MATH 3012 Report

OPTIMIZING AIRLINE FLIGHT PATHS UTILIZING GRAPH ALGORITHMS

SHAWNAK SAMADDAR
SEAN LEE
DAVID XIBILLE
RILEY JOCHAM

Georgia Institute of Technology

November 2024

Abstract

When taking long flights via airplane, the destination is often not reached directly; layovers occur. The efficiency of taking layovers compared to taking a direct flight is a question that can be solved via graph algorithms; particularly, we use Djikstra's algorithm to find the minimum path from one airport, v_A, to another airport, v_B. This is done by treating the collection of airports as a graph, where flights between airports are edges. These edges can be assigned weights corresponding to costs, such as time, monetary cost, and efficiency; using Djikstra's algorithm will allow us to find a path that minimizes these quantities. Additionally, we compare the practicality of other algorithms, such as BFS (Breadth-first search) and DFS (Depth-first search). All of our programmed algorithms are in Python.

Contents

1 Introduction

While planning flights, people often have different preferences depending on their situations. While some may simply desire the fastest route, others may want the cheapest or the most convenient, such as those with the least layovers. Algorithms can be used to find efficient paths that satisfy certain criteria. However, not all algorithms accomplish the task in the same way, as each uses a different approach. Our project explores the usefulness of several algorithms, such as Breadth-first search, DFS, and Dijkstra's algorithm in finding the most efficient path from one location to another under different strategies. Additionally, we define a modified version of Djikstra's Algorithm to optimize efficiency. We will be utilizing flight data from the US, although our results can be extended to other countries or even to an international scale. The findings of this project could not only be used for individuals trying to find flight paths, but also by airlines to analyze customers' tendencies to choose certain routes.

We learned about our methods from a variety of sources, including *Graph Algorithms* [2], *Algorithms and Data Structures: The Basic Toolbox*, [4], *Applied Combinatorics* [5], and the online article *Graph Traversal in Python: Breadth First Search (BFS)* [1].

2 Methodology

2.1 Data processing

The dataset *US Airline Flight Routes and Fares 1993-2024* [3] is used to obtain data for US domestic flights and their fares. The dataset is processed in Python using the Numpy and Pandas libraries. Initially, the dataset contains several extraneous parameters for flights, such as airline and market share of airline. These columns are dropped from the dataset. The data is also filtered so that so that only years after 2018 inclusive are counted. Any rows containing N/A and 0 values are also dropped due to data input error. The columns are also reordered for viewing convenience. The following code accomplishes this task:

```python
#importing numpy and pandas
import numpy as np
import pandas as pd
'''
flight_data is a Pandas DataFrame containing the flight data
'''
#dropping unneccessary columns
flight_data = flight_data.drop(['tbl', 'citymarketid_1',
    'citymarketid_2', 'airportid_1', 'airportid_2', 'large_ms',
    'Geocoded_City1', 'Geocoded_City2', 'tbl1apk', 'lf_ms']
    , axis=1)
#removing years before 2018
```

```
flight_data = flight_data[flight_data['Year']>2017]
flight_data.dropna()
flight_data
#reordering columns
cols = list(flight_data.columns.values)
cols
cols = ['Year','airport_1','airport_2','fare', 'nsmiles',
    'carrier_lg', 'fare_lg', 'carrier_low', 'fare_low','city1',
    'city2', 'passengers', 'quarter']
flight_data= flight_data[cols]
```

Once the data was analyzed, it was found that the dataset includes only one direction of a route between 2 airports. So, every flight had to be reversed and added to the original dataset. This process assumes that flights have the same cost forwards and in the reverse direction. The following code accomplishes this task:

```
#reverse all routes and add back to original data
copy = flight_data.copy()
copy1 = flight_data.copy()
col_list = list(copy1)
#swap position of airport_1 and airport_2 column
col_list[1], col_list[2] = col_list[2], col_list[1]
copy.columns = col_list
#swap airport_1 and airport_2 column headers to swap data
cols = ['Year','airport_1','airport_2','fare', 'nsmiles',
    'carrier_lg', 'fare_lg', 'carrier_low', 'fare_low',
    'city1', 'city2', 'passengers', 'quarter']
copy = copy[cols]
flight_data = pd.concat([copy,copy1])
flight_data
```

It is noted that multiple flight routes are repeated every year with different pricings. We will merge these rows by taking the means of the flight fares and distances. The following code accomplishes this task:

```
#merging rows with identical routes
flight_data= flight_data.groupby(['airport_1', 'airport_2'])
.agg({'fare': 'mean', 'nsmiles':'mean', 'carrier_lg': 'first',
    'fare_lg': 'mean', 'carrier_low': 'first',
    'fare_low': 'mean','city1': 'first', 'city2': 'first'})
    .reset_index()
```

2.2 Converting Data to Graph

Now, the data is properly formatted and ready to be transformed into a graph. Create an adjancency dictionary for the airports. The following code creates the dictionary:

```python
#creating graph
graph_dict = {}
#iterate through every route
for index, row in flight_data.iterrows():
  #add airport_2 to adjacency of airport_1
  if row['airport_1'] not in graph_dict.keys():
    graph_dict[row['airport_1']] = [row['airport_2']]
  else:
    graph_dict[row['airport_1']].append(row['airport_2'])
graph_dict
```

Additionally, create a weighted graph for Djikstra's algorithm to run on. The following code accomplishes that:

```python
#creating weighted graph, same as before with weights
weighted_graph_dict = {}
for index, row in flight_data.iterrows():
  if row['airport_1'] not in weighted_graph_dict.keys():
    #add airport_2 and weight to adjacency of airport_1
    weighted_graph_dict[row['airport_1']] = {row['airport_2']
        : row['fare']}
  else:
    weighted_graph_dict[row['airport_1']][row['airport_2']]
        = row['fare']
weighted_graph_dict
```

Lastly, we need to create another graph that is identical to the weighted graph but with an added cost constant added to each weight. The use of this graph will be discussed later in the paper. The following code accomplishes the task:

```python
#creating weighted graph, with a constant added to the weights
layover_graph_dict = {}
layover_cost = 80
for index, row in flight_data.iterrows():
  if row['airport_1'] not in layover_graph_dict.keys():
    #add airport_2 and weight to adjacency of airport_1
    layover_graph_dict[row['airport_1']] = {row['airport_2'] :
        row['fare']+layover_cost}
  else:
    layover_graph_dict[row['airport_1']][row['airport_2']]
        = row['fare']+layover_cost
```

Now, the graphs are ready to run algorithms on.

3 Algorithms

3.1 Depth First Search (DFS)

One of the simplest methods to find a path from one airport to another is through a depth-first search (DFS).

3.1.1 Mechanism and Limitations

DFS works by selecting a path from the start node and exploring it fully, before backtracking to the most recent branching path and repeating. Consider the following example, where DFS will be used to find a path from A to H

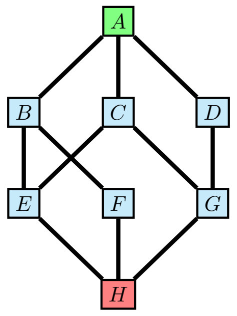

Figure 1 : DFS example graph

Every time we have to pick a neighbor in the DFS algorithm, we will pick the leftmost neighbor in the graph. Analyze how many steps it will take to find a path.

1. $A \rightarrow B$
2. $B \rightarrow E$
3. $E \rightarrow H$

DFS finds the path in the most efficient way possible with just 3 steps. Now, consider what would happen if we sought to find a path from A to D instead:

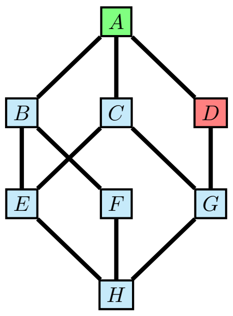

Figure 2 : Modified DFS example graph

Consider the steps needed to find the path from A to D:

1. $A \rightarrow B$
2. $B \rightarrow E$
3. $E \rightarrow H$
4. $H \rightarrow E$
5. $E \rightarrow B$
6. $B \rightarrow F$
7. $F \rightarrow H$
8. $H \rightarrow F$
9. $F \rightarrow B$
10. $B \rightarrow A$
11. $E \rightarrow H$
12. $\ldots etc$

It will take 19 steps to find the path from A to D, despite them being neighbors. This highlights one of the key limitations of DFS. Even if 2 nodes are extremely close together, DFS can take many steps to find the correct path due to it traversing paths fully. In this case, DFS still generates an optimal path from A to D. Now let's consider a case where DFS generates an nonoptimal path between 2 nodes. Use DFS to find a path between A and F in the following graph:

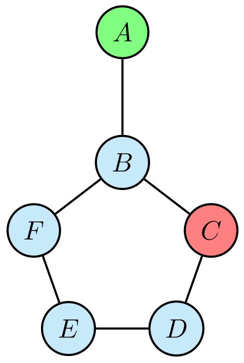

Figure 3 : DFS in a cycle

DFS will generate the path $A \to B \to F \to E \to D \to C$. However, the optimal solution is simply $A \to B \to C$. In this case, DFS both takes more steps than necessary and generates an nonoptimal path.

We can generalize the previous result to cycles. In a graph containing a cycle, finding a path to a node in the cycle often generates an nonoptimal and slow solution if DFS picks the wrong direction of the cycle to traverse along.

Using the previous generalizations, we can come up with the best use case for DFS. The ideal graph will have no cycles and the nodes will be very far apart. This fits the concept of a tree. Thus, the optimal use case for DFS is trees where the starting and ending nodes are very far apart.

3.1.2 Algorithm Pseudocode

The following steps are used to run DFS:

```
 1: Create empty stack
 2: Set all points to unvisited
 3: Add current node to stack
 4: if node is goal then
 5:     Return stack as path
 6: end if
 7: if node not visited then
 8:     Mark node as visited
 9:     if node has neighbors then
10:         Randomly shuffle neighbors                ▷ Generates unique paths
11:         for each neighbor do
12:             if neighbor node is not visited then
13:                 Run DFS on neighbor node                      ▷ Recursion
14:                 if goal node found then
15:                     Return stack as path
16:                 end if
17:             else
```

```
18:             Pop path from stack
19:           end if
20:         end for
21:       end if
22:   else
23:       Pop path from stack
24:   end if
```

This is the standard DFS algorithm used in most applications. However, one unique detail implemented in this algorithm is the random shuffling of neighbors. This ensures that when DFS traverses the array, it does so randomly and not in any order found in the data. This allows the DFS algorithm to create unique solutions, even with the same parameters. This will be of use later in the paper where a time complexity analysis of DFS is conducted.

3.1.3 Implementation

With the pseudocode, an implementation of DFS can be made in Python. Note that the deque data structure is imported and used.

```python
#import random for random path choice
import random
#import queue structure
from collections import deque
#initialize visited nodes set and stack
visited = set()
stack = deque()
def dfs(visited, graph, node, goal):
    #add current node to stack
    stack.append(node)
    #check if current node is goal, then return stack
    if node == goal:
        return stack
    #check to make sure node not already visited
    if node not in visited:
        #add node to visited set
        visited.add(node)
        #if next nodes exist, traverse to them
        if graph[node]:
          #randomly pick a neighbor to traverse to
          random.shuffle(graph[node])
          #for all neigbors, recursively run dfs
          for neighbor in graph[node]:
              if neighbor not in visited:
                path = dfs(visited, graph, neighbor, goal)
              #return stack if goal found
                if path:
                  return path
              #if no more nodes to traverse, pop path
                else:
                  stack.pop()
        #if no more nodes to traverse, pop path
        else:
          stack.pop()
    return None
```

Deque (doubly ended queue) was used in this case instead of a list. Although a list could have been used to serve as a queue, it is inefficient compared to deque. Popping from the end of a list is an O(1) operation, identical to a deque. However, popping from the start of a list is an O(N) operation, compared to O(1) for a deqeue. While the stack in DFS didn't involve popping from the start, the deque was used to maintain data structure homogeneity throughout the code since queues would be used later in the code.

3.1.4 DFS in Context

The issue with using DFS for our purposes is that more often than not, it will not find the most efficient path to a target destination, instead sticking with whatever path leading to the target it found first, regardless of the distance from the start airport. This relates to the optimal use case of DFS: trees with very distant starting and ending nodes. In the case of flights, cycles are extremely common and starting and ending airports are usually within 2 steps of each other. This goes to show that DFS will be very likely to generate far from optimal solutions.

3.2 BFS

Another commonly used search algorithm is Breadth First Search (BFS).

3.2.1 Mechanism

Unlike DFS, BFS explores multiple paths simultaneously. BFS traverses the graph by moving to vertices a certain distance from the start node. In the first iteration, BFS creates a path from the start node to each its neighbors. In every subsequent iteration, every path is extended by 1 in every possible direction. Consider the following graph, where BFS is started from A:

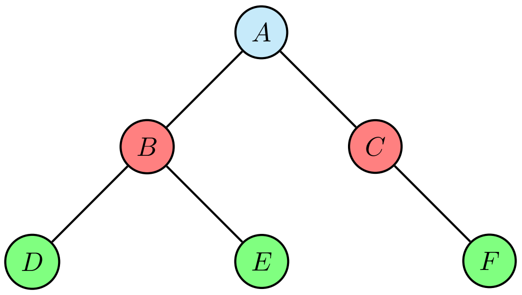

Figure 4 : BFS in a tree

The red nodes represent nodes traversed to in the first iteration of BFS. These nodes are within 1 edge traversal from the start (A). The green represent nodes traversed to in the second iteration of BFS. These nodes are 1 edge traversal away from the nodes from the first BFS iteration.

This graph highlights a key feature of BFS. When nodes are close together, they are traversed to very quickly. In this example, the red nodes, which are 1 away from the starting blue node, are reached within 1 traversal. However, nodes far from the starting node, such as the green nodes, are traversed to at the end.

3.2.2 Algorithm Pseudocode

The following steps are used to run BFS:

```
 1: Set all points to unvisited
 2: Create empty queue
 3: Add current node to queue
 4: while queue contains paths do
 5:     Remove first path from queue
 6:     Access last node of path
 7:     if node not visited then
 8:         if node is goal then
 9:             Return path
10:         end if
11:         for all neighbors of node do
12:             Add neighbor to original path and add new path to queue
13:         end for
14:     end if
15:     Return false                                  ▷ No paths found
16: end while
```

3.2.3 Implementation

With the pseudocode, an implementation of BFS can be made.

```
#import queue structure
from collections import deque
visited = set()
def bfs(start, end, graph):
  #create queue with starting node
  queue = deque()
  queue.append([start])
  #iterate while queue contains elements
  while queue:
    #take out last explored path from queue
    path = queue.popleft()
    node = path[-1]
    if node not in visited:
      visited.add(node)
      #check if the path ends with goal node
      if node == end:
        return path
      #traverse to all neighbors and add paths to stack
      for i in graph_dict[node]:
        temp_path = list(path)
        temp_path.append(i)
        queue.append(temp_path)
  #false if no path exists
  return ['false']
```

As described in 3.1.3, a deque is used instead of a list due to its increased efficiency in popping from the start.

3.2.4 BFS in Context

BFS always finds the shortest path between nodes. This aspect of BFS is useful for our purposes. People looking for the fastest way to get from one airport to another, without regard to cost, will be able to use BFS to find the quickest way to get to their destination. This is especially useful to plan emergency flights. However, BFS does not optimize for cost.

3.3 Dijkstra's Algorithm

Unlike BFS and DFS, Dijkstra's algorithm [4] takes the weight of an edge into account when determining the optimal path, and seeks to minimize the weights of the edges in the path when going from the starting destination to the goal destination.

3.3.1 Mechanism

Djikstra's algorithm begins by setting the distances of every node to infinity, except the start node, which is given a distance of 0. In every iteration, the node with the least distance is removed from the queue. Then, the theoretical distances to the node's neighbors are calculated. If the theoretical distance is less than the current distance of the neighbor node, the distance of the neighbor node is updated to the theoretical distance, and the neighbor node and its new distance are added to the queue. Consider an example where we want to traverse from A to E in the least costly path possible:

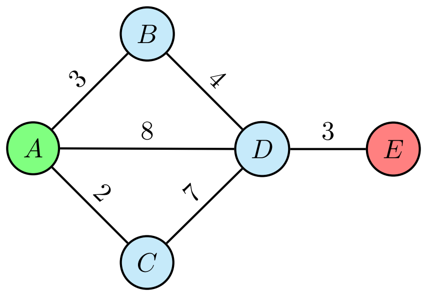

Figure 5 : Djikstra's Example

First, all the nodes are set to infinite distance except the starting node (A)

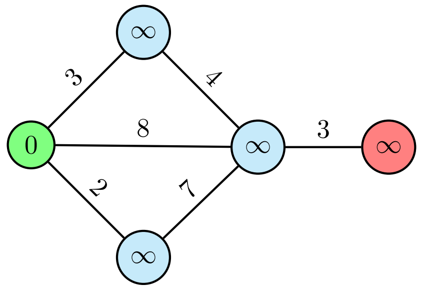

Figure 6 : Step 1

Then, the node with the lowest weight(A) is picked and its neighors are traversed to, updating their weights accordingly

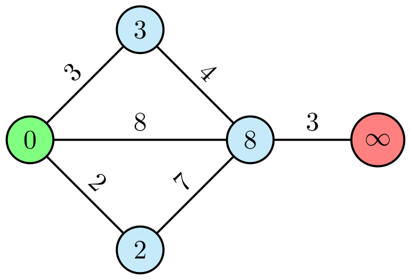

Figure 7 : Step 2

The next node with lowest distance is $(C) : 2$. Repeat, updating weights. There are no weights to update. Pick the next lowest distance node, B:3. Repeat, updating weights

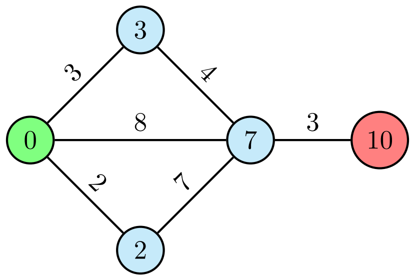

Figure 8 : Step 4

The algorithm is now done running and the final distance to E, the red vertex, can be observed as its distance value, 10.
This example highlights the weight-minimizing aspect of Djikstra's Algorithm. Note that the shortest path length path was not found but rather a longer path that minimized the weight.
Now consider how to make longer paths less favorable. For example, if there is a path of size 4 and size 2 with the same weight, how can Djikstra's Algorithm be modified to favor the path of size 2. One can simply add a constant to each weight, which has the effect of favoring shorter paths. Let us prove this statement. Let there be 2 paths of weight w with one path of size m and another path of size n in the same graph with $m > n$. If we add a constant c to every weight in the path, the new weights w_{newm}, w_{newn} are

$$w_{newm} = w + cm$$
$$w_{newn} = w + cn$$
$$w_{newm} ? w_{newn}$$
$$w + cm ? w + cn$$
$$cm ? cn$$
$$m ? n$$

Since $m > n$, $w_{newm} > w_{newn}$

So, the transformation causes the weight of the longer path to increase more than the shorter path, causing the shorter path to be favored.

3.3.2 Algorithm Pseudocode

The following steps are used to run Djikstra's Algorithm. Note that keeping track of the last node in the path to a node is not normally done when Djikstra's algorithm is implemented, but is neccessary to add in our puposes to determine the least costly path between 2 nodes.

```
Set all node distances to infinity
Set starting node distance to 0
Set dictionary to keep track of last node in path
Add starting node to queue
while queue contains paths do
    Remove least weight path from queue
    Take last node of path
    for Each neighbor of node do
        Add current path weight to weight to traverse to neighbor
        if If tentative neighbor weight less than current neighbor weight then
            Update neighbor weight to tentative weight
            Set current node to last node for neighbor path
            Add neighbor node and neighbor weight to queue
        end if
    end for
end while
```

This algorithm finds the distance from the start node to the end node for all the nodes in the graph. However, it does not return the path from the start node to the end node. To add that functionality, we will need to add another algorithm to return the path to a node given the last node dictionary from the previous algorithm. This algorithm will backtrack from the ending node to get to the starting node. The algorithm is as follows:

```
Find previous node for ending node using dictionary
Create path list and add ending node
while previous node exists do
    Add previous node to path list and make current node
    Find previous node of current node
end while
Reverse path list and return        ▷ Adding to end of list made path reverse
```

3.3.3 Implementation

With the pseudocode, an implementation of Djikstra's algorithms can be made.

```python
#import priority queue
import heapq

def djikstras(start, end, graph):
  #initialize distances as infinity, except start
  distances = {node : float('inf') for node in graph}
  distances[start] = 0
  #initialze dictionary to keep track of previous node in path
  last_node = {node: None for node in graph}
  #create priority queue to traverse by dfs through graph
  queue = [(0, start)]

  while queue:
    #pop least weight path
    temp_distance, temp_node = heapq.heappop(queue)
    #traverse to neighbors
    for next,weight in graph[temp_node].items():
      #calculate tentative weight for neighbor node
      next_distance_temp = temp_distance + weight
      #if path weight less than current, overwrite current weight
      if next_distance_temp < distances[next]:
        distances[next] = next_distance_temp
        last_node[next] = temp_node
        #push path to priority queue
        heapq.heappush(queue, (next_distance_temp, next))

    #backtrack from node to find path
    exist = last_node[end]
    path = [end]
    while exist:
      path.append(exist)
      exist = last_node[exist]
    path.reverse()

  return distances, last_node, path
```

Note that the heapq structure was imported for use. Heapq is a priority queue structure that can pop the element with least priority from the queue. In this case, the weight of the path was used as the priority and the priority queue was able to pop the least weight path in every iteration. Priority queue does this operation efficiently with a time complexity of O(log(N)).

3.3.4 Djikstra's Algorithm in Context

Djikstra's Algorithm always finds the least costly path between nodes. This aspect is useful for our purposes. Using Djikstra's algorithm, we can find the cheapest flight route between 2 airports. Even if there is a direct flight between airports, it is often cheaper to take a layover, and Djikstra's can help us find these cases, helping customers who want the most economical way to go from one airport to another.
Additionally, we can run Djikstra's Algorithm on the modified graph with the constant added to each weight created in 2.2. This would result in shorter routes being favored, which represents the general mentality towards flights: people favor shorter flights unless a route with additional layovers is significantly cheaper. The constant chosen was 80 since that is the approximate opportunity cost of taking a layover.

4 Algorithm Comparison Analysis

4.1 Testing

Algorithms were evaluated based on 3 different factors, namely time to run, price of route generated, and number of flights in the route generated. The Python library, Matplotlib.pyplot was used to visualize the results. 25,000 random routes were chosen and each algorithm was run on each of the routes.

4.2 Picking Random Routes

The Random library in Python was used to generate random routes. A start and ending airport were randomly picked from the total set of airports. If the starting and ending airport turned out to be the same, the ending airport was randomly picked again to ensure that the starting and ending airport are distinct. This ensures that the data isn't skewed with routes of size 0 with 0 cost. The following code accomplishes the task:

```
keys = list(set(graph_dict.keys()))
start = random.choice(keys)
end = random.choice(keys)
while end==start:
end = random.choice(keys)
```

4.3 Running the Algorithms

Using the method describe in 4.0.2, 25,000 random routes were chosen. For each route, each algorithm, namely, DFS, BFS, Djikstra's, and the Modified Djikstra's, and the time to run, path length generated, and cost of route were evaluated.

4.4 Run Time

To evaluate the time for each algorithm to run, the Time library was used. By similar code as the code below, the time for the algorithm to run could be measured in seconds.

```python
#measure time for algorithm to run
t0 = time.time()
#algorithm()
t1 = time.time()
time_to_run = t1-t0
```

The run times for each algorithm over the 25,000 trials were averaged and stored in a 2D list. Using the library, Matplotlib, the average times for the algorithms were graphed. The following graph was generated:

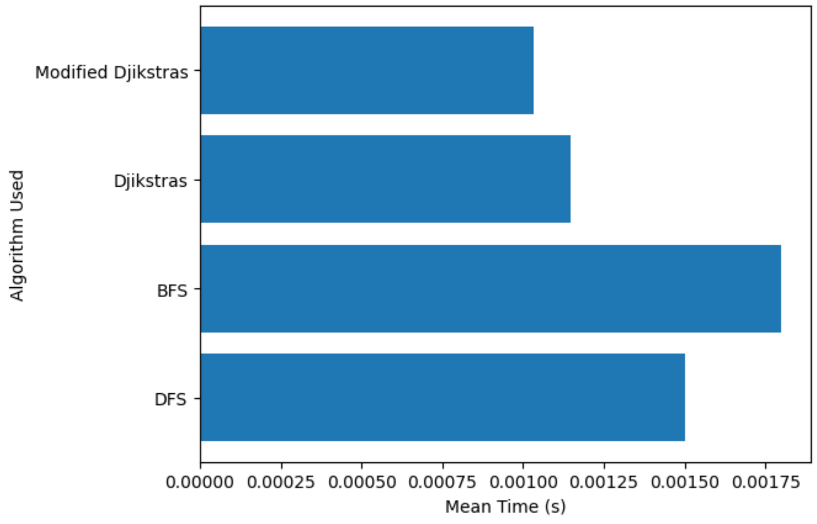

Figure 9 : Average Run Time

The following table shows the exact values for time obtained:

DFS	BFS	Djikstra's	Modified Djikstra's
0.001500 s	0.001800 s	0.001147 s	0.001032 s

4.5 Path Length

For each algorithm, the length of the path generated was stored and averaged over the 25,000 trials. Graphing the path lengths for each algorithm generated the following graphs.

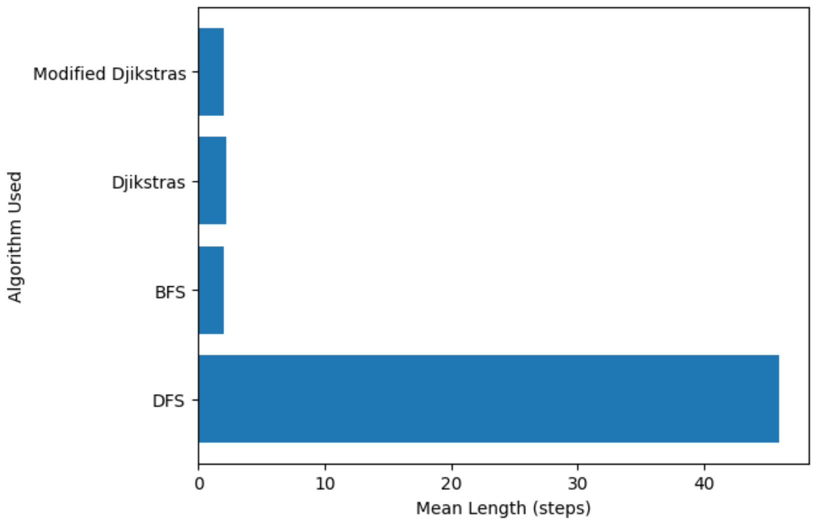

Figure 10 : Average Path Length

A closer look at just BFS, Djikstra's, and Modified Djikstra's

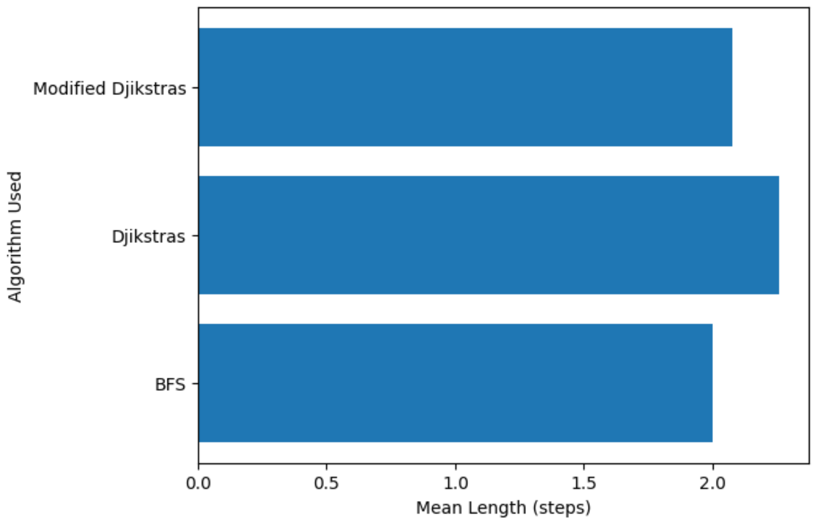

Figure 11 : Mean Path Length

The following table shows the exact values for mean path length obtained:

DFS	BFS	Djikstra's	Modified Djikstra's
45.97452	2.00268	2.26092	2.07588

4.6 Path Cost

For each algorithm, the cost of the path generated was evaluated. The following function was written to find the cost of a path. It looks up the value of each weight in the path and sums it up.

```python
#find cost of a given path
def cost_from_path(path):
  cost = 0
  for i in range(len(path)-1):
    j = i+1
    cost+=weighted_graph_dict[path[i]][path[j]]
  return cost
```

The path costs for each algorithm over the 25,000 trials was averaged. The following graphs represent the average path cost for each algorithm:

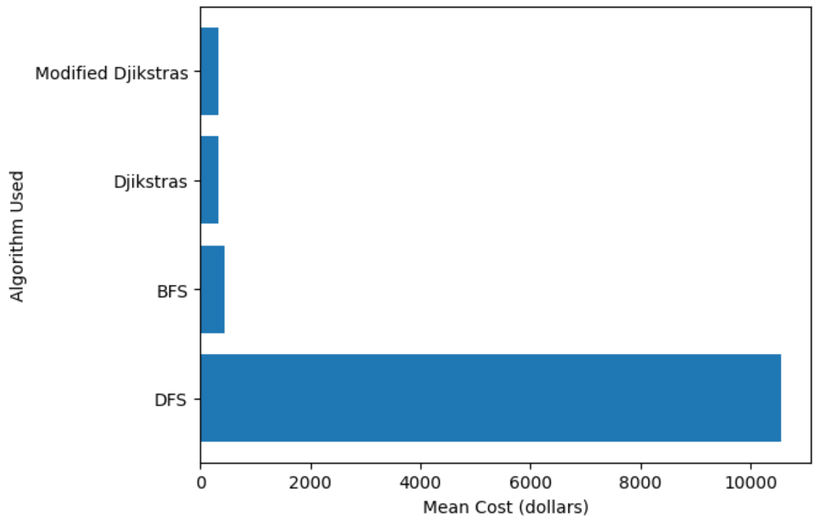

Figure 12 : Average Route Cost

A closer look at just BFS, Djikstra's, and Modified Djikstra's.

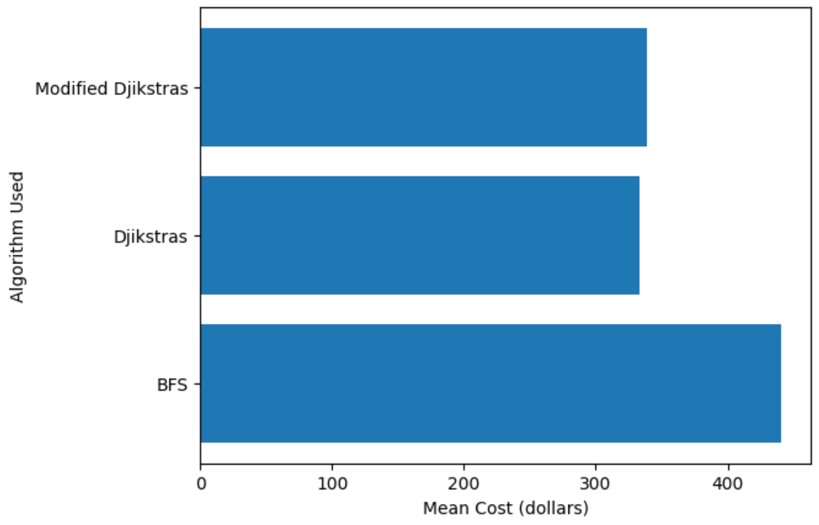

Figure 13 : Mean Route Price

The following table shows the mean values for cost obtained for each algorithm:

DFS	BFS	Djikstra's	Modified Djikstra's
\$10557.98	\$440.98	\$333.24	\$339.00

5 Conclusion

After analyzing the results, it is clear that DFS is an inefficient and does not generate viable results for flight paths. The cost of a DFS-generated route tends to be unrealistically high and takes over 40 flights for a route that can be done within 4 flights. This makes DFS unfeasible for this application. A unique detail to note is that DFS ran slower than Djikstra's Algorithm which is not supported by the time complexity. This might be due to large scale of the graph which made DFS traverse to many airports before finding the actual ending airport.

BFS, on the other hand seems to be a practical algorithm for finding an optimal flight route. Its run time is similar to the other algorithms and runs extremely fast. Since BFS always finds the shortest path between 2 nodes, BFS had the shortest mean path length. This can be useful for people looking to get from one airport to another in the shortest possible time without regard for cost; for example, people looking to fly for emergency reasons may use BFS to quickly get to their destination. It can be seen that flight routes generated by BFS cost approximately \$101.98 greater than the routes generated by Djikstra's Algorithm and the the modified Djikstra's. Interestingly, the mean path size difference between modified Djikstra's and BFS is just .073, while the mean cost differs by approximately \$101.98. This can be since BFS simply picks the first path of shortest length that it traverses to while the modified Djikstra's might find a path of the same length but with a lower cost.

Djikstra's Algorithm seems to be a balanced algorithm. It runs slower than the modified Djikstra's algorithm but faster than BFS and DFS. Additionally, it generates a mean path length similar to , but slightly longer than, BFS and modified Djikstra's. Djikstra's Algorithm generated the lowest mean route cost due to its weight-minimizing aspect, which could be useful for travelers on a budget trying to minimize costs without regard for how many layovers were taken.

The modified Djikstra's algorithm seems to be the best overall algorithm that serves a general use case. The algorithm ran the fastest out of all the algorithms. This might have happened since the algorithm does not favor long paths and halts earlier than the other algorithms. It had a mean path length very close to BFS, while saving \$101.98 per route on average, making the algorithm useful for people looking for short routes while also saving money. The modified Djikstra's Algorithm also spent \$5.76 more than Djikstra's Algorithm on average, while having a noticeably lower mean flight route size, which reflects the preferences of most people: paying slightly more for a shorter route is fine due to the opportunity cost of taking a layover. The modified Djikstra's algorithm is similar to a hybrid between BFS and Djikstra's Algorithm, accounting for both route size and cost, making it a good general purpose algorithm for people looking for flight routes.

6 Further Study/Discussion

Future studies can be conducted with an improved dataset. Our current dataset contained only one side of a flight route, but having a dataset with both sides of a flight route will allow us to account for the difference in cost between different directions of the same route. Additionally, it seemed that some specific airports in our dataset were missing several routes, so having a more accurate dataset will generate more accurate results.

Additionally, it would be desirable to have a dataset with the geographic coordinates so that the A* algorithm can be used. With the coordinates of the starting and ending airport, a heuristic function that contributes to the weights can be generated for each airport. The heuristic function would generate low values for airports that are in the general direction of the ending airport and high values for airports that are moving away from the ending airport. This would allow our algorithm to prioritize the layover flights that are moving towards the final airport. By combining A* with our modified Djikstra's we will have an algorithm that is faster than the previous algorithms and still generates short and low-cost paths, an optimal combination of factors.

References

[1] Miao Bin. *Graph Traversal in Python: Breadth First Search (BFS)*. URL: https://medium.com/nerd-for-tech/graph-traversal-in-python-breadth-first-search-bfs-b6cff138d516.

[2] Shimon Even. "Graph Algorithms". In: Cambridge University Press, 2008, pp. 46–48.

[3] Bhavik Jikadara. *US Airline Flight Routes and Fares 1993-2024*. URL: https://www.kaggle.com/datasets/bhavikjikadara/us-airline-flight-routes-and-fares-1993-2024.

[4] Kurt Melhorn. "Algorithms and Data Structures: The Basic Toolbox". In: Springer, 2008. Chap. 10.

[5] William T. Trotter Mitchel T. Keller. *Applied Combinatorics*. 1986.

Appendix A : Source Code

All source code used in the paper can be found in the following link: `https://www.github.com/shawnaks/MATH-3012-Project`