University: Isfahan University - Faculty of Computer Engineering

Course: Fundamentals and Applications of Artificial Intelligence

Finding the Best Flight Route Using Dijkstra and A* Algorithms

Course Instructor: Dr. Hossein Karshenas

Student Name: **Sheida Abedpour**

Student ID: 4003623025

2023 Nov

Introduction:

This project focuses on finding the best flight route using the Dijkstra and A* algorithms. The objective is to optimize flight planning by minimizing travel time, price and duration of the flight . We aim to compare the performance of these algorithms and provide a practical solution for flight route optimization. This report presents a detailed explanation of the algorithms, their implementation, and the results obtained. By achieving our objectives, we contribute to the field of aviation with an automated and efficient approach to finding optimal flight routes.

Project Objective:

The primary objective of this project is to analyze and compare the performance of the Dijkstra and A* algorithms in determining the most efficient flight route. The Dijkstra algorithm, a well-known shortest path algorithm, utilizes a breadth-first search approach to explore all possible paths in a weighted graph. On the other hand, the A* algorithm combines the Dijkstra algorithm with a heuristic function, which effectively guides the search towards the most promising paths, reducing the overall computational complexity.

Problem Overview:

The problem we address involves considering various factors such as distance, flight duration, and fly time, among others, to determine the optimal route. By incorporating real-world data, including flight duration, cost ansd distance between airports, we aim to provide a comprehensive and practical solution for flight route optimization.

Throughout this report, we will present a detailed explanation of the Dijkstra and A* algorithms, describe their implementation in our project, and discuss the results obtained from their execution.

Dijkstra Algorithm Explanation

The algorithm was developed and published by Edsger W. Dijkstra, a prominent software engineer and computer scientist. In 1959, he published a 3-page paper titled "A note on two problems in connexion with graphs" introducing this algorithm.

Dijkstra's algorithm is a graph traversal algorithm used to find the shortest path between two nodes in a weighted graph. It guarantees finding the shortest path if all edge weights are non-negative. The algorithm starts from a source node and iteratively explores the neighboring nodes, updating the shortest distance to reach each node along the way.

Here's a step-by-step explanation of Dijkstra's algorithm:

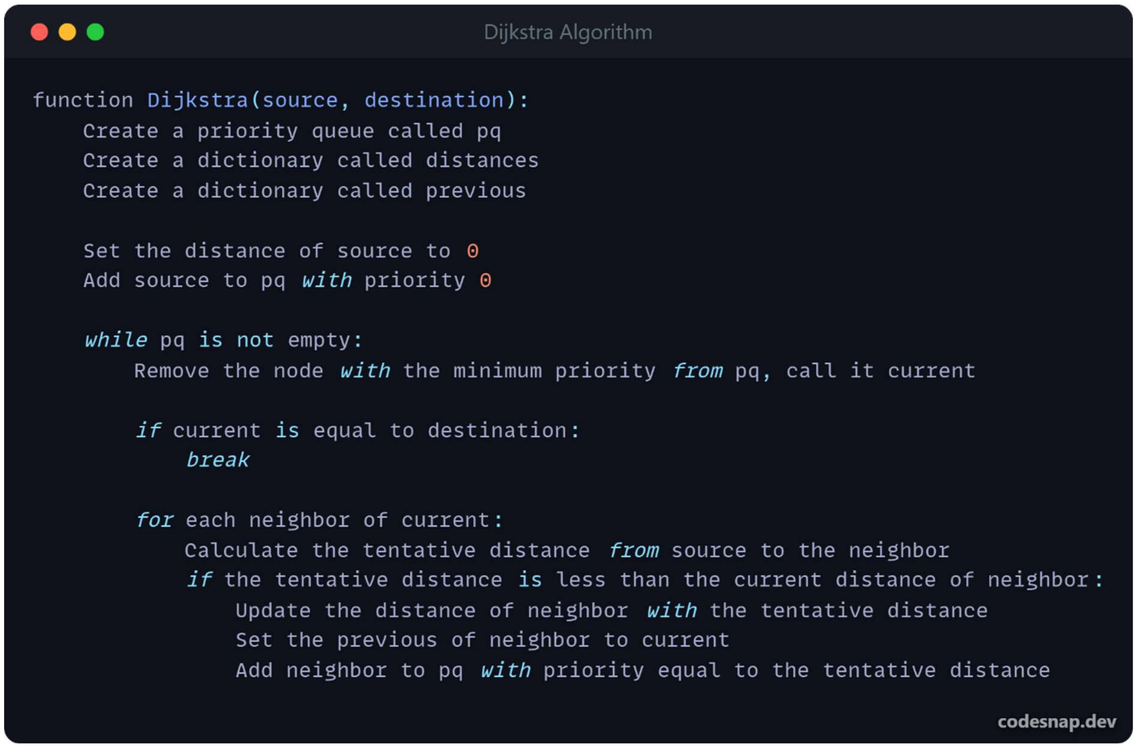
1. Initialize the algorithm:
    - Create a priority queue to to store the cost from the source node to each node, and it is sorted in ascending order based on the minimum cost. This allows the algorithm to efficiently select the node with the minimum cost at each iteration.
    - Creatt a dictionary to store the cost of the source node to others. Set cost 0 for source node and infinity for all other nodes.
    - Creat a dictionary to track the periouse nodes in path.
2. Explore neighbors:
    - For the node with the minimum cost, consider all its neighboring nodes. Pop the current node from priority queue.
    - Calculate the tentative cost from the source node to each neighbor through the current node.
    - If the tentative cost is less than the current recorded cost for the neighbor, update the neighbor's cost with the tentative.
    - Add new distance of neighbors in priority queue.
    - Update the pervious node of neighbor with lower cost.
3. Select the next node:
    - Among the pririty queue, select the node with the smallest distance from the source node as the next current node.

4. Repeat steps 2-3:
    - Repeat steps 2 and 3 until priority queue is empy or the destination nodehas been visited.
5. Backtrack to find the shortest path:
    - Once the priority queue is empty or the destination node has been visited, you can backtrack from the destination node to the source node using the recorded nodes.

Complexity:

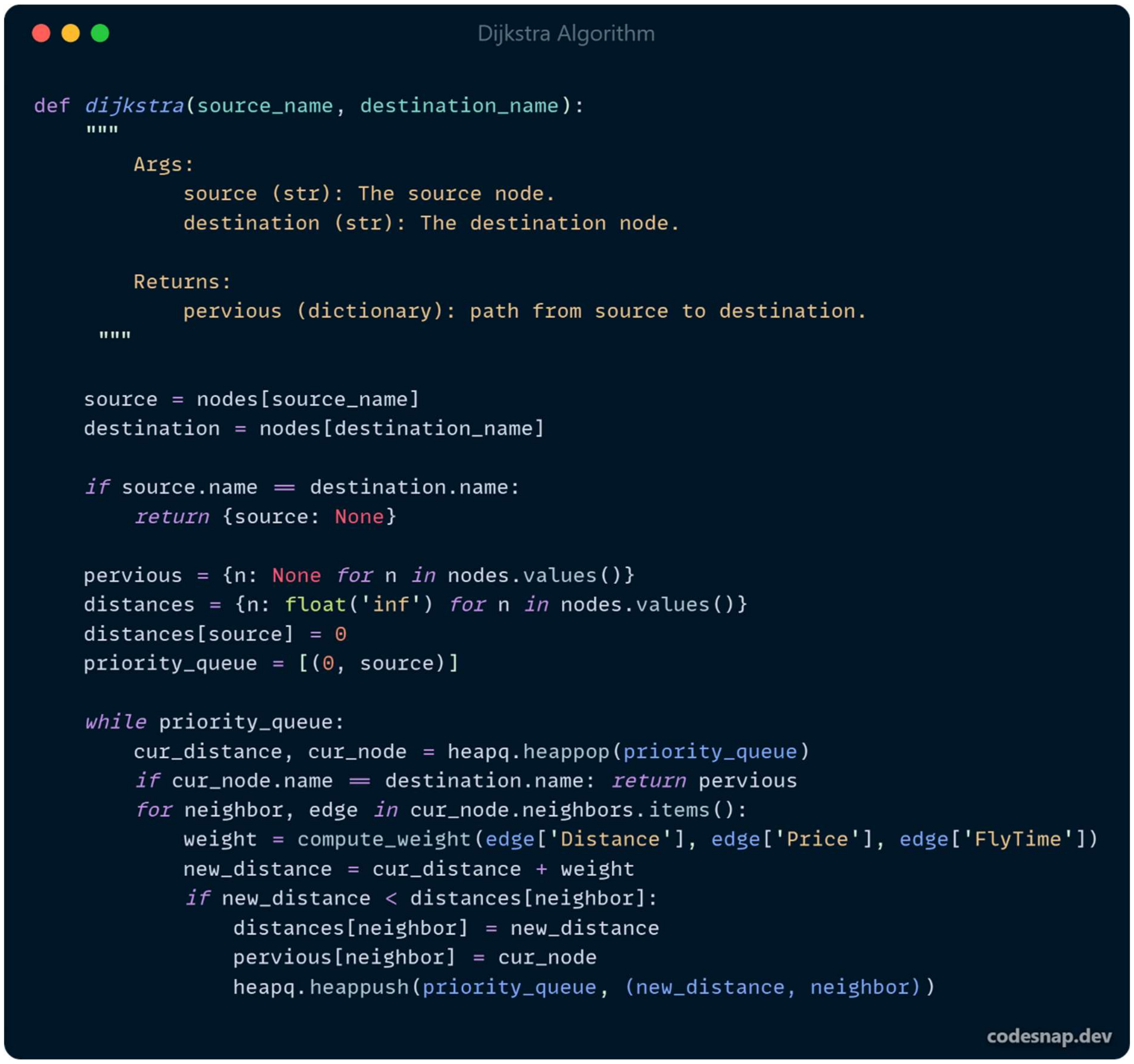
Dijkstra's algorithm has a time complexity of O((V + E) log V), where V is the number of vertices and E is the number of edges in the graph.

Pseudo-code of Dijkstra's algorithm:

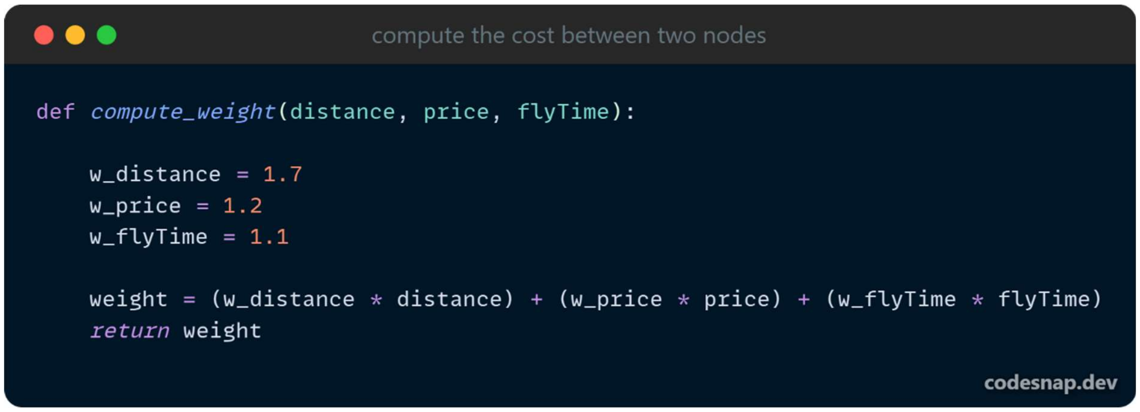


```
function Dijkstra(source, destination):
    Create a priority queue called pq
    Create a dictionary called distances
    Create a dictionary called previous

    Set the distance of source to 0
    Add source to pq with priority 0

    while pq is not empty:
        Remove the node with the minimum priority from pq, call it current

        if current is equal to destination:
            break

        for each neighbor of current:
            Calculate the tentative distance from source to the neighbor
            if the tentative distance is less than the current distance of neighbor:
                Update the distance of neighbor with the tentative distance
                Set the previous of neighbor to current
                Add neighbor to pq with priority equal to the tentative distance
```

Implementation of Dijkstra Algorithm:



```python
def dijkstra(source_name, destination_name):
    """
        Args:
            source (str): The source node.
            destination (str): The destination node.

        Returns:
            pervious (dictionary): path from source to destination.
     """

    source = nodes[source_name]
    destination = nodes[destination_name]

    if source.name == destination.name:
        return {source: None}

    pervious = {n: None for n in nodes.values()}
    distances = {n: float('inf') for n in nodes.values()}
    distances[source] = 0
    priority_queue = [(0, source)]

    while priority_queue:
        cur_distance, cur_node = heapq.heappop(priority_queue)
        if cur_node.name == destination.name: return pervious
        for neighbor, edge in cur_node.neighbors.items():
            weight = compute_weight(edge['Distance'], edge['Price'], edge['FlyTime'])
            new_distance = cur_distance + weight
            if new_distance < distances[neighbor]:
                distances[neighbor] = new_distance
                pervious[neighbor] = cur_node
                heapq.heappush(priority_queue, (new_distance, neighbor))
```



```python
def compute_weight(distance, price, flyTime):

    w_distance = 1.7
    w_price = 1.2
    w_flyTime = 1.1

    weight = (w_distance * distance) + (w_price * price) + (w_flyTime * flyTime)
    return weight
```

A* Algorithm Explanation:

The A* (pronounced "A-star") algorithm is a popular pathfinding algorithm that combines the benefits of Dijkstra's algorithm and greedy best-first search. It finds the shortest path between a given start node and a goal node in a graph or a grid with weighted edges. The A* algorithm uses a heuristic function to guide its search towards the goal, making it more efficient than Dijkstra's algorithm.

Here's a step-by-step breakdown of the A* algorithm:

1. Initialize the algorithm:
    - Create a priority queue to store the final cost from the source node to each node, and it is sorted in ascending order based on the minimum cost. This allows the algorithm to efficiently select the node with the minimum cost at each iteration.
    - Create a dictionary called g to store the heuristic cost of the source node to others. Set cost 0 for source node and infinity for all other nodes.
    - Create a dictionary called f to store the final cost(cost from source to current node + heuristic from current node to destination) of the source node to others. Calculate heuristic for source node.
    - Creat a dictionary to track the periouse nodes in path.
2. Explore neighbors:
    - For the node with the minimum final cost, consider all its neighboring nodes. Pop the current node from priority queue.
    - Calculate the tentative cost(cost from source to neighbor + heuristic from neighbor to destination).
    - If the tentative cost is less than the current recorded cost in f for the neighbor, update the neighbor's cost with the tentative.
    - Add new distance of neighbors in priority queue.
    - Update the pervious node of neighbor with lower cost.
3. Select the next node:

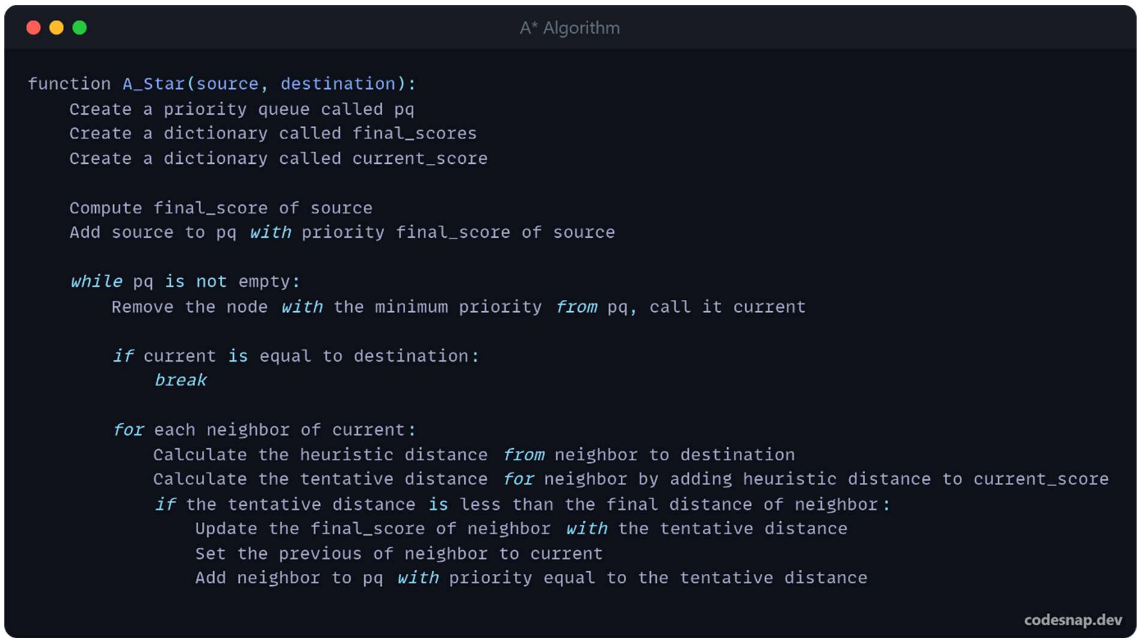
- Among the pririty queue, select the node with the smallest final cost from the source node as the next current node.

4. Repeat steps 2-3:
    - Repeat steps 2 and 3 until priority queue is empy or the destination nodehas been visited.
5. Backtrack to find the shortest path:
    - Once the priority queue is empty or the destination node has been visited, you can backtrack from the destination node to the source node using the recorded nodes.

Complexity:

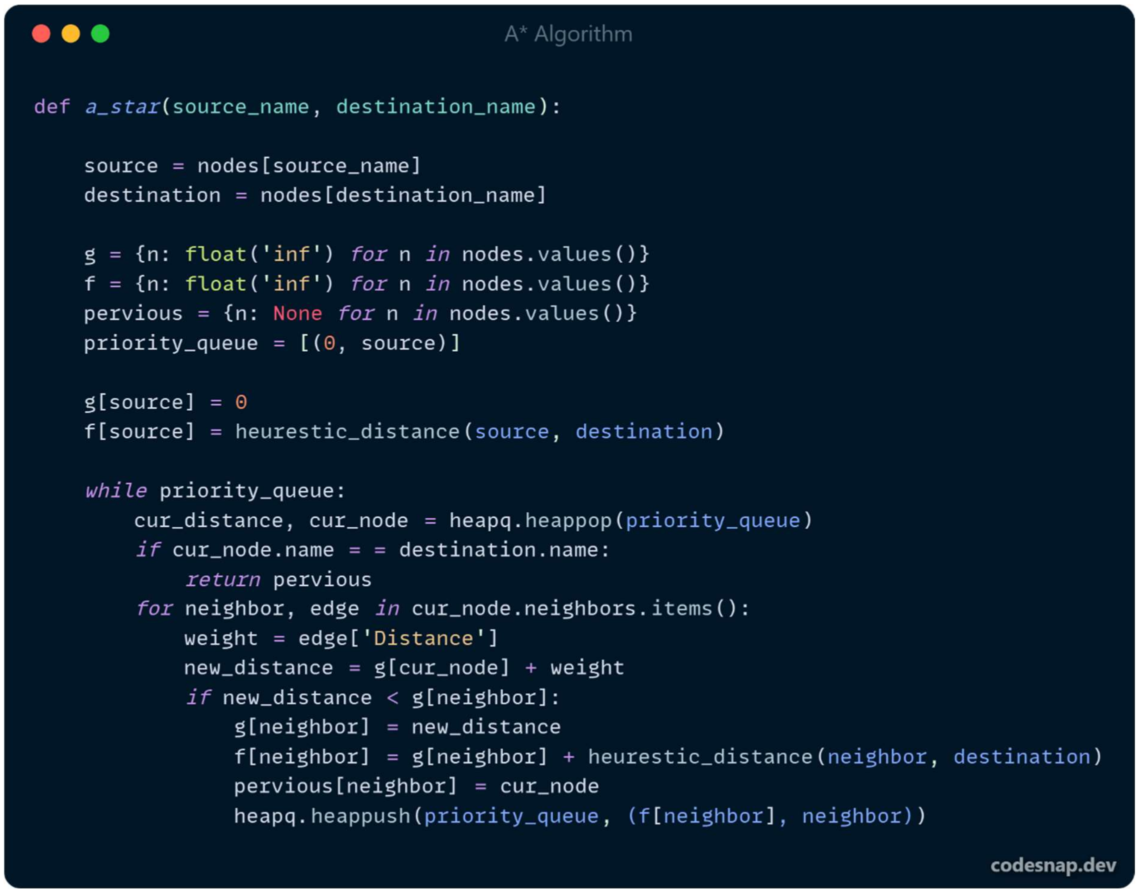
The time complexity of A* algorithm is similar to Dijkstra's algorithm, with a worst-case time complexity of O((V + E) log V). However, the actual performance can be significantly better in practice due to the use of heuristic guidance. If the heuristic is well-designed and admissible, A* can often explore fewer nodes and converge to the goal more efficiently.

Pseudo-code of A* algorithm:

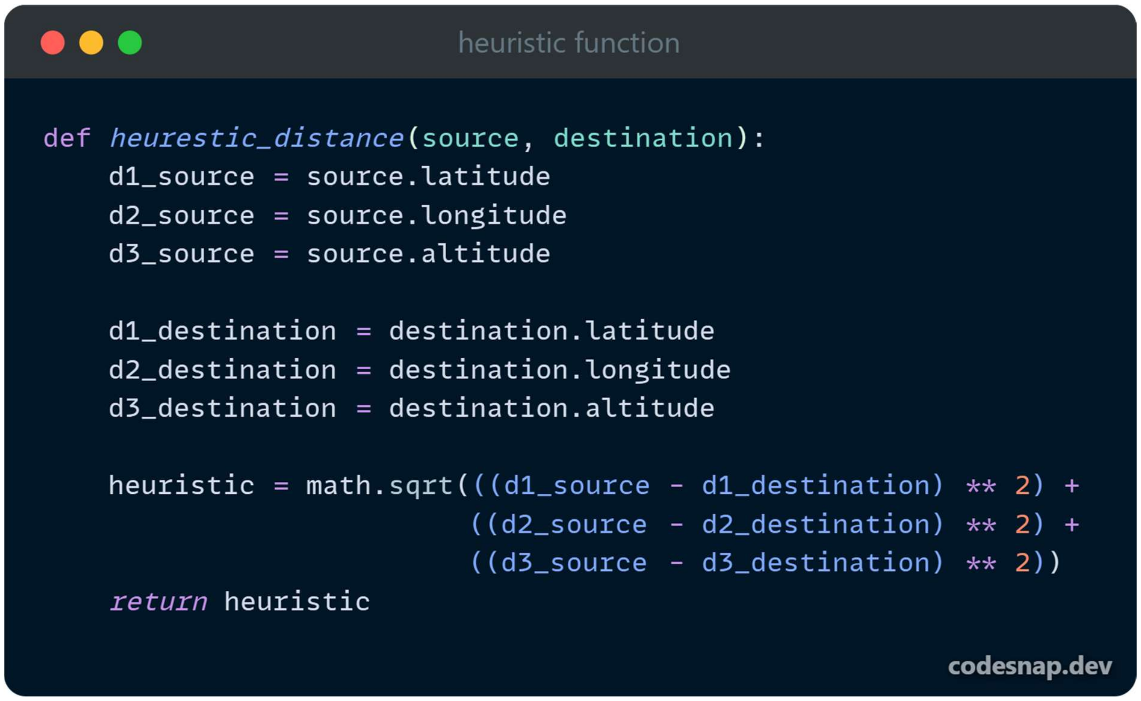


```
function A_Star(source, destination):
    Create a priority queue called pq
    Create a dictionary called final_scores
    Create a dictionary called current_score

    Compute final_score of source
    Add source to pq with priority final_score of source

    while pq is not empty:
        Remove the node with the minimum priority from pq, call it current

        if current is equal to destination:
            break

        for each neighbor of current:
            Calculate the heuristic distance from neighbor to destination
            Calculate the tentative distance for neighbor by adding heuristic distance to current_score
            if the tentative distance is less than the final distance of neighbor:
                Update the final_score of neighbor with the tentative distance
                Set the previous of neighbor to current
                Add neighbor to pq with priority equal to the tentative distance
```

Implementation of A* algorithm:



```python
def a_star(source_name, destination_name):

    source = nodes[source_name]
    destination = nodes[destination_name]

    g = {n: float('inf') for n in nodes.values()}
    f = {n: float('inf') for n in nodes.values()}
    pervious = {n: None for n in nodes.values()}
    priority_queue = [(0, source)]

    g[source] = 0
    f[source] = heurestic_distance(source, destination)

    while priority_queue:
        cur_distance, cur_node = heapq.heappop(priority_queue)
        if cur_node.name = = destination.name:
            return pervious
        for neighbor, edge in cur_node.neighbors.items():
            weight = edge['Distance']
            new_distance = g[cur_node] + weight
            if new_distance < g[neighbor]:
                g[neighbor] = new_distance
                f[neighbor] = g[neighbor] + heurestic_distance(neighbor, destination)
                pervious[neighbor] = cur_node
                heapq.heappush(priority_queue, (f[neighbor], neighbor))
```



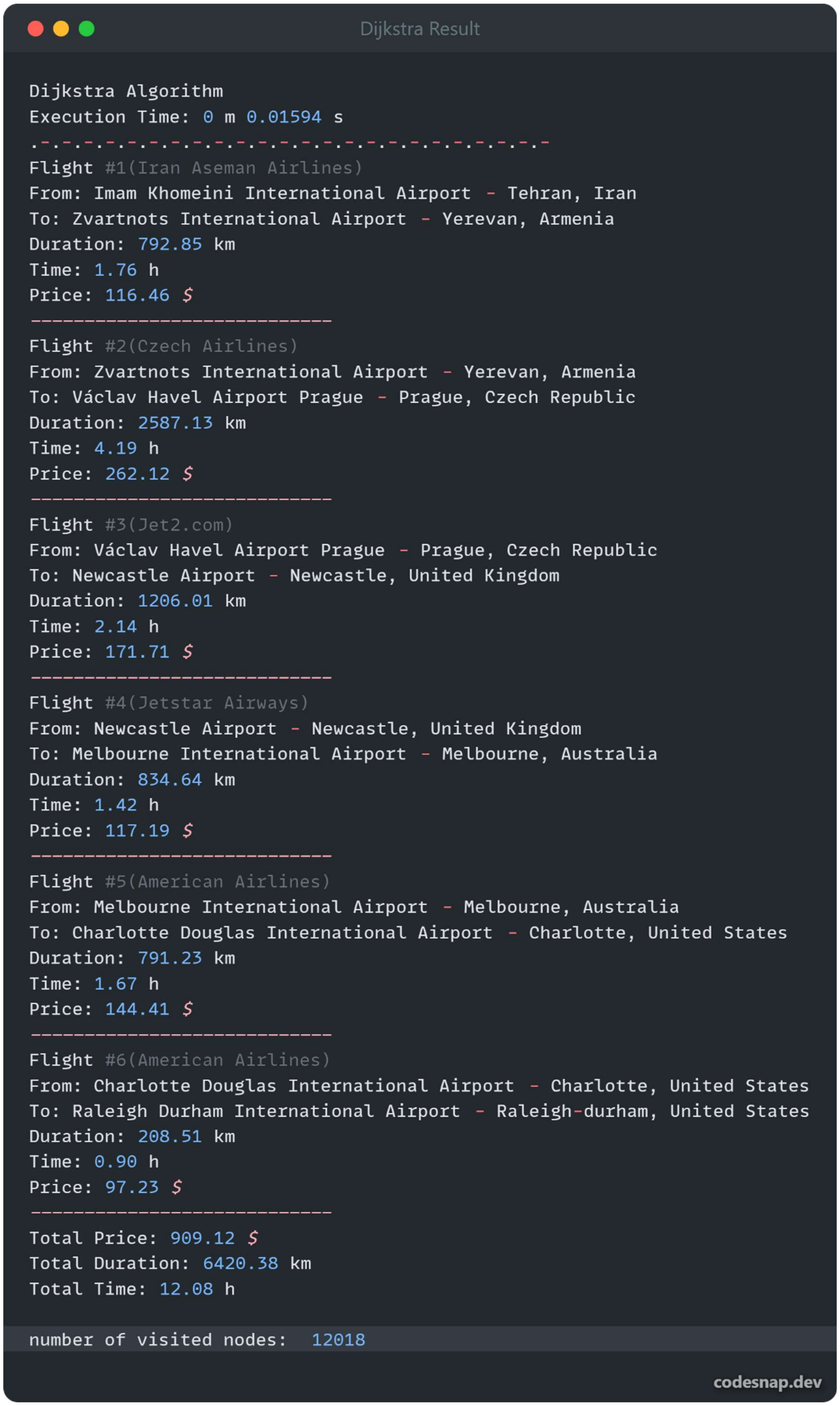
```python
def heurestic_distance(source, destination):
    d1_source = source.latitude
    d2_source = source.longitude
    d3_source = source.altitude

    d1_destination = destination.latitude
    d2_destination = destination.longitude
    d3_destination = destination.altitude

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

Result and analysis:

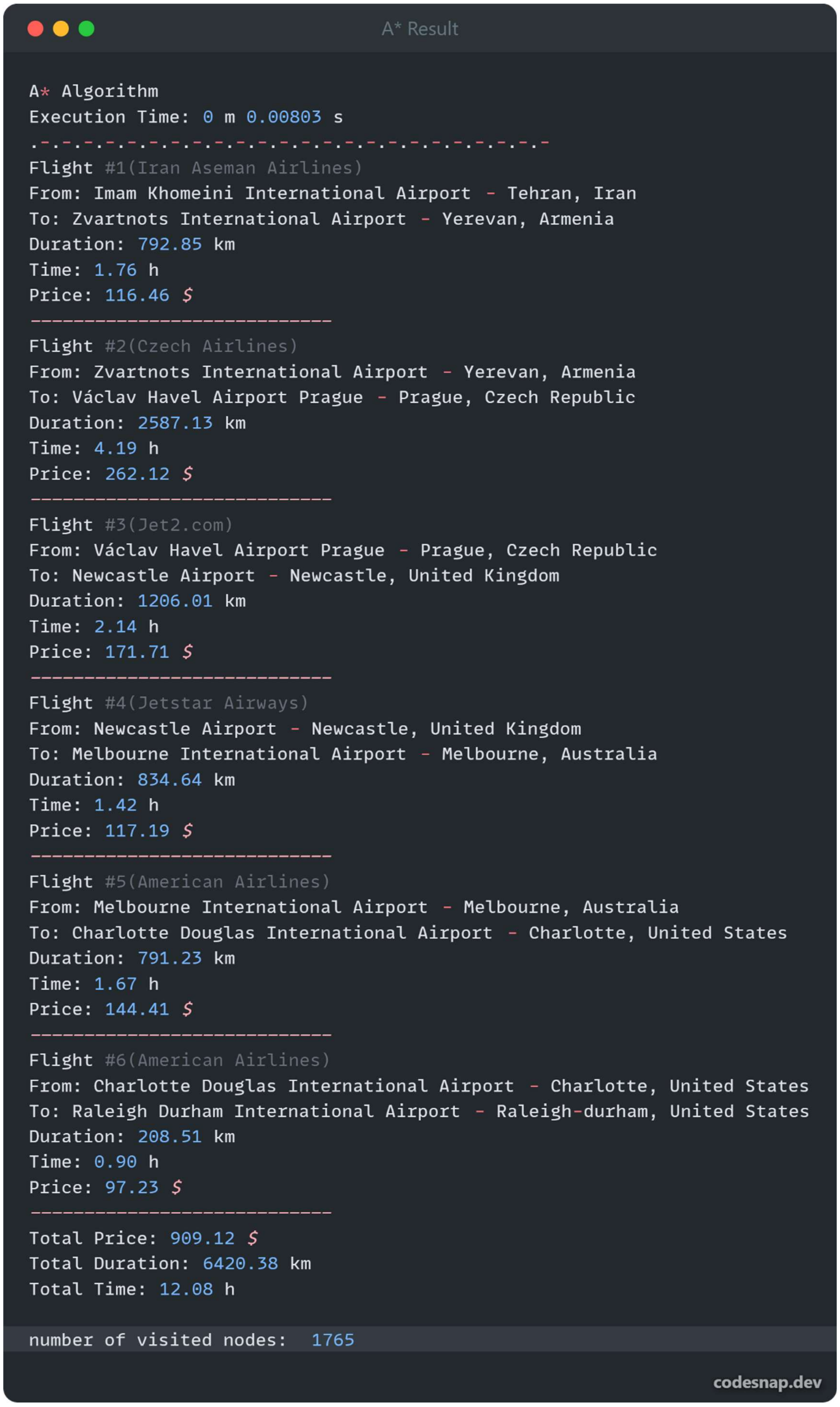
Dijksta and A* algorithms are tested with the same inputs. The results show:

- In terms of the number of nodes visited, the A* algorithm often explores fewer nodes compared to Dijkstra's algorithm. This is because A* algorithm utilizes heuristic information to guide the search towards the goal, effectively pruning away less promising paths. Dijkstra's algorithm explores nodes uniformly, without any knowledge of the goal or the potential cost to reach the goal. It expands nodes in all directions, considering all possible paths until it reaches the destination. As a result, Dijkstra's algorithm can visit more nodes, especially when there are many nodes or complex paths in the graph.
- In general, A* algorithm can be more time-efficient compared to Dijkstra's algorithm, especially when a good heuristic function is used.
- In terms of theoretical worst-case time complexity, both Dijkstra's algorithm and A* algorithm have the same complexity of O((V + E) log V), where V is the number of vertices (nodes) and E is the number of edges in the graph. However, in practice, A* algorithm can be more time-efficient due to its ability to prioritize paths based on the heuristic information.

In summary, Dijkstra's algorithm is a reliable and widely applicable algorithm for finding the shortest path, while A* algorithm offers potential efficiency improvements by incorporating a heuristic function. In this problem I choose A* algorithm because its elapsed time is less and it can find the best flight route with heuristic function well.


Dijkstra Result

```
Dijkstra Algorithm
Execution Time: 0 m 0.01594 s
.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-
Flight #1(Iran Aseman Airlines)
From: Imam Khomeini International Airport - Tehran, Iran
To: Zvartnots International Airport - Yerevan, Armenia
Duration: 792.85 km
Time: 1.76 h
Price: 116.46 $
----------------------------
Flight #2(Czech Airlines)
From: Zvartnots International Airport - Yerevan, Armenia
To: Václav Havel Airport Prague - Prague, Czech Republic
Duration: 2587.13 km
Time: 4.19 h
Price: 262.12 $
----------------------------
Flight #3(Jet2.com)
From: Václav Havel Airport Prague - Prague, Czech Republic
To: Newcastle Airport - Newcastle, United Kingdom
Duration: 1206.01 km
Time: 2.14 h
Price: 171.71 $
----------------------------
Flight #4(Jetstar Airways)
From: Newcastle Airport - Newcastle, United Kingdom
To: Melbourne International Airport - Melbourne, Australia
Duration: 834.64 km
Time: 1.42 h
Price: 117.19 $
----------------------------
Flight #5(American Airlines)
From: Melbourne International Airport - Melbourne, Australia
To: Charlotte Douglas International Airport - Charlotte, United States
Duration: 791.23 km
Time: 1.67 h
Price: 144.41 $
----------------------------
Flight #6(American Airlines)
From: Charlotte Douglas International Airport - Charlotte, United States
To: Raleigh Durham International Airport - Raleigh-durham, United States
Duration: 208.51 km
Time: 0.90 h
Price: 97.23 $
----------------------------
Total Price: 909.12 $
Total Duration: 6420.38 km
Total Time: 12.08 h

number of visited nodes:  12018
```


A* Result

```
A* Algorithm
Execution Time: 0 m 0.00803 s
.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-
Flight #1(Iran Aseman Airlines)
From: Imam Khomeini International Airport - Tehran, Iran
To: Zvartnots International Airport - Yerevan, Armenia
Duration: 792.85 km
Time: 1.76 h
Price: 116.46 $
----------------------------
Flight #2(Czech Airlines)
From: Zvartnots International Airport - Yerevan, Armenia
To: Václav Havel Airport Prague - Prague, Czech Republic
Duration: 2587.13 km
Time: 4.19 h
Price: 262.12 $
----------------------------
Flight #3(Jet2.com)
From: Václav Havel Airport Prague - Prague, Czech Republic
To: Newcastle Airport - Newcastle, United Kingdom
Duration: 1206.01 km
Time: 2.14 h
Price: 171.71 $
----------------------------
Flight #4(Jetstar Airways)
From: Newcastle Airport - Newcastle, United Kingdom
To: Melbourne International Airport - Melbourne, Australia
Duration: 834.64 km
Time: 1.42 h
Price: 117.19 $
----------------------------
Flight #5(American Airlines)
From: Melbourne International Airport - Melbourne, Australia
To: Charlotte Douglas International Airport - Charlotte, United States
Duration: 791.23 km
Time: 1.67 h
Price: 144.41 $
----------------------------
Flight #6(American Airlines)
From: Charlotte Douglas International Airport - Charlotte, United States
To: Raleigh Durham International Airport - Raleigh-durham, United States
Duration: 208.51 km
Time: 0.90 h
Price: 97.23 $
----------------------------
Total Price: 909.12 $
Total Duration: 6420.38 km
Total Time: 12.08 h

number of visited nodes:  1765
```

codesnap.dev

Refrences:

The resources I used in this project include:

- Chat GPT
- baeldung.com
- medium.com
- youtube.com/@LearningOrbis
- youtube.com/@SimplilearnOfficial

Python Libraries used in project:

- Numpy
- Pandas
- Heapq