Optimized Flight Trip

CCE414-Artificial Intelligence

Team members

1)Ahmed Mohamed Fawzy. G01
ID:221903175

2) Passant El-Tonsy Ali. G01
ID:221903220

3)Saif Emad El-Deen Abdel-Kareem. G02
ID:231903756

4) Mohamed Emad Fawzy. G02
ID:231903548

Under Supervision

—

Prof.Dr. Lamiaa Elrefaei.

Date Due	**May 7th,2024.**
Date Handed in	**May 4th,2024.**

ABSTRACT

Efficient flight trip planning is crucial in today's fast-paced travel industry. This report investigates the application of Artificial Intelligence (AI) searching algorithms to optimize flight itinerary selection. By leveraging AI techniques such as A*, DFS, BFS and UCS algorithms, we aim to develop a system that can rapidly compute and recommend the most optimal flight routes based on user preferences of paths and it can be upgraded . The methodology involves the implementation of these AI algorithms within a prototype flight trip planning system. Real-world flight data is utilized to test and evaluate the system's performance in terms of route accuracy, computation time, and adaptability to changing constraints. Through rigorous experimentation and analysis, we demonstrate the effectiveness of AI searching algorithms in efficiently navigating complex flight networks and generating optimized itineraries. The results highlight the potential of AI-driven solutions to revolutionize the travel planning process, offering personalized and cost-effective flight options to travellers. We also offer you a web application to better visualization.

ACKNOWLEDGEMENT

I would like to extend my heartfelt appreciation to the individuals whose work has greatly contributed to the completion of this project. Their dedication and expertise have been invaluable in shaping the outcomes presented in this report.

We're also indebted to Prof.Dr. Lamiaa El-Refaei and Eng.Mohamed Rehan for generously sharing their expertise and providing invaluable guidance throughout the duration of this project. Their depth of knowledge and willingness to offer assistance have been instrumental in overcoming challenges and achieving our objectives.

Additionally, we extend our appreciation to the authors of the various studies, reports, and publications cited in this work. Their groundbreaking research laid the foundation for our understanding and served as a source of inspiration for our endeavours.

Lastly, we wish to thank our Parents, mentors, advisors, and supervisors for their continuous encouragement and guidance throughout this journey. Their unwavering support and mentorship have been instrumental in shaping our growth and development as researchers.

TABLE OF CONTENT

1- Introduction:

This report explores the application of AI searching algorithms e.g. BFS-DFS-A*-Greedy-UCS-IDS-DLS in optimizing flight trip planning. By harnessing the power of AI, we aim to develop a system capable of efficiently navigating complex flight networks to identify the most cost-effective and time-efficient routes. This project not only delves into the technical aspects of AI algorithms but also examines their real-world implications for travellers, airlines, and the broader travel industry.

2- Exploratory Data Analysis:

2.1- Libraries:

- In the following figure you can find all used libraries to build our project.

```
In [1]: import numpy as np
        import pandas as pd
        import plotly.express as px
        import math
        import plotly.graph_objects as go
        import time
        import sys

        from collections import deque
        from utils import *
```

2.2- Loading Data:

```
In [2]: dataset_df=pd.read_csv("Dataset.csv")
```

2.3- Exploring Data:

- Run **head()** to get first 5 rows.

```
In [3]: # The whole world's real data
        dataset_df.head()
Out[3]:
```

	Airline	SourceAirport	DestinationAirport	SourceAirport_City	SourceAirport_Country	SourceAirport_Latitude	SourceAirport_Longitude	SourceAirport_Altit
0	Aerocondor	Sochi International Airport	Kazan International Airport	Sochi	Russia	43.449902	39.956600	
1	Aerocondor	Astrakhan Airport	Kazan International Airport	Astrakhan	Russia	46.283298	48.006302	
2	Aerocondor	Chelyabinsk Balandino Airport	Kazan International Airport	Chelyabinsk	Russia	55.305801	61.503300	
3	Aerocondor	Domodedovo International Airport	Kazan International Airport	Moscow	Russia	55.408798	37.906300	
4	Aerocondor	Belgorod International Airport	Kazan International Airport	Belgorod	Russia	50.643799	36.590099	

- to get size and shape:

```
In [4]: dataset_df.size
Out[4]: 584320

In [6]: dataset_df.shape
Out[6]: (36520, 16)
```

- All information:

```
In [7]: dataset_df.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 36520 entries, 0 to 36519
Data columns (total 16 columns):
 #   Column                        Non-Null Count  Dtype
---  ------                        --------------  -----
 0   Airline                       36520 non-null  object
 1   SourceAirport                 36520 non-null  object
 2   DestinationAirport            36520 non-null  object
 3   SourceAirport_City            36520 non-null  object
 4   SourceAirport_Country         36520 non-null  object
 5   SourceAirport_Latitude        36520 non-null  float64
 6   SourceAirport_Longitude       36520 non-null  float64
 7   SourceAirport_Altitude        36520 non-null  int64
 8   DestinationAirport_City       36520 non-null  object
 9   DestinationAirport_Country    36520 non-null  object
 10  DestinationAirport_Latitude   36520 non-null  float64
 11  DestinationAirport_Longitude  36520 non-null  float64
 12  DestinationAirport_Altitude   36520 non-null  int64
 13  Distance                      36520 non-null  float64
 14  FlyTime                       36520 non-null  float64
 15  Price                         36520 non-null  float64
dtypes: float64(7), int64(2), object(7)
memory usage: 4.5+ MB
```

- To get ranges of altitude and longitude:

In [8]:

```
# The ranges of latitude, longitude, and altitude are naturally occurring
dataset_df.describe()
```

Out[8]:

	SourceAirport_Latitude	SourceAirport_Longitude	SourceAirport_Altitude	DestinationAirport_Latitude	DestinationAirport_Longitude	DestinationAirport_Altitu
count	36520.000000	36520.000000	36520.000000	36520.000000	36520.000000	36520.000
mean	32.296961	8.902498	743.114595	32.297468	8.919798	746.644
std	21.665060	77.269967	1430.097451	21.672549	77.269350	1435.930
min	-54.843300	-179.876999	-72.000000	-54.843300	-179.876999	-72.000
25%	24.957600	-68.363403	41.000000	24.957600	-68.268501	41.000
50%	37.618999	9.221960	184.000000	37.618999	9.276740	184.000
75%	47.121899	58.284401	681.000000	46.991100	58.284401	681.000
max	78.246101	179.341003	14472.000000	78.246101	179.341003	14472.000

3- Problem Formulation:

3.1- Problem Class

3.1.1- Initial State:

```
def __init__(self, initial, goal=None):
    """The constructor specifies the initial state, and possibly a goal
    state, if there is a unique goal. Your subclass's constructor can add
    other arguments."""
    self.initial = initial
    self.goal = goal
```

3.1.2- Actions:

```
def actions(self, state):
    """Return the actions that can be executed in the given
    state. The result would typically be a list, but if there are
    many actions, consider yielding them one at a time in an
    iterator, rather than building them all at once."""
    raise NotImplementedError
```

3.1.3- Transition Model

```
def result(self, state, action):
    """Return the state that results from executing the given
    action in the given state. The action must be one of
    self.actions(state)."""
    raise NotImplementedError
```

3.1.4- Goal Test:

```python
def goal_test(self, state):
    """Return True if the state is a goal. The default method compares the
    state to self.goal or checks for state in self.goal if it is a
    list, as specified in the constructor. Override this method if
    checking against a single self.goal is not enough."""
    if isinstance(self.goal, list):
        return is_in(state, self.goal)
    else:
        return state == self.goal
```

3.1.5- Path Cost:

```python
def path_cost(self, c, state1, action, state2):
    """Return the cost of a solution path that arrives at state2 from
    state1 via action, assuming cost c to get up to state1. If the problem
    is such that the path doesn't matter, this function will only look at
    state2. If the path does matter, it will consider c and maybe state1
    and action. The default method costs 1 for every step in the path."""
    return c + 1
```

3.2- Node Class:

A node in a search tree. Contains a pointer to the parent (the node that this is a successor of) and to the actual state for this node. Note that if a state is arrived at by two paths, then there are two nodes with the same state. Also includes the action that got us to this state, and the total path_cost (also known as g) to reach the node.

3.3- Graph Class

A graph connects nodes (vertices) by edges (links). Each edge can also have a length associated with it. The constructor call is something like g = Graph ({'A': {'B': 1, 'C': 2}.

```
In [11]: class Graph:

    def __init__(self, graph_dict=None, directed=True):
        self.graph_dict = graph_dict or {}
        self.directed = directed
        if not directed:
            self.make_undirected()

    def make_undirected(self):
        """Make a digraph into an undirected graph by adding symmetric edges."""
        for a in list(self.graph_dict.keys()):
            for (b, dist) in self.graph_dict[a].items():
                self.connect1(b, a, dist)

    def connect(self, A, B, distance=1):
        """Add a link from A and B of given distance, and also add the inverse
        link if the graph is undirected."""
        self.connect1(A, B, distance)
        if not self.directed:
            self.connect1(B, A, distance)

    def connect1(self, A, B, distance):
        """Add a link from A to B of given distance, in one direction only."""
        self.graph_dict.setdefault(A, {})[B] = distance

    def get(self, a, b=None):
        """Return a link distance or a dict of {node: distance} entries.
        .get(a,b) returns the distance or None;
        .get(a) returns a dict of {node: distance} entries, possibly {}."""
        links = self.graph_dict.setdefault(a, {})
        if b is None:
            return links
        else:
            return links.get(b)

    def nodes(self):
        """Return a list of nodes in the graph."""
        s1 = set([k for k in self.graph_dict.keys()])
        s2 = set([k2 for v in self.graph_dict.values() for k2, v2 in v.items()])
        nodes = s1.union(s2)
        return list(nodes)
```

3.4- Graph Problem

Subclass from Problem class to Build specific problem definitions which is Flight Trip.

```
class GraphProblem(Problem):
    """The problem of searching a graph from one node to another."""

    def __init__(self, initial, goal, graph):
        super().__init__(initial, goal)
        self.graph = graph

    def actions(self, A):
        """The actions at a graph node are just its neighbors."""
        return list(self.graph.get(A).keys())

    def result(self, state, action):
        """The result of going to a neighbor is just that neighbor."""
        return action

    def path_cost(self, cost_so_far, A, action, B):
        return cost_so_far + (self.graph.get(A, B) or np.inf)    # get(a,b) --> get the distance between node a and b

    def find_min_edge(self):
        """Find minimum value of edges."""
        m = np.inf
        for d in self.graph.graph_dict.values():
            local_min = min(d.values())
            m = min(m, local_min)

        return m
# Write your heuristac function (3d distance)
    def h(self, node):
        """h function is straight-line distance from a node's state to goal."""
        locs = getattr(self.graph, 'locations', None)
        if locs:
            if type(node) is str:
                return int(distance(locs[node], locs[self.goal]))

            return int(distance(locs[node.state], locs[self.goal]))
        else:
            return np.inf     # Heuristic for unavailable node
```

3.5- Heuristic Function:

Heuristic function is the shortest path between source airport and destination airport, it can be calculated using Euclidean distance.

```
In [14]: # Our heuristac (the stright line distance)
         def distance(source, destination):
             d1_source, d2_source, d3_source=source
             d1_destination, d2_destination, d3_destination=destination

             heuristic = math.sqrt(((d1_source - d1_destination) ** 2) + ((d2_source - d2_destination) ** 2) +
                                   ((d3_source - d3_destination) ** 2))
             return heuristic
```

3.6- Flight Trip Graph:

- **Build node and successors dictionary for passing it to the graph.**

```
world_dict={}
for source_airport in list(df['SourceAirport'].unique()):
    source_to_destinations_df=df[df['SourceAirport']==source_airport][['SourceAirport','DestinationAirport','Distance']]

    destinations_dict={}
    for index, row in source_to_destinations_df.iterrows():
        destinations_dict.update({row['DestinationAirport']: row['Distance']})

    world_dict.update({source_airport: destinations_dict})
```

- **Instantiate Undirected Graph.**
- **Locations in Latitude, Longitude and Altitude for calculate Heuristic function.**

```
In [17]: # Locations in Latitude, Longitude and Altitude for calculate Heuristac function
         locations_dict={}
         for nod in list(df['SourceAirport'].unique()):
             nod_df=df[df['SourceAirport']==nod][['SourceAirport_Latitude','SourceAirport_Longitude','SourceAirport_Altitude']]
             locations_dict.update({nod:tuple(nod_df.iloc[0])})

In [18]: # Handle all Airpots Locations -- sources and destinations
         for nod in list(df['DestinationAirport'].unique()):
             nod_df=df[df['DestinationAirport']==nod][['DestinationAirport_Latitude','DestinationAirport_Longitude','DestinationAirport_Al
             if nod not in locations_dict.keys():
                 locations_dict.update({nod:tuple(nod_df.iloc[0])})
```

- **Handle all Airports locations -- sources and destinations then pass a new attribute location.**

```
In [18]: # Handle all Airpots Locations -- sources and destinations
         for nod in list(df['DestinationAirport'].unique()):
             nod_df=df[df['DestinationAirport']==nod][['DestinationAirport_Latitude','DestinationAirport_Longitude','DestinationAirport_Al
             if nod not in locations_dict.keys():
                 locations_dict.update({nod:tuple(nod_df.iloc[0])})

In [19]: # Pass a new attribute locations
         world_map.locations=locations_dict
```

- **Pass initial and goal states then instantiate our problem from 'Imam Khomeini International Airport' to 'Raleigh Durham International Airport'.**

```
In [20]: # Pass intial and goal states
         airport_intial='Imam Khomeini International Airport'
         airport_goal='Raleigh Durham International Airport'
         # Instantiate Our Problem
         world_problem = GraphProblem(airport_intial , airport_goal , world_map)
```

- **Draw the path solution for the problem and adding another path solution for your figure using longitude and latitude.**

```
l]: # Draw the path solution for the problem
    def draw_path (lat_list,lon_list):
        fig = go.Figure(go.Scattermapbox(
        mode = "markers+lines",
        lon = lon_list,
        lat = lat_list,
        marker = {'size': 10}))

        fig.update_layout(
        margin ={'l':0,'t':0,'b':0,'r':0},
        mapbox = {
            'center': {'lon': 10, 'lat': 10},
            'style': "open-street-map",
            'center': {'lon': -20, 'lat': -20},
            'zoom': 1})

        return fig
```

```
2]: # Add another path solution for your figure
    def add_trace_path(fig,lat_list,lon_list,name=None):
        fig.add_trace(go.Scattermapbox(
        mode = "markers+text+lines",
        lon = lon_list,
        lat = lat_list,
        text=name,
        marker = {'size': 10}))
        return fig
```

4- Uninformed Search Algorithms:

4.1- Breadth First Search:

BFS is an algorithm that explores a graph level by level. Starting at the root (or an initial node), it systematically explores all neighbours at the present depth before moving on to nodes at the next depth level.

```
In [26]: def breadth_first_graph_search(problem):
             """[Figure 3.11]
             Note that this function can be implemented in a
             single line as below:
             return graph_search(problem, FIFOQueue())
             """
             node = Node(problem.initial)
             if problem.goal_test(node.state):
                 return node

             frontier = deque([node])
             explored = set()
             while frontier:
                 node = frontier.popleft()
                 explored.add(node.state)
                 for child in node.expand(problem):
                     if child.state not in explored and child not in frontier:
                         if problem.goal_test(child.state):
                             return child
                         frontier.append(child)
             return None
```

- **Execute algorithm and calculate execution time.**

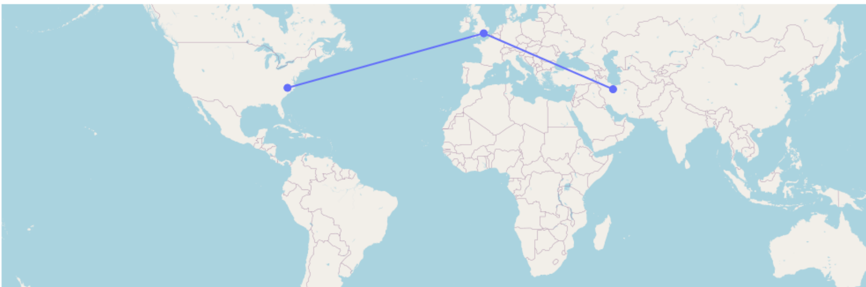
```
In [27]: # Execute algorithm and calculate execution time
         start_time = time.time()
         breadth_node=breadth_first_graph_search(world_problem)
         elapsed_time = time.time()  - start_time
         minutes, seconds = divmod(elapsed_time, 60)
         print("Execution Time: {:.0f}m {:.5f}s".format(minutes, seconds))

         Execution Time: 0m 0.17955s
```

- **Sequence of actions from the initial node to the goal.**

```
In [28]: # Sequence of actions from the initial node to the Goal
         print(breadth_node.solution())

         ['London Heathrow Airport', 'Raleigh Durham International Airport']
```

- **Visualization the solution path.**

```
In [29]: # Visualiztion the solution path
         path_states=[]
         for node in breadth_node.path():
             path_states.append(node.state)

         lat_list=[world_map.locations_2d[state][0] for state in path_states ]
         lon_list=[world_map.locations_2d[state][1] for state in path_states ]

         path_fig=draw_path(lat_list,lon_list)
         path_fig.show()
```



4.2- Depth First Search:

DFS is an algorithm that explores a graph by going as deeply as possible down one path before backing up and exploring another path.

```
In [30]: def depth_first_graph_search(problem):
             """
             [Figure 3.7]
             Search the deepest nodes in the search tree first.
             Search through the successors of a problem to find a goal.
             The argument frontier should be an empty queue.
             Does not get trapped by loops.
             If two paths reach a state, only use the first one.
             """
             frontier = [(Node(problem.initial))]  # Stack

             explored = set()
             while frontier:
                 node = frontier.pop()
                 if problem.goal_test(node.state):
                     return node
                 explored.add(node.state)
                 frontier.extend(child for child in node.expand(problem)
                                 if child.state not in explored and child not in frontier)
             return None
```

- **Execute algorithm and calculate execution time.**

```
In [31]: # Execute algorithm and calculate execution time
         start_time = time.time()
         depth_first_node=depth_first_graph_search(world_problem)
         elapsed_time = time.time()  - start_time
         minutes, seconds = divmod(elapsed_time, 60)
         print("Execution Time: {:.0f}m {:.5f}s".format(minutes, seconds))

         Execution Time: 0m 1.03989s
```

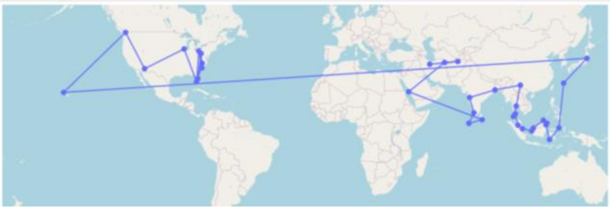
- **Sequence of actions from the initial node to the goal.**


```
In [31]: # Execute algorithm and calculate execution time
         start_time = time.time()
         depth_first_node=depth_first_graph_search(world_problem)
         elapsed_time = time.time()  - start_time
         minutes, seconds = divmod(elapsed_time, 60)
         print("Execution Time: {:.0f}m {:.5f}s".format(minutes, seconds))

         Execution Time: 0m 1.03989s
```


```
In [32]: # Sequence of actions from the initial node to the Goal
         print(depth_first_node.solution())

         ['Mazar I Sharif Airport', 'Mashhad International Airport', 'King Abdulaziz International Airport', 'Mattala Rajapaksa International Airport', 'Malé International Airport', 'Cochin International Airport', 'Pune Airport', 'Netaji Subhash Chandra Bose International Airport', 'Kunming Changshui International Airport', 'Don Mueang International Airport', 'Krabi Airport', 'Samui Airport', 'Sultan Abdul Aziz Shah International Airport', 'Hang Nadim International Airport', 'Supadio Airport', 'Kuching International Airport', 'Kota Kinabalu International Airport', 'Tawau Airport', 'Juwata Airport', 'Hasanuddin International Airport', 'Sam Ratulangi Airport', 'Naha Airport', 'Sendai Airport', 'Daniel K Inouye International Airport', 'Bellingham International Airport', 'Phoenix-Mesa-Gateway Airport', 'Chicago Rockford International Airport', 'Charlotte County Airport', 'Youngstown Warren Regional Airport', 'Myrtle Beach International Airport', 'Arnold Palmer Regional Airport', 'Orlando International Airport', 'Raleigh Durham International Airport']
```

- **Visualization the path.**

```
In [33]: # Visualiztion the path
         path_states=[]
         for node in depth_first_node.path():
             path_states.append(node.state)

         lat_list=[world_map.locations_2d[state][0] for state in path_states ]
         lon_list=[world_map.locations_2d[state][1] for state in path_states ]

         path_fig=draw_path(lat_list,lon_list)
         path_fig.show()
```



4.3- Depth Limited Search:

DLS is a variant of DFS where the search is limited to a specified depth. It avoids infinite loops by not expanding nodes beyond a certain depth limit.

```
In [34]: def depth_limited_search(problem, limit=50):
             """[Figure 3.17]"""

             def recursive_dls(node, problem, limit):
                 if problem.goal_test(node.state):
                     return node
                 elif limit == 0:
                     return 'cutoff'
                 else:
                     cutoff_occurred = False
                     for child in node.expand(problem):
                         result = recursive_dls(child, problem, limit - 1)
                         if result == 'cutoff':
                             cutoff_occurred = True
                         elif result is not None:
                             return result
                     return 'cutoff' if cutoff_occurred else None

             # Body of depth_limited_search:
             return recursive_dls(Node(problem.initial), problem, limit)
```

- **Execute algorithm and calculate execution time.**

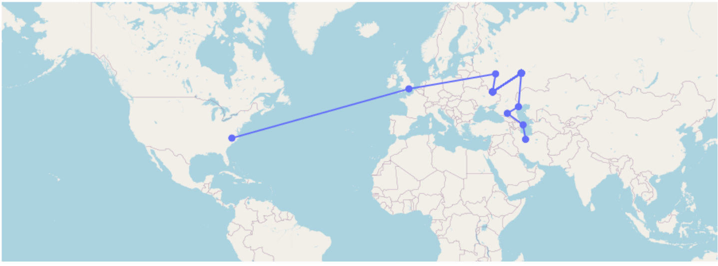
```
In [35]: # Execute algorithm and calculate execution time
         start_time = time.time()
         depth_limited_node=depth_limited_search(world_problem)
         elapsed_time = time.time()  - start_time
         minutes, seconds = divmod(elapsed_time, 60)
         print("Execution Time: {:.0f}m {:.5f}s".format(minutes, seconds))

         Execution Time: 0m 0.02020s
```

- **Sequence of actions from the initial node to the goal.**

```
In [36]: # Sequence of actions from the initial node to the Goal
         print(depth_limited_node.solution())

         ['Heydar Aliyev International Airport', 'Mineralnyye Vody Airport', 'Astrakhan Airport', 'Kazan International Airport', 'Belgor
         od International Airport', 'Kazan International Airport', 'Belgorod International Airport', 'Kazan International Airport', 'Bel
         gorod International Airport', 'Kazan International Airport', 'Belgorod International Airport', 'Kazan International Airport',
         'Belgorod International Airport', 'Kazan International Airport', 'Belgorod International Airport', 'Kazan International Airpor
         t', 'Belgorod International Airport', 'Kazan International Airport', 'Belgorod International Airport', 'Kazan International Air
         port', 'Belgorod International Airport', 'Kazan International Airport', 'Belgorod International Airport', 'Kazan International
         Airport', 'Belgorod International Airport', 'Kazan International Airport', 'Belgorod International Airport', 'Kazan Internation
         al Airport', 'Belgorod International Airport', 'Kazan International Airport', 'Belgorod International Airport', 'Kazan Internat
         ional Airport', 'Belgorod International Airport', 'Kazan International Airport', 'Belgorod International Airport', 'Kazan Inter
         national Airport', 'Belgorod International Airport', 'Kazan International Airport', 'Belgorod International Airport', 'Kazan In
         ternational Airport', 'Belgorod International Airport', 'Kazan International Airport', 'Belgorod International Airport', 'Kazan
         International Airport', 'Belgorod International Airport', 'Kazan International Airport', 'Belgorod International Airport', 'Dom
         odedovo International Airport', 'London Heathrow Airport', 'Raleigh Durham International Airport']
```

- Visualization the path.

```
# Visualiztion the path
path_states=[]
for node in depth_limited_node.path():
    path_states.append(node.state)

lat_list=[world_map.locations_2d[state][0] for state in path_states ]
lon_list=[world_map.locations_2d[state][1] for state in path_states ]

path_fig=draw_path(lat_list,lon_list)
path_fig.show()
```



4.4- Iterative Deepening Search:

IDS is an algorithm that combines the benefits of BFS and DFS. It performs a series of DFS with increasing depth limits until the goal is found.

```
In [38]: def iterative_deepening_search(problem):
             """[Figure 3.18]"""
             for depth in range(sys.maxsize):
                 result = depth_limited_search(problem, depth)
                 if result != 'cutoff':
                     return result
```

- **Execute algorithm and calculate execution time.**

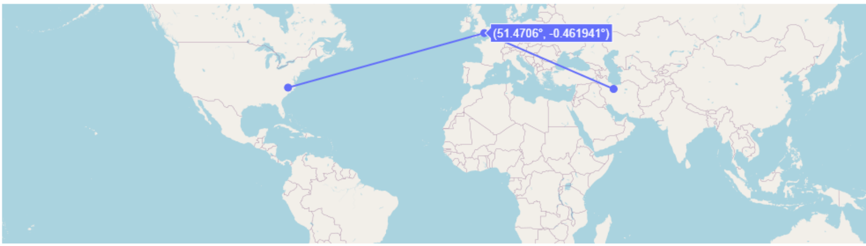
```
In [39]: # Execute algorithm and calculate execution time
         start_time = time.time()
         iterative_deepening_node=iterative_deepening_search(world_problem)
         elapsed_time = time.time()  - start_time
         minutes, seconds = divmod(elapsed_time, 60)
         print("Execution Time: {:.0f}m {:.5f}s".format(minutes, seconds))

         Execution Time: 0m 0.01064s
```

- **Sequence of actions from the initial node to the goal.**

```
In [40]: # Sequence of actions from the initial node to the Goal
         print(iterative_deepening_node.solution())

         ['London Heathrow Airport', 'Raleigh Durham International Airport']
```

- **Visualization the path.**

```
In [41]: # Visualiztion the path
         path_states=[]
         for node in iterative_deepening_node.path():
             path_states.append(node.state)

         lat_list=[world_map.locations_2d[state][0] for state in path_states ]
         lon_list=[world_map.locations_2d[state][1] for state in path_states ]

         path_fig=draw_path(lat_list,lon_list)
         path_fig.show()
```



4.5- Uniform Cost Search:

UCS is an algorithm used for traversing a weighted graph. It expands the node with the lowest cost (path weight) first.

```
# f=g=distance
def uniform_cost_search(problem, display=False):
    """[Figure 3.14]"""
    return best_first_graph_search(problem, lambda node: node.path_cost, display)
```

- **Execute algorithm and calculate execution time.**

```
# Execute algorithm and calculate execution time
start_time = time.time()
uniform_cost_node=uniform_cost_search(world_problem)
elapsed_time = time.time()  - start_time
minutes, seconds = divmod(elapsed_time, 60)
print("Execution Time: {:.0f}m {:.5f}s".format(minutes, seconds))
```

```
Execution Time: 0m 4.30918s
```

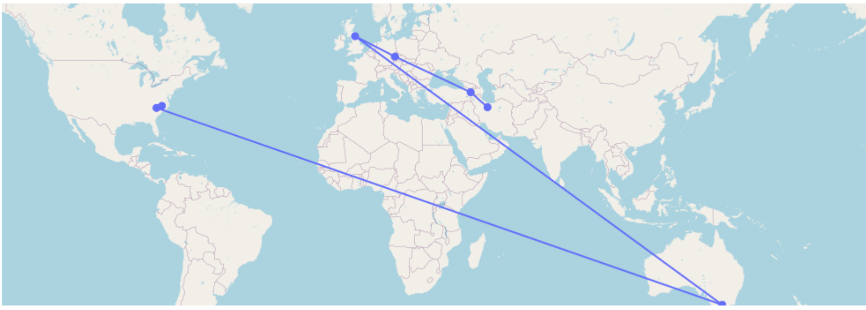
- **Sequence of actions from the initial node to the goal.**

```
# Sequence of actions from the initial node to the Goal
print(uniform_cost_node.solution())
```

```
['Zvartnots International Airport', 'Václav Havel Airport Prague', 'Newcastle Airport', 'Melbourne International Airport', 'Cha
rlotte Douglas International Airport', 'Raleigh Durham International Airport']
```

- **Visualization the path.**

```
path_states=[]
for node in uniform_cost_node.path():
    path_states.append(node.state)

lat_list=[world_map.locations[state][0] for state in path_states ]
lon_list=[world_map.locations[state][1] for state in path_states ]

path_fig=draw_path(lat_list,lon_list)
path_fig.show()
```



5- Informed Search Algorithms:

5.1- Best First Search

5.1.1- Greedy Search Algorithm:

```
In [44]: #Greedy best-first search is accomplished by specifying f(n) = h(n)
greedy_best_first_graph_search = best_first_graph_search
```

- **Execute algorithm and calculate execution time.**

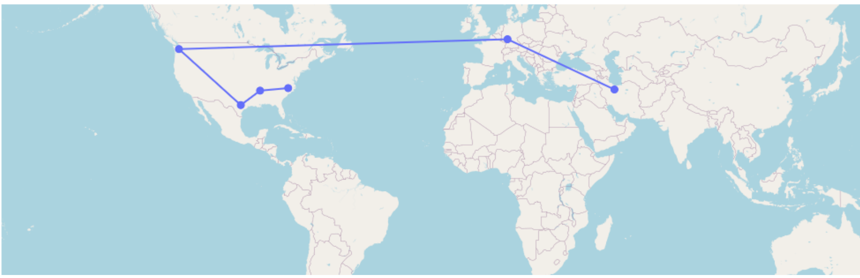
```
In [45]: # Execute algorithm and calculate execution time
start_time = time.time()
greedy_node=greedy_best_first_graph_search(world_problem,world_problem.h)
elapsed_time = time.time()  - start_time
minutes, seconds = divmod(elapsed_time, 60)
print("Execution Time: {:.0f}m {:.5f}s".format(minutes, seconds))

Execution Time: 0m 0.35590s
```

- **Sequence of actions from the initial node to the goal.**

```
In [46]: # Sequence of actions from the initial node to the Goal
print(greedy_node.solution())

['Frankfurt am Main Airport', 'Seattle Tacoma International Airport', 'Austin Bergstrom International Airport', 'Memphis Intern
ational Airport', 'Raleigh Durham International Airport']
```

- **Visualization the path.**

```
In [47]: # Visualiztion the path
path_states=[]
for node in greedy_node.path():
    path_states.append(node.state)

lat_list=[world_map.locations[state][0] for state in path_states ]
lon_list=[world_map.locations[state][1] for state in path_states ]

path_fig=draw_path(lat_list,lon_list)
path_fig.show()
```



5.1.2- A* Search Algorithm:

```
def astar_search(problem, h=None, display=False):
    """A* search is best-first graph search with f(n) = g(n)+h(n).
    You need to specify the h function when you call astar_search, or
    else in your Problem subclass."""
    h = memoize(h or problem.h, 'h')
    return best_first_graph_search(problem, lambda n: n.path_cost + h(n), display)
```

- **Execute algorithm and calculate execution time.**

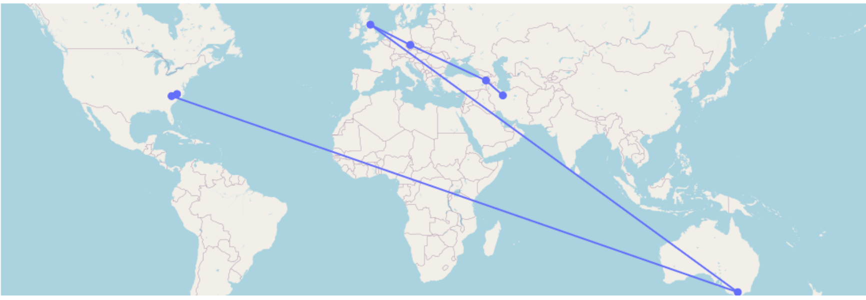
```
In [49]: # Execute algorithm and calculate execution time
start_time = time.time()
astar_node=astar_search(world_problem,world_problem.h)
elapsed_time = time.time()  - start_time
minutes, seconds = divmod(elapsed_time, 60)
print("Execution Time: {:.0f}m {:.5f}s".format(minutes, seconds))

Execution Time: 0m 4.12580s
```

- **Sequence of actions from the initial node to the goal.**

```
In [50]: # Sequence of actions from the initial node to the Goal
print(astar_node.solution())

['Zvartnots International Airport', 'Václav Havel Airport Prague', 'Newcastle Airport', 'Melbourne International Airport', 'Cha
rlotte Douglas International Airport', 'Raleigh Durham International Airport']
```

- **Visualization the path.**

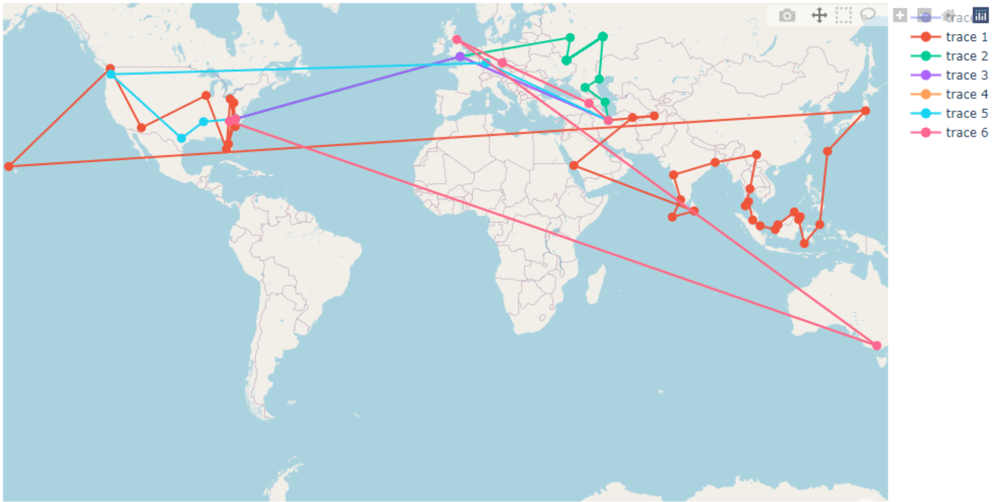
```
In [51]: # Visualiztion the path
path_states=[]
for node in astar_node.path():
    path_states.append(node.state)

lat_list=[world_map.locations[state][0] for state in path_states ]
lon_list=[world_map.locations[state][1] for state in path_states ]

path_fig=draw_path(lat_list,lon_list)
path_fig.show()
```



6- Comparison:

6.1- Different Paths:

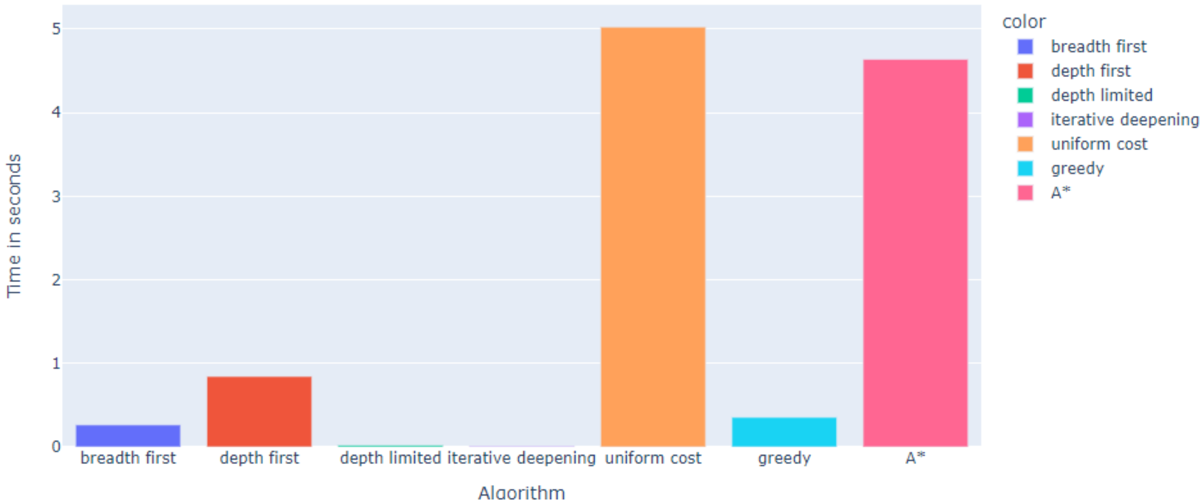
```
In [52]: start_time = time.time()
breadth_node=breadth_first_graph_search(world_problem)
elapsed_time = time.time() - start_time

path_states=[]
for node in breadth_node.path():
    path_states.append(node.state)

lat_list=[world_map.locations[state][0] for state in path_states ]
lon_list=[world_map.locations[state][1] for state in path_states ]

path_fig=draw_path(lat_list,lon_list)

algorithms_elapsed_times = {'breadth first':elapsed_time}
search_algorthms={'depth first': depth_first_graph_search,
                  'depth limited': depth_limited_search,
                  'iterative deepening': iterative_deepening_search,
                   'uniform cost': uniform_cost_search,
                    'greedy': greedy_best_first_graph_search,
                    'A*': astar_search}
for name,search in search_algorthms.items():
    start_time = time.time()
    if search not in [greedy_best_first_graph_search,astar_search]:
        search_node=search(world_problem)
    else:
        search_node=search(world_problem,world_problem.h)
    elapsed_time = time.time() - start_time
    algorithms_elapsed_times.update({name: elapsed_time})

    path_states=[]
    for node in search_node.path():
        path_states.append(node.state)

    lat_list=[world_map.locations[state][0] for state in path_states ]
    lon_list=[world_map.locations[state][1] for state in path_states ]

    path_fig=add_trace_path(path_fig,lat_list,lon_list,name)
```



6.2- Execution Time:

```
In [53]: fig_algorithms_elapsed_times = px.bar(
             x=list(algorithms_elapsed_times.keys()),
             y=list(algorithms_elapsed_times.values()),
             color=list(algorithms_elapsed_times.keys()),
             title="Algorithms Execution Elapsed Time",
             labels={"x": "Algorithm", "y": "Time in seconds"},
         )
         fig_algorithms_elapsed_times.show()
```

this code snippet efficiently creates and displays a bar chart using **plotly** visualizing the execution times of various algorithms based on the provided **algorithms_elapsed_times dictionary**. Eac bar in the chart represents an algorithm, with its height indicating the elapsed time taken by that algorithm. The chart provides a clear and intuitive visualization of algorithm performance**.**

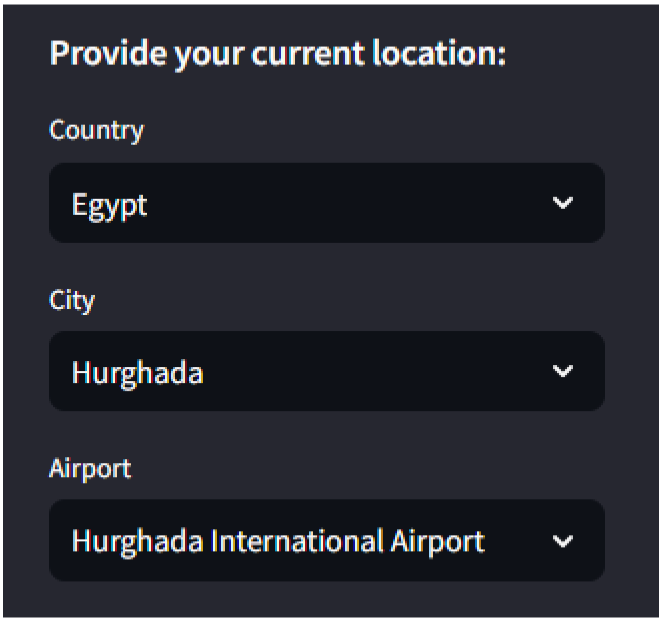


7- Visualization of Results on Web Application:

7.1-Filters:

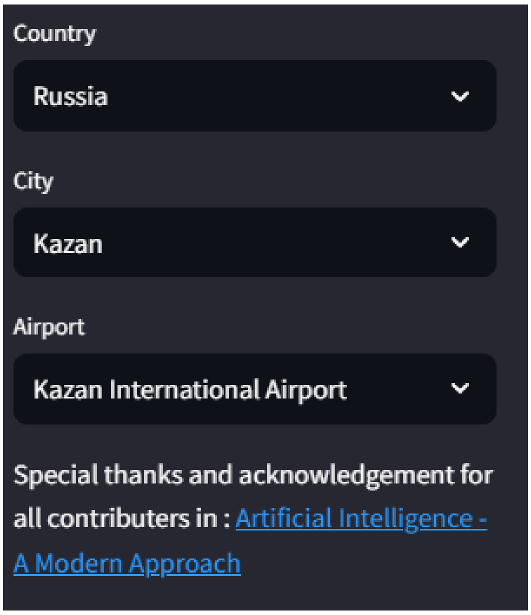
7.1.1- current location:

- Visit our web Application: https://optimized-flight-trip.streamlit.app/.
- First, you’ve to choose from the list and provide your current location information which contains: Airport, City and Country.



7.1.2- destination information:

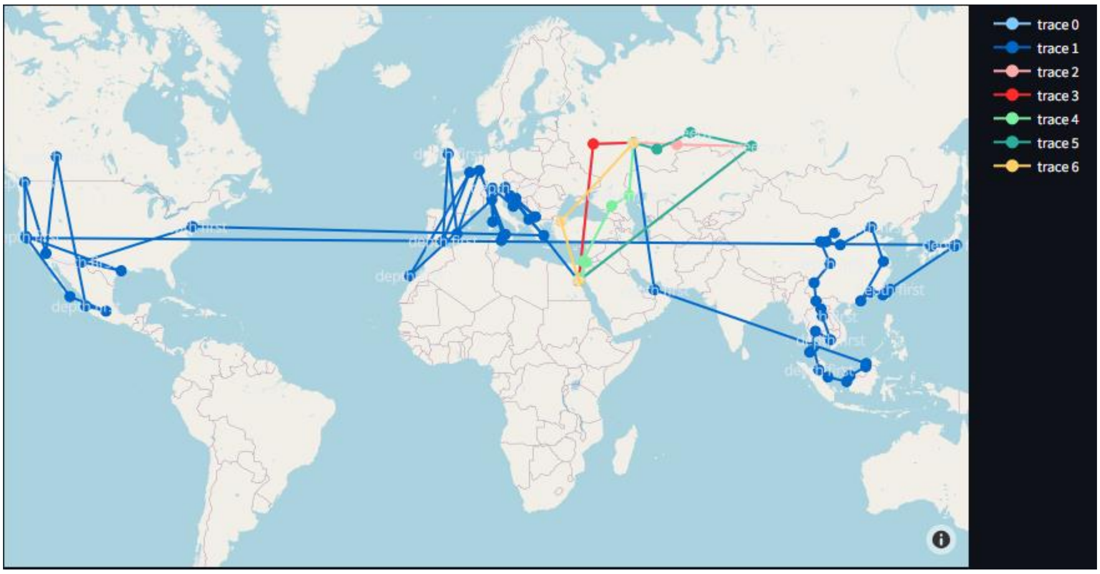
- Choose your destination information from the list.



7.1-Result:

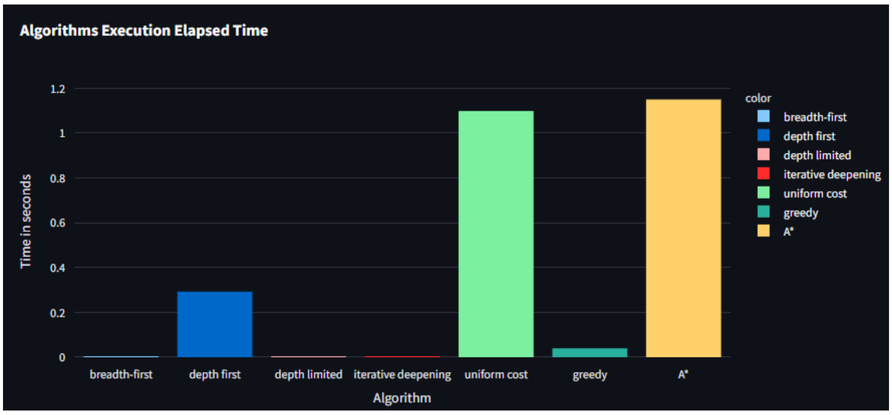
7.2.1- Optimized Flight paths:

- Here’s a map containing all routes using all searching Algorithms to choose the shortest route which suitable for you.



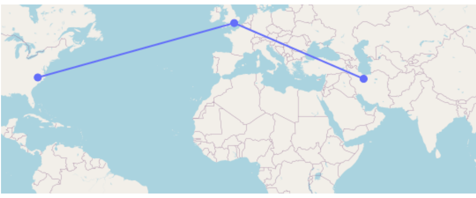
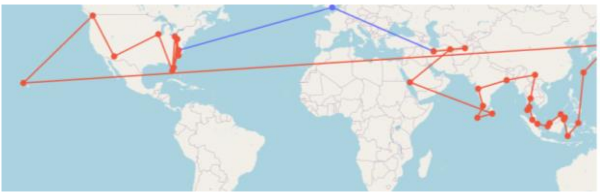
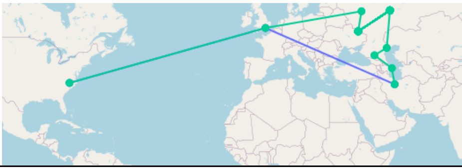
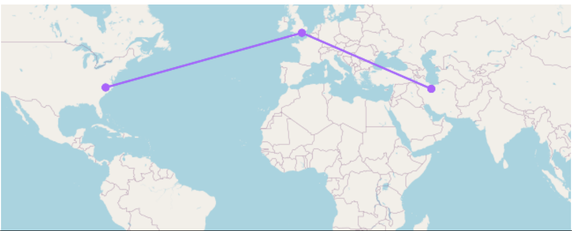
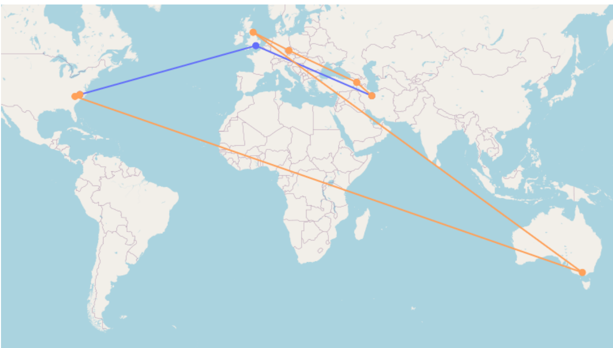
7.2.2- Algorithms Execution Elapsed Time:

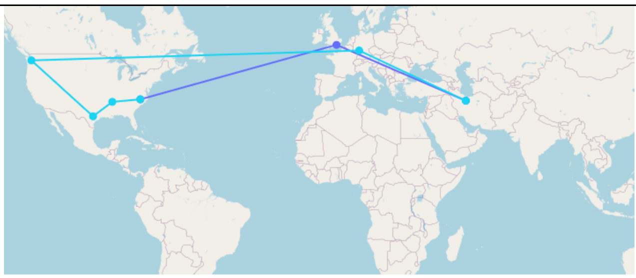
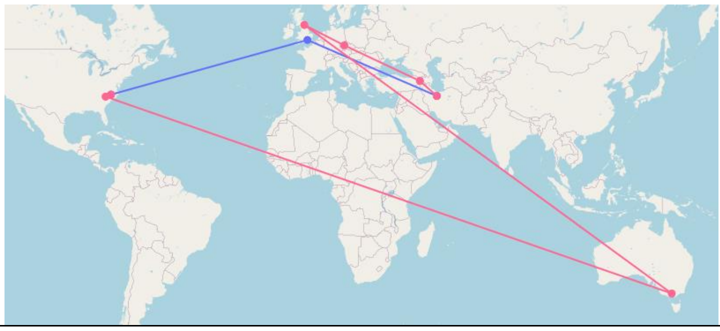
- The following figure shows the Algorithms’ elapsed time, and we can conclude that the algorithm which succeeds to get the best results and optimized flight trip doesn’t have to be the fastest one as we can see A* Algorithm expands more nodes; in this case it expands 1765 nodes and it also check heuristic and path cost which takes more time.



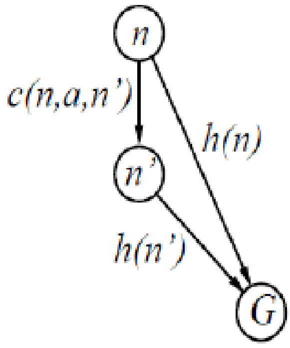
7.2.3Discussion:

- User can choose the preferred path according to the map.

Algorithm	Path
BFS	
DFS	
DLS	
IDS	
UCS	

Greedy	
A*	

- **To get A* optimal solution: consistency condition must be satisfied.**



- **More features can be added to upgrade our project as: Fly Time – Cost……etc. it can also be updated to show the best algorithm after checking optimality for each one of them according to the following table.**

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes*	Yes*	No	Yes, if $l \geq d$	Yes
Time	b^{d+1}	$b^{\lceil 1+C^*/\epsilon \rceil}$	b^m	b^l	b^d
Space	b^{d+1}	$b^{\lceil 1+C^*/\epsilon \rceil}$	bm	bl	bd
Optimal?	Yes*	Yes	No	No	Yes*

8- Appendices:

[1]-Data set:

https://drive.google.com/file/d/1-8ykSx-2Iqwt9fVSaE5e3du1I1fls_oZ/view?usp=drive_link

[2]-Our Source Code:

https://drive.google.com/file/d/1-yyRYkIqZig30rbDA5E807nVlxtxgAWR/view?usp=sharing

[4]-Our DEMO:

https://drive.google.com/file/d/13Ays0C-H6OOhzNKDnPVTWXaRrXYJmEbP/view?usp=sharing

[3]-Our web Application:

https://optimized-flight-trip.streamlit.app/

10-References:

[1]- https://aima.cs.berkeley.edu/

[2]- https://github.com/aimacode

Name	Contribution Percentage	Signature
Ahmed Mohamed Fawzy.	25% of Coding 25% of Report. 25% of Presentation. Poster & DEMO.	
Passant El-Tonsy Ali.	25% of Coding 25% of Report. 25% of Presentation. Poster & DEMO.	
Saif Emad ElDeen Abd-Elkareem.	25% of Coding 25% of Report. 25% of Presentation. Poster & DEMO.	
Mohamed Emad Fawzy.	25% of Coding 25% of Report. 25% of Presentation. Poster & DEMO.	