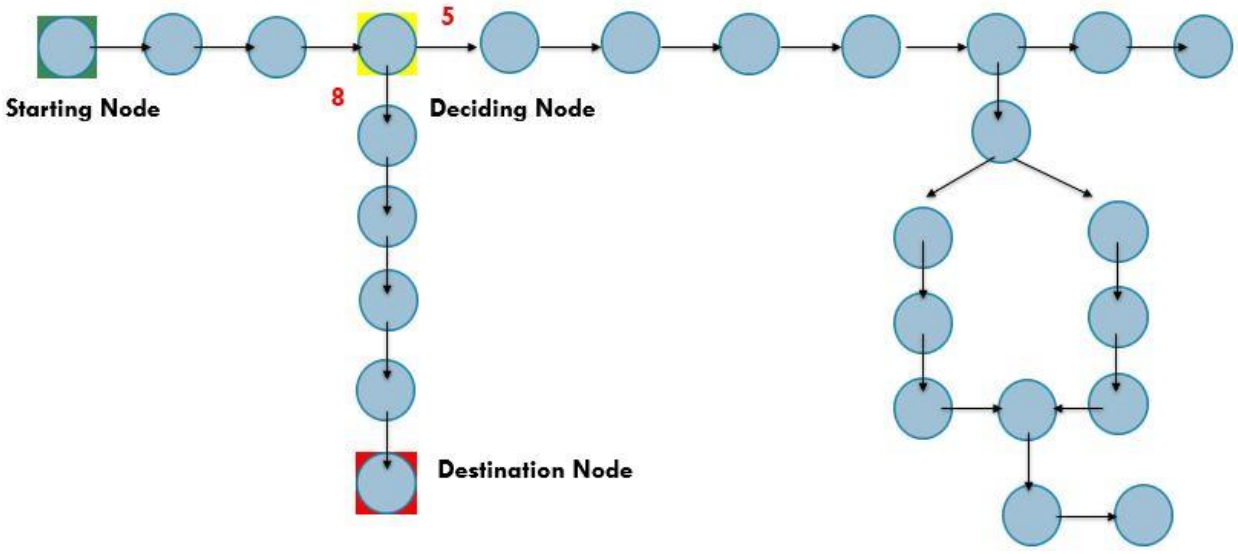

Fig.3 Shortcoming of Dijkstra's algorithm.

4) Greedy Best First Search

As we saw, an absence of a guiding sign board leads to Dijkstra's shortcoming in some situations. To overcome this, we employ another algorithm, known as Greedy Best First algorithm. It uses a "heuristic" to select the best possible next node in each step. Heuristic is a guiding principle (an estimate of the actual distance) that helps us in deciding whether we are moving in the right direction or not. It essentially behaves as a signboard which directs us where to go next.

Fig.4 Heuristic to guide us in the right direction

The Pseudo code for GBF is:

GBF (vertex s)

> ***Consider the vertices in increasing order of heuristic value from source vertex s (Vertex with lowest heuristic value that is not yet added to the shortest path)***
>
> ***Add vertex to the path and relax all edges pointing from that vertex***

This leads to us to an important question – Is Greedy Best First search the best algorithm for finding the shortest distance?

The answer is **No**.

Greedy Best First search algorithm can be easily tricked and thus does not necessarily gives us the shortest path. It can only prevent us from wandering in the wrong direction.

For example, as shown in Fig.5, the algorithm is not capable of differentiating between two paths with the same direction towards the destination (i.e. same heuristic value) but having different weights. It will try to reach the destination

faster by taking a path similar to the optimum path but not necessarily the optimum one.

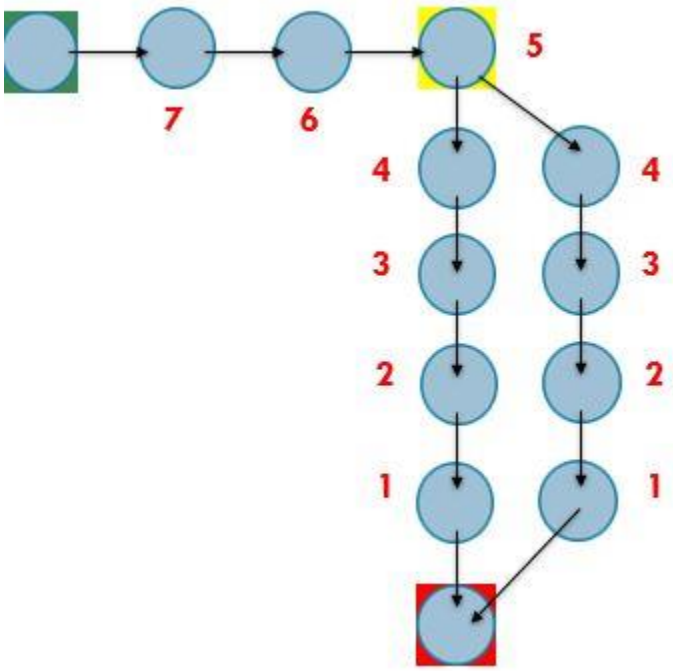

Fig.5 Shortcoming of Greedy Best First algorithm.

5) A* Search Algorithm

This algorithm essentially gives us best of the both algorithms (Dijkstra's and Greedy Best First). It combines the shortest path approach taken by Dijkstra's algorithm with the heuristic based directional movement approach from the Greedy Best First algorithm to provide us the shortest path between two nodes in the graph in the fastest possible time. The A* search algorithm was given in 1968 by Peter Hart, Nils Nilsson and Bertram Raphael.

The cost function used while selecting the edge is given by:
Current cost of node = cost of reaching the previous node + edge weight + heuristic cost (estimated distance to target)

The Pseudo code for A* is:
A*(vertex s)

> ***Consider the vertices in increasing order of cost value from source vertex s (Vertex with lowest cost value that is not yet added to the shortest path)***
> ***Add vertex to the path and relax all edges pointing from that vertex***

A key property of this algorithm is that it gives us the shortest path if heuristic(node) ≤ actual distance from node to target.

Fig.6 depicts the working of the algorithm. The algorithm is deployed in this small graph to find the shortest path from the source node (shown in green) to the destination node (shown in red). Node with heuristic value 50 is disregarded as there are two other nodes with smaller heuristic values. Amongst the two nodes with same heuristic of 7, the longer path is rejected based on the Dijkstra's algorithm.

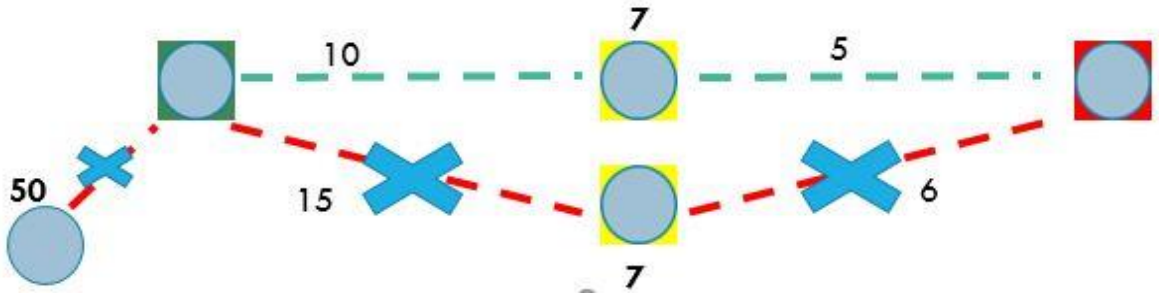

Fig.6 Working of A* search algorithm.

To completely understand the working of the algorithm, consider the network topology shown in Fig.7

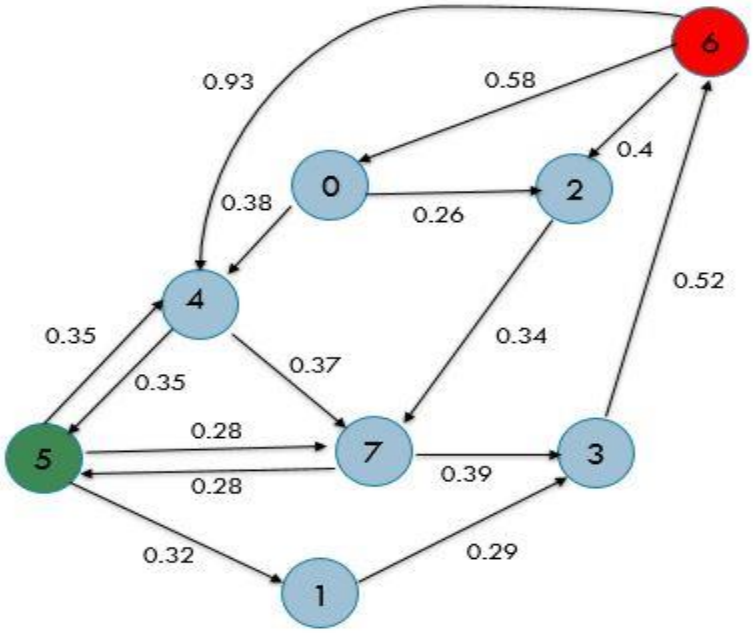

Fig.7 Network topology to understand the working of A* search algorithm.

Adjacency list for 5:

5	5->4	5->7	5->1

When Dijkstra's algorithm runs on the network topology given in Fig.7, it will start by selecting the edge 5->4 as it is the edge present first in the adjacency list. Though the weight for the edges 5-7 and 5-1 will push the edge 5-4 down in the queue, it will still evaluate paths from the vertex 4 after exhausting 7 and 1. Thus, it will end up evaluating all the possible paths. It selects the shortest path based on sum of weights at the end, but at the expense of high computation.

When Greedy Best First algorithm runs on the network topology given in Fig.6, it will disregard the 5->4 edge as it has higher heuristic value (3) as compared to the nodes 7 and 1 (both have heuristic value of 2). But the algorithm can select either 7 or 2 in its shortest path evaluation depending on which vertex is processed first and thus fails to distinguish between them. Thus, in our evaluation, it ends up giving a ~optimal path (5-7-3-6) faster than Dijkstra's.

For A* search algorithm, we require a heuristic to help us in our search. The process for heuristic generation for the topology in Fig.7 is shown in Fig.8

1) Reverse all the directed edges present in the graph.

2) Run BFS on this graph to compute and assign heuristic to each node based on its hop count from the destination node.

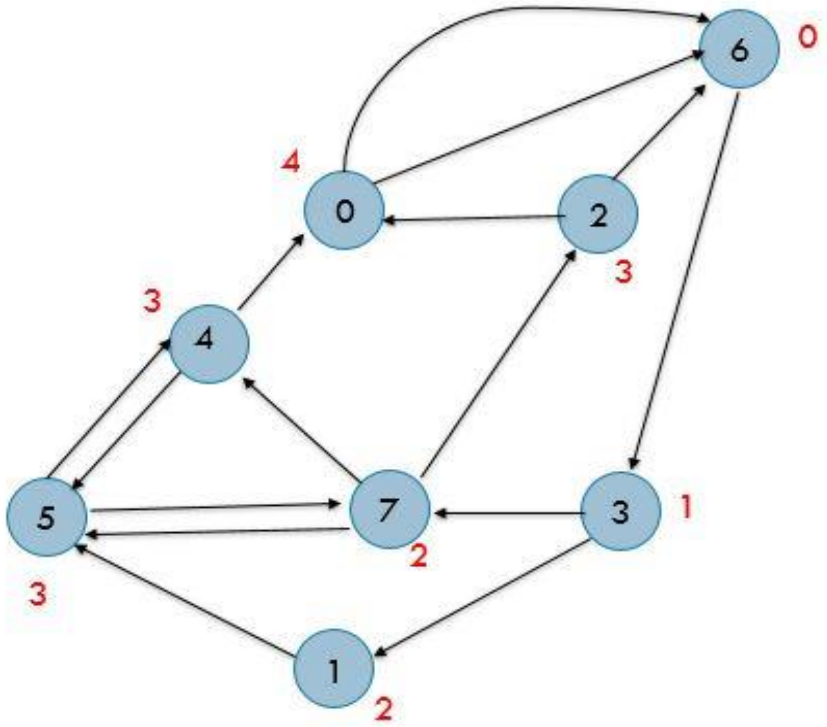

Fig.8 Network topology along with generated heuristics.

When A* search algorithm runs on the network topology given in Fig7, it disregards the vertex 4 because of high heuristic value. It then evaluates vertices 7 and 1. Given the two vertices, it will select the edge 5->1 as it leads us to the shortest path from source to the destination node.

A* Search and Its Shortcomings

The important difference between A*, Greedy Best-First search and Dijkstra's is that A* combines the value given by the heuristic function with the weight of an edge along the shortest known path to the destination vertex under consideration.

A* maintains a priority queue of options that are to be considered, ordered by the feasibility of those paths in term of direction and distance. It keeps searching for an optimal path until it finds a route to the destination, and none of the other options could possibly make it better.

But finding the optimal path depends upon an optimal heuristic function for the graph. This means that the heuristics defined should accurately estimate the distance between the vertices, exactly as the they are supposed to be if they represent a real-life situation. For instance, let us assume that the distance between city A and city B is 5 miles and the distance between city C and city D is 10 miles. The heuristic function should be able to estimate these distances accurately, giving a heuristic value of 5K for A-B and 10K for C-D, where K is a constant.

Inaccurate heuristics can either underestimate or overestimate a path(edge). Giving a low heuristic value to the vertex which is far away from destination will lead to underestimating it. Giving a high heuristic value to a vertex which is close to the destination will lead to overestimating the vertex.

If the heuristic underestimates, the distinction between the optimal and non-optimal path reduces. The other options will look better than they actually are. A* search is forced to believe that those other options might improve the path, so it computes those options and evaluates them. If the heuristic only underestimates by a little bit, maybe some of those routes will turn out to be useful. This will lead to more computation and decrease the key quality of A* search- distinguishing between the optimal and sub-optimal paths. Since those paths are sub-optimal, the weights after some computations will guide the A* algorithm to check for other

paths. In the end, A* will return an optimal path. But will fail in providing the shortest path in minimum time.

On the other hand, if the heuristic overestimates, A* can think that the alternatives to the route it already has are all worse, so it won't evaluate them. But the heuristic overestimates so they might be much better than they are. Thus, A* search will find the first path to the destination and provide that as the shortest path.

For instance, suppose we are trying to drive from Chicago to New York and our heuristic function is what our friends think about geography. If your first friend claims that Boston is close to New York (underestimating), then we will look for routes though Boston. After a considerable duration, we realise that any optimal route from Chicago to Boston already gets fairly close to New York before reaching Boston and that actually going via Boston just adds more computation. Thus, we do not consider routes via Boston. We continue to find the optimal route. Underestimating the route cost us a bit of time but, in the end, we will find the right route.

Suppose we consider the other extreme of the heuristic: Indiana is 100,000 miles from New York. Nowhere else on earth is more than 13,000 miles from New York. Thus, if we take that heuristic as a guiding function, we won't even consider any route through Indiana. This makes out duration for nearly twice as long as we move around the state, covering 50% more distance.

Dijkstra's, on the other hand, guarantees an optimal path irrespective of any other consideration. Thus, if the optimality of the path between the two vertices is of utmost importance, Dijkstra's perform better than A*.

EXPERIMENTAL CONFIGURATION AND DETAILS

A* is the quickest of the three algorithms discussed to compute the shortest path between two vertices, apart from the situations where the heuristics are sub-optimal.
To quantitatively establish this statement, as well as check the scalability and the performance of the above-mentioned algorithms, graphs with increasing number of vertices and edges were analysed. We also implement the above algorithms on existing data set taken from http://konect.uni-koblenz.de/networks/ website. The dataset from the website consists of 1000 vertices representing airports and edges define a path between them.
We computed the shortest path between a source vertex (5) and a destination vertex (6) for the following configurations:

FILE NAME	NO. OF VERTICES	NO. OF EDGES
8EWD.txt	8	15
250EWD.txt	250	2546
Airport_data.txt	1000	16,866
10000EWD.txt	10000	123,462

Table.1 Experimental configurations.

The algorithms were studied on a 64-Bit Operating system, 2.50GHz i-7 processor with a 16GB RAM. The code was written in C++.

RESULT AND ANALYSIS

1) Number of vertices = 8, Number of edges = 15

Graphs where the number of vertices and the edges are not large, the computational cost and the time (in microseconds) does not differ significantly. But we can still observe that even for smaller number of vertices, the A* search is quantitatively better than Dijkstra's and Greedy Best First.

8EWD.txt			
ALGORITHM	COMPUTATION COST	SP LENGTH	TIME (in microseconds Log_{10} scale)
Dijkstra's	10	1.13	2.326
Greedy Best First	9	1.19	2.279
A* Search	7	1.13	2.265

Table.2 Results for file 8EWD.txt

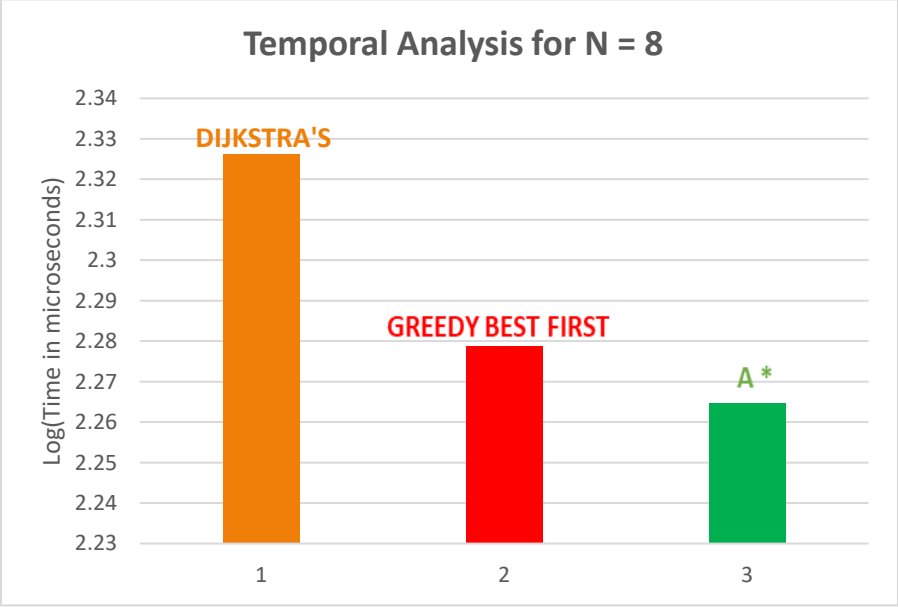

Figure.9 Temporal analysis for file 8EWD.txt.

2) Number of vertices = 250, Number of edges = 2546

As the number of vertices and edges increase, we observe that the performance of A* search, both computational and temporal, is significantly better than Greedy Best First and Dijkstra's.

250EWD.txt			
ALGORITHM	COMPUTATION COST	SP LENGTH	TIME (in microseconds Log_{10} scale)
Dijkstra's	15960	0.71	5.399
Greedy Best First	9866	0.72	5.225
A* Search	25	0.71	2.647

Table.3 Results for file 250EWD.txt.

An increase in the number of vertices and edges leads to an increase in the number of options that may lead to an optimal path. In other words, the number of paths a system must analyze before finding the best one increases. This leads to an increase computational and temporal cost.

Dijkstra's as discussed before, will analyze all the paths and then give out the best one. Thus, it takes the maximum time to converge to an optimal path. Greedy Best First is more efficient than Dijkstra's as it eliminates all the paths that are not in the direction of the destination vertex. But as the graph becomes denser with an increase in edges, so does the number of paths in the direction of our destination. Hence, it takes lesser time and computational effort than Dijkstra's, but is still slower than A*.

A*, apart from eliminating all paths not in the direction of the destination, computes the shortest path between every vertex till we reach the destination. Only the paths which are both heuristically correct and return the shortest weight sum for an edge are considered. The graph below depicts a decrease in the time (in microseconds) for the A* search. This decrease is much significant with respect to graphs with lower number of edges.

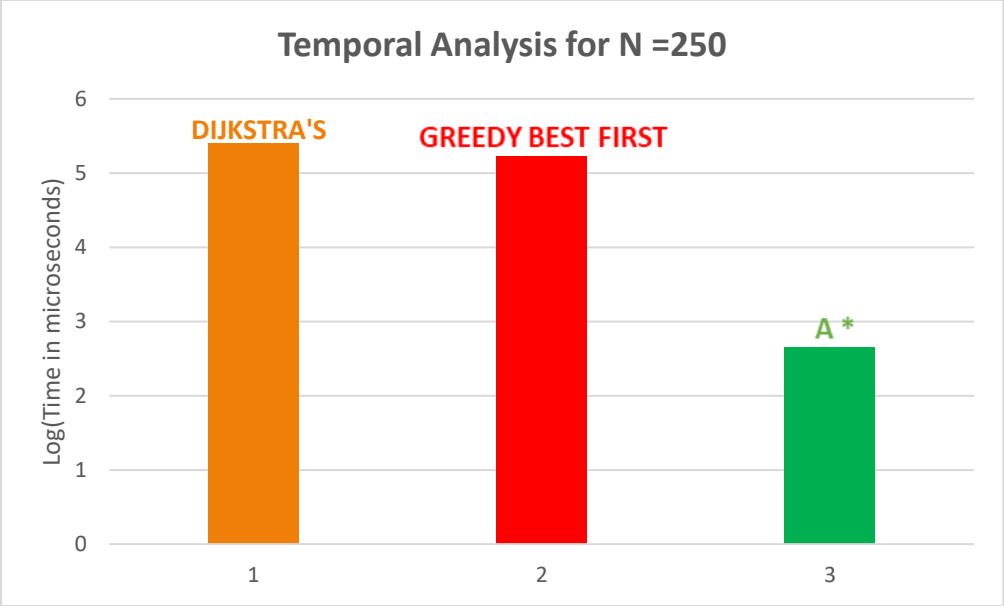

Fig.10 Temporal analysis for file 250EWD.txt.

3) Number of vertices = 1000, Number of edges = 16866

We can see a similar trend for the airport data. As the number of edges increase, the graph becomes denser. This leads to an increase in possible options for both Dijkstra's and Greedy Best First search. A*, as before, computes only those paths that are in the direction of the destination vertex and produce minimum edge weight sum to each vertex.

Airport_data.txt			
ALGORITHM	COMPUTATION COST	SP LENGTH	TIME (in microseconds Log_{10} scale)
Dijkstra's	152238	0.71	6.304
Greedy Best First	90031	0.74	6.091
A* Search	1210	0.71	2.491

Table.4 Results for file Airport_data.txt.

As the graph becomes denser, we also observe that a A* search with optimum heuristics outperforms both the other algorithms significantly.

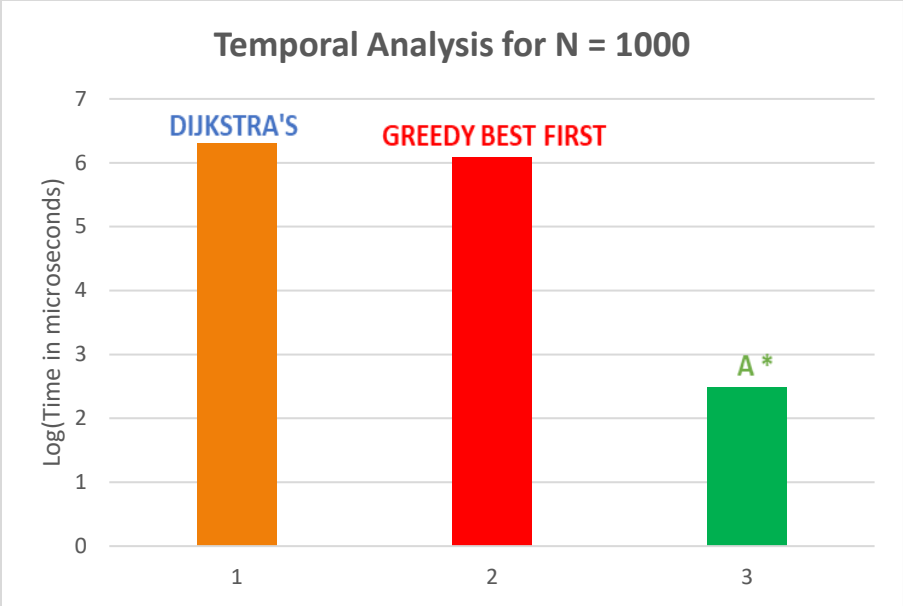

Fig.11 Temporal analysis for file Airport_data.txt.

4) Number of vertices = 10000, Number of edges = 123,462

The A* search continue to present optimal computational and temporal cost as the number of edges increase with respect to Dijkstra's and Greedy Best First search. The complexity involved with evaluating the heuristics is not as significant as compared to the increase in efficiency provided by the A* search.

10000EWD.txt			
ALGORITHM	COMPUTATION COST	SP LENGTH	TIME (in microseconds Log_{10} scale)
Dijkstra's	5941100	0.27	7.960
Greedy Best First	2261554	0.28	7.528
A* Search	162	0.27	2.553

Table.5 Results for file 10000EWD.txt.

Though Greedy Best First computes the ~ shortest path in half the cost with respect to Dijkstra's, it is significantly consuming more resources with respect to A*.

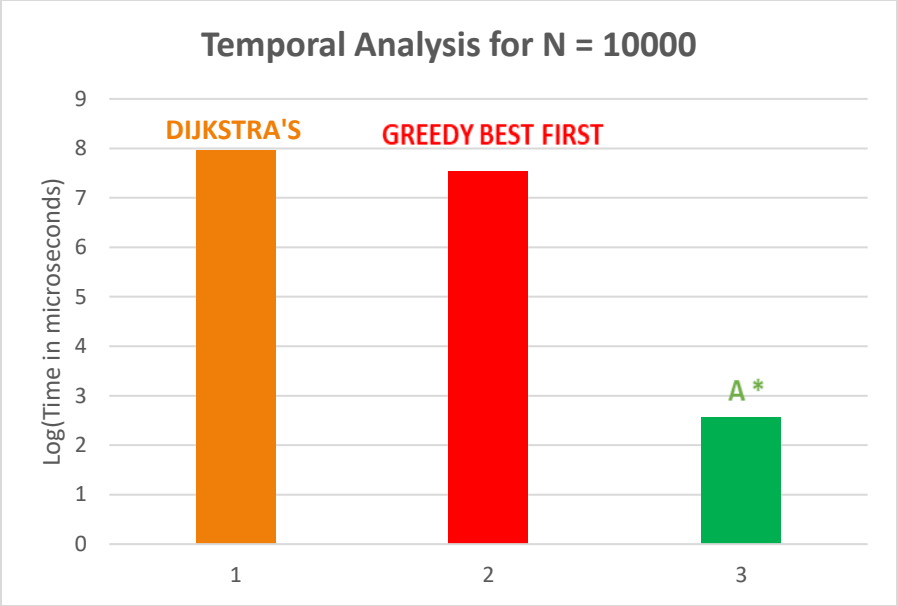

Fig.12 Temporal analysis for file 10000EWD.txt.

DISCUSSION AND CONCLUSION

This study starts by finding the answer to the question of whether there exists a connection between two airports or not? To answer this question, we employ Depth First Search algorithm. We follow this up with an attempt of finding the shortest path between two airports. This can be answered by Breadth First Search algorithm. These algorithms compute the shortest path from a source vertex to all the vertices of the unweighted graph.

Algorithm	Remark
Breadth First Search	Optimal but slow (uses queue for processing vertices)
Depth First Search	Meandering and not optimal (uses stack for processing vertices)

Table.6 Overview of the graph traversing algorithms for unweighted graphs.

The above-mentioned algorithms were discussed in the context of undirected and unweighted graphs. Since, most of the real-worlds graphs (such as representing airline connection networks) are directed and weighted, we employ the following algorithms to answer the questions discussed so far.

Dijkstra's algorithm gives us the shortest path tree from the source vertex to all the other vertices by processing the vertices at an increasing distance from the source. A major shortcoming of this algorithm is, that though it could always select the shortest path after analysing all the available options (and thus relaxing each edge), it will not lead us to the destination node in the fastest time. This led us to explore another algorithm, Greedy Best First Search. This algorithm moves towards the destination node in a streamlined manner, using heuristics (approximate distance of each node from the destination node) as its guide for selecting the subsequent nodes to traverse in order to reach the destination. Due to its focussed approach the algorithm takes less time to run than Dijkstra's algorithm but is not the best algorithm to find the shortest path in the fastest time. When faced with choice of selecting a path having two nodes of equal heuristic value, it selects the one discovered first, irrespective of the sum of edge weights. This shortcoming is overcome by A* search algorithm which traverses the graph in time much faster Dijkstra's algorithm or Greedy Best First algorithm. The A* search algorithm achieves this performance by selecting the next node to traverse based upon the edge weights and the heuristic values, thus giving an optimal path in shortest time.

Algorithm	Remark
Dijkstra's algorithm	Explores in increasing order of cost, optimal but slow
Greedy Best First Search	Proceeds fast towards the target, but can be tricked easily
A* search algorithm	Optimal and fast

Table.7 Overview of the shortest path algorithms for directed weighted graphs.

REFERENCES

[1] https://cs.stanford.edu/people/abisee/gs.pdf

[2] https://www.redblobgames.com/pathfinding/a-star/introduction.html

[3] https://algs4.cs.princeton.edu/44sp/

[4] http://konect.uni-koblenz.de/networks/