University of Isfahan

Best Flight Project

Authors:

Melika Shirian

Kianoosh Vadaei

Fundamentals and Applications of Artificial Intelligence

Dr. Hossein Karshenas

2023-11-08

Introduction

In the world of flying, finding the best path between two airports is really important. In this project, we're using two special ways (Dijkstra and A* algorithms) to help us figure out the best routes through a dataset of flights. This dataset includes details about where the flights start and end, how far they travel, how much they cost, and how long they take.

Our goal is to find the smartest way to go between any two airports in this network of flights. This project is important because it can be used to save fuel, plan flights better, and make traveling more convenient for passengers.

We'll explore the Dijkstra and A* algorithms, which are like secret tools that help us find these smart routes. These tools are used in lots of places to solve problems, and we're using them to make flying better.

Join us on this journey as we dig into flight data and find the best paths in the sky, thanks to Dijkstra and A* algorithms.

Background

Pathfinding and route planning are fundamental challenges in the field of computer science and have far-reaching applications in various domains, including aviation. To address these challenges, several algorithms have been developed, and two prominent ones, Dijkstra and A*, have become pillars of pathfinding and optimization.

Dijkstra Algorithm: The Dijkstra algorithm, introduced by Dutch computer scientist Edsger W. Dijkstra in 1956, is a foundational approach to solving the single-source shortest path problem in weighted graphs. It finds the shortest path from a source node to all other nodes in a graph, making it invaluable for route planning. Dijkstra's algorithm is characterized by its simplicity and ability to guarantee the shortest path in non-negative weighted graphs. It has been a cornerstone in the realm of pathfinding and is widely utilized in diverse applications.

A* Algorithm: A* (pronounced as "A-star") is another renowned algorithm that emerged in the late 1960s. Developed by Peter Hart, Nils Nilsson, and Bertram Raphael, A* extends the capabilities of Dijkstra's algorithm by incorporating heuristic functions to guide the search process. This enhancement enables A* to find optimal paths in graphs while considering the estimated cost to reach the goal node. The A* algorithm excels in efficiency and accuracy, making it a compelling choice for route planning in scenarios where time and resource constraints are crucial.

These algorithms, although initially designed for general pathfinding, have found exceptional relevance in the field of aviation. The complexity of flight route planning, encompassing numerous airports, geographical coordinates, pricing, distances, and travel times, demands advanced solutions for route optimization. Dijkstra and A* algorithms offer the aviation industry valuable tools to enhance flight planning, reduce costs, and improve passenger experiences.

Methodology

In this project, the goal was to find the shortest path between two airports with the lowest price and time possible by implementing Dijkstra's and A* algorithms. The dataset used in this project is relatively large, with each record containing origin and destination information such as airport, city, country, geographical coordinates (longitude, latitude, altitude), as well as flight cost, time, and distance between two airports. In summary, each record represents a mapping from the origin airport to the destination airport.

To achieve this, we formed a graph of airports and implemented the mentioned algorithms on them for comparison.

The implementation of Dijkstra's algorithm in this project utilized a min-heap tree and a graph. In each iteration of a loop, we calculated the function f, which is the sum of the costs of traversing the graph up to the current node. For possible moves from the current node to its neighboring nodes, a node was added to the heap tree, and the minimum node was popped. The cost variable was the comparison metric for the heap tree, and we'll delve into it further.

The implementation of A* is similar to Dijkstra's, with the difference that instead of f, the function f + h is calculated each time. Here, h is the heuristic function specific to our problem, which we'll explain later.

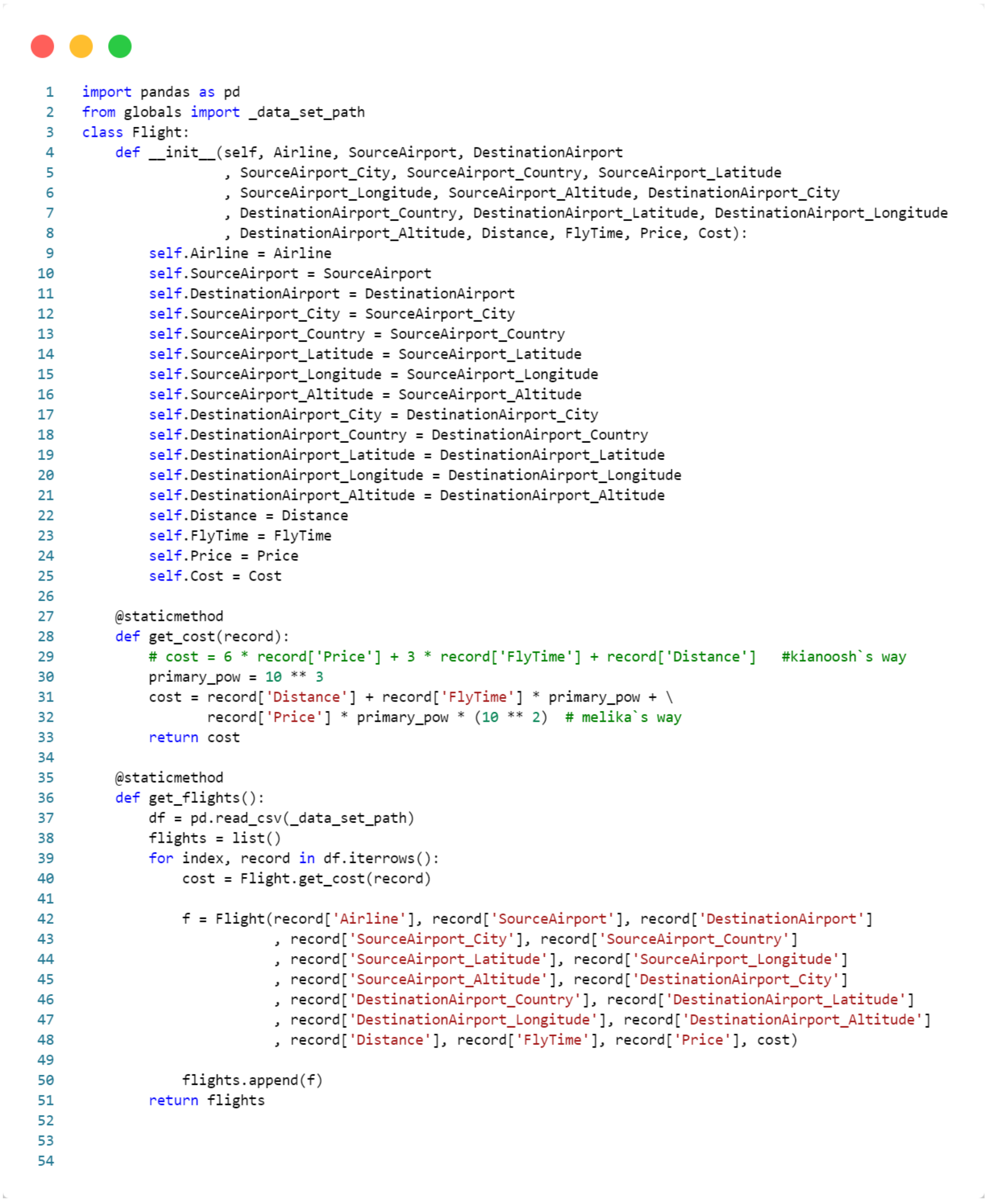
No changes were made to the flight data. The variables were read from the input CSV file using the pandas library, stored in a class exactly mirroring the dataset variables. The cost variable, which we will explain, was added to it.

All significant implementations of this problem were done by us.

Code Explanation

In this project, various modules have been implemented for the implementation of the mentioned algorithms. We will now elaborate on each of them:

Flight Module:



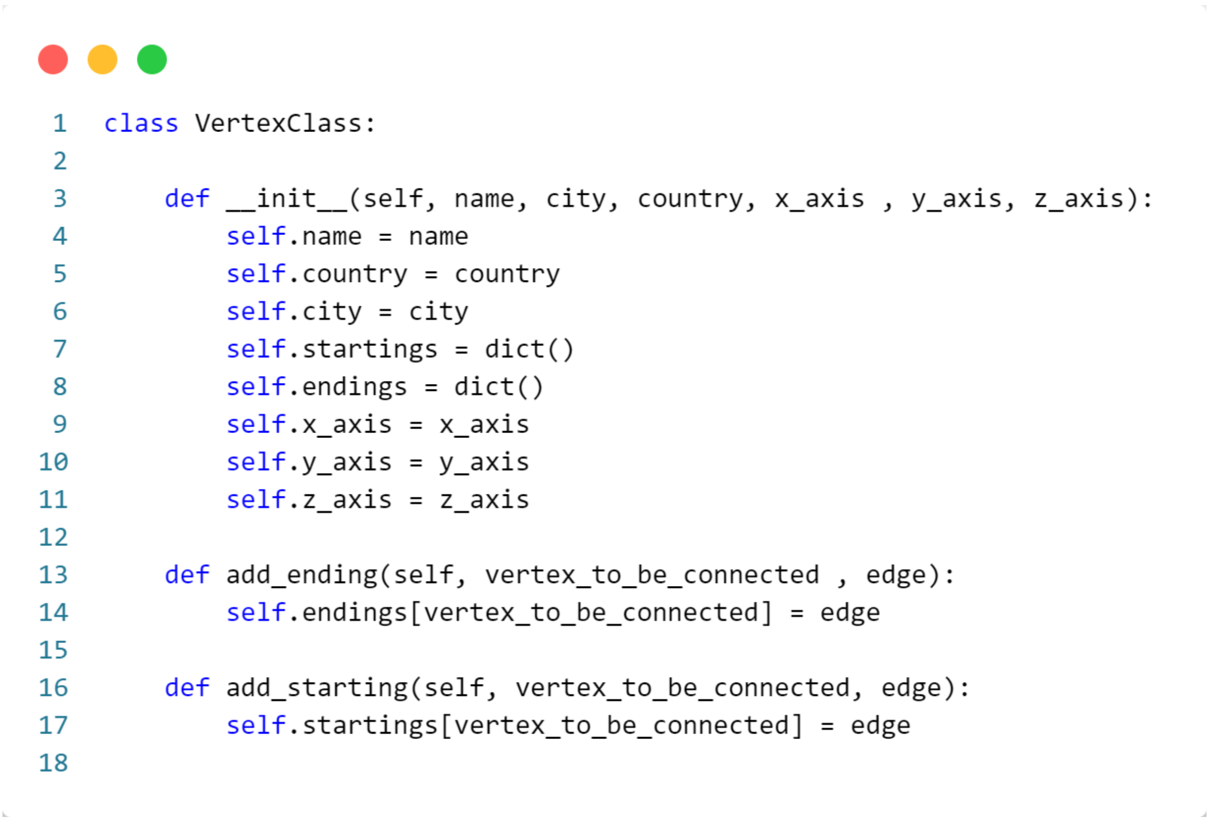
```python
import pandas as pd
from globals import _data_set_path
class Flight:
    def __init__(self, Airline, SourceAirport, DestinationAirport
                 , SourceAirport_City, SourceAirport_Country, SourceAirport_Latitude
                 , SourceAirport_Longitude, SourceAirport_Altitude, DestinationAirport_City
                 , DestinationAirport_Country, DestinationAirport_Latitude, DestinationAirport_Longitude
                 , DestinationAirport_Altitude, Distance, FlyTime, Price, Cost):
        self.Airline = Airline
        self.SourceAirport = SourceAirport
        self.DestinationAirport = DestinationAirport
        self.SourceAirport_City = SourceAirport_City
        self.SourceAirport_Country = SourceAirport_Country
        self.SourceAirport_Latitude = SourceAirport_Latitude
        self.SourceAirport_Longitude = SourceAirport_Longitude
        self.SourceAirport_Altitude = SourceAirport_Altitude
        self.DestinationAirport_City = DestinationAirport_City
        self.DestinationAirport_Country = DestinationAirport_Country
        self.DestinationAirport_Latitude = DestinationAirport_Latitude
        self.DestinationAirport_Longitude = DestinationAirport_Longitude
        self.DestinationAirport_Altitude = DestinationAirport_Altitude
        self.Distance = Distance
        self.FlyTime = FlyTime
        self.Price = Price
        self.Cost = Cost

    @staticmethod
    def get_cost(record):
        # cost = 6 * record['Price'] + 3 * record['FlyTime'] + record['Distance']   #kianoosh`s way
        primary_pow = 10 ** 3
        cost = record['Distance'] + record['FlyTime'] * primary_pow + \
               record['Price'] * primary_pow * (10 ** 2)  # melika`s way
        return cost

    @staticmethod
    def get_flights():
        df = pd.read_csv(_data_set_path)
        flights = list()
        for index, record in df.iterrows():
            cost = Flight.get_cost(record)

            f = Flight(record['Airline'], record['SourceAirport'], record['DestinationAirport']
                       , record['SourceAirport_City'], record['SourceAirport_Country']
                       , record['SourceAirport_Latitude'], record['SourceAirport_Longitude']
                       , record['SourceAirport_Altitude'], record['DestinationAirport_City']
                       , record['DestinationAirport_Country'], record['DestinationAirport_Latitude']
                       , record['DestinationAirport_Longitude'], record['DestinationAirport_Altitude']
                       , record['Distance'], record['FlyTime'], record['Price'], cost)

            flights.append(f)
        return flights
```

By instantiating an object from this class, all relevant fields related to a record are placed in the created object.

By invoking the static function **get_flights**, the program reads the CSV file dataset using the pandas library and stores it in a DataFrame. After that, for each flight in the DataFrame, the cost value, which we will explain, is calculated. A Flight object is then added to the list of flights so that we can work with them in the future.

Note that in the **globals** module, the file path for the dataset is stored.

Vertex Module:



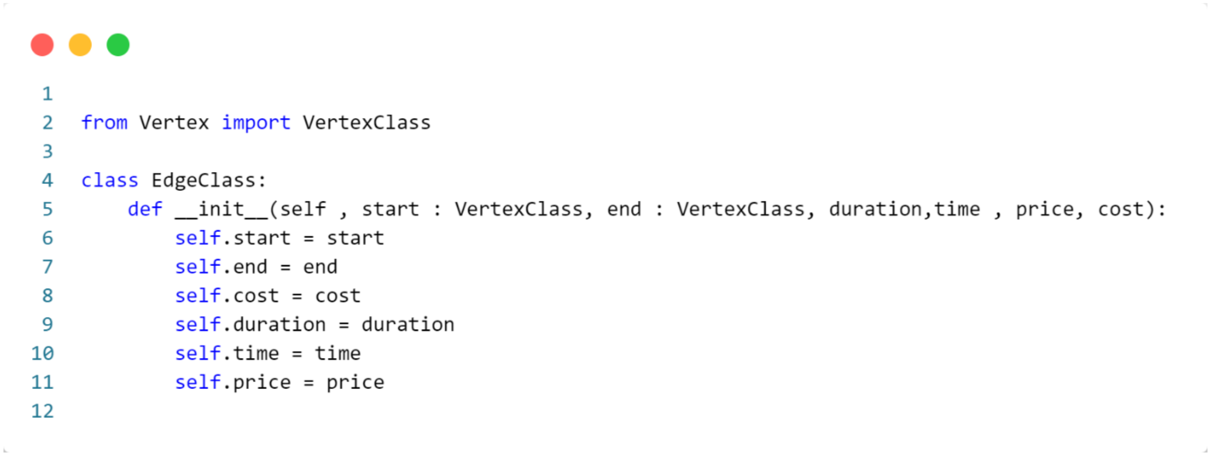
```python
class VertexClass:

    def __init__(self, name, city, country, x_axis , y_axis, z_axis):
        self.name = name
        self.country = country
        self.city = city
        self.startings = dict()
        self.endings = dict()
        self.x_axis = x_axis
        self.y_axis = y_axis
        self.z_axis = z_axis

    def add_ending(self, vertex_to_be_connected , edge):
        self.endings[vertex_to_be_connected] = edge

    def add_starting(self, vertex_to_be_connected, edge):
        self.startings[vertex_to_be_connected] = edge

```

The class Vertex is essentially an implementation for each node in our graph.

Since our graph is a directed graph, each node, in addition to the airport name, city, country, and geographical coordinates, has two dictionaries: **startings** and **endings**. These dictionaries represent the vertices to which this vertex is connected.

In each dictionary, the key is a connected vertex, and the value is the edge between these two vertices.

Edge Module:

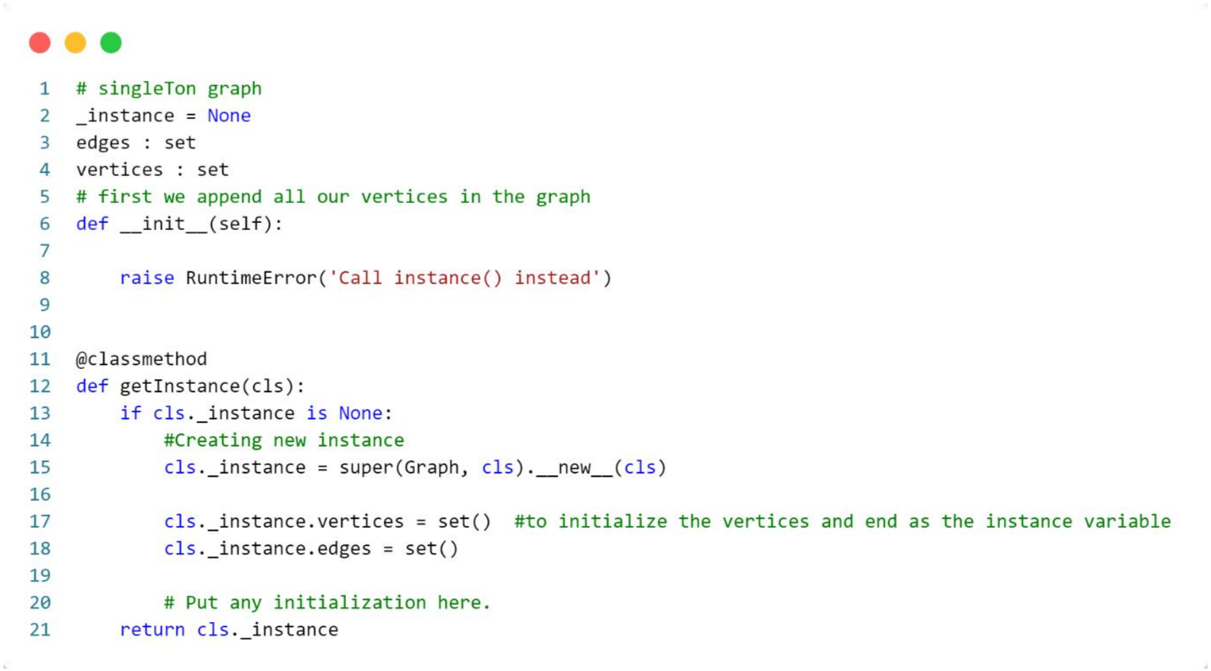


```python

from Vertex import VertexClass

class EdgeClass:
    def __init__(self , start : VertexClass, end : VertexClass, duration,time , price, cost):
        self.start = start
        self.end = end
        self.cost = cost
        self.duration = duration
        self.time = time
        self.price = price

```

The Edge class is also designed to implement an edge between two vertices. The class variables specify the connected vertices and the direction of the edge. Additionally, for implementing Dijkstra's and A* algorithms, cost variables have been incorporated into it.

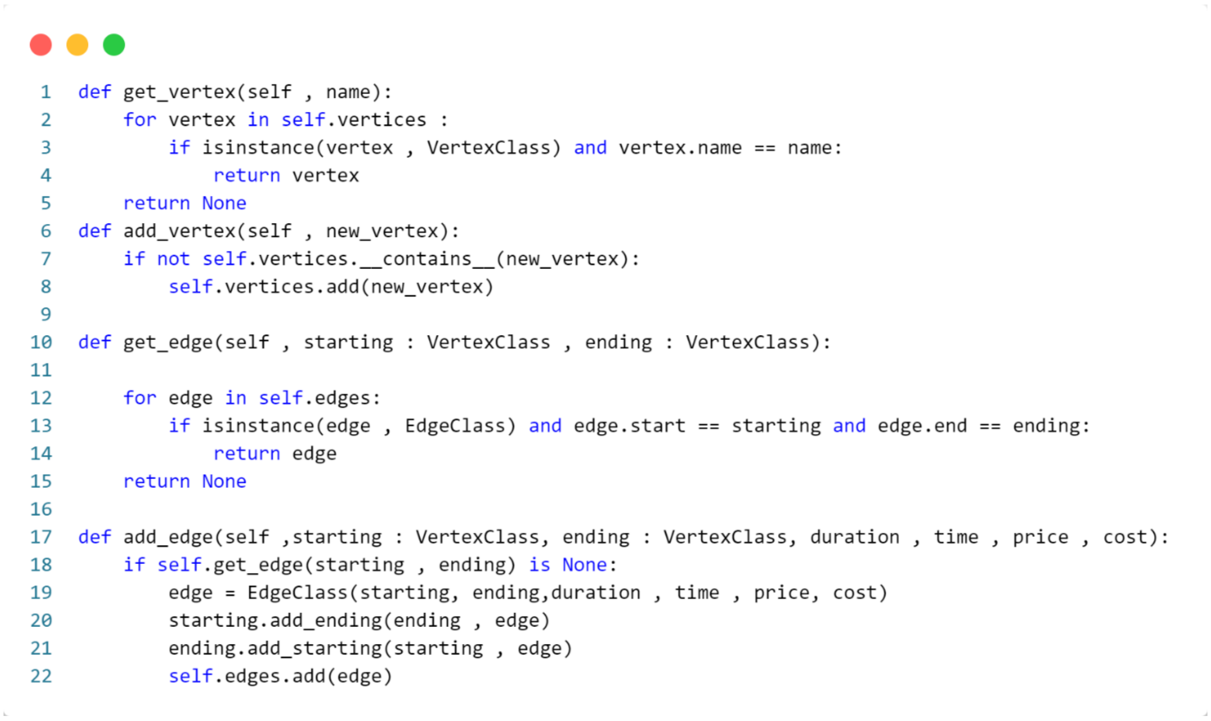
Graph Module:

we will divide the Graph class into several sections for clarification due to its length.



```python
# singleTon graph
_instance = None
edges : set
vertices : set
# first we append all our vertices in the graph
def __init__(self):

    raise RuntimeError('Call instance() instead')


@classmethod
def getInstance(cls):
    if cls._instance is None:
        #Creating new instance
        cls._instance = super(Graph, cls).__new__(cls)

        cls._instance.vertices = set()  #to initialize the vertices and end as the instance variable
        cls._instance.edges = set()

        # Put any initialization here.
    return cls._instance
```

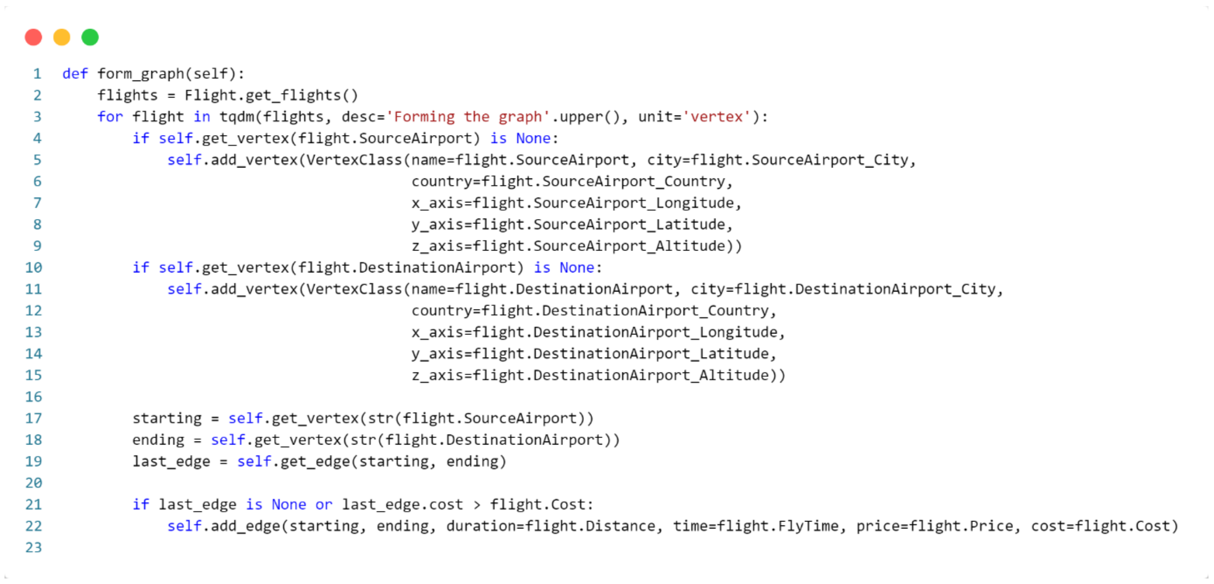
Due to the fact that only one graph is used throughout this project, the Graph class has been implemented using the Singleton design pattern, which you can observe in the code.



```python
def get_vertex(self , name):
    for vertex in self.vertices :
        if isinstance(vertex , VertexClass) and vertex.name == name:
            return vertex
    return None
def add_vertex(self , new_vertex):
    if not self.vertices.__contains__(new_vertex):
        self.vertices.add(new_vertex)

def get_edge(self , starting : VertexClass , ending : VertexClass):

    for edge in self.edges:
        if isinstance(edge , EdgeClass) and edge.start == starting and edge.end == ending:
            return edge
    return None

def add_edge(self ,starting : VertexClass, ending : VertexClass, duration , time , price , cost):
    if self.get_edge(starting , ending) is None:
        edge = EdgeClass(starting, ending,duration , time , price, cost)
        starting.add_ending(ending , edge)
        ending.add_starting(starting , edge)
        self.edges.add(edge)
```

The functions above, in order:

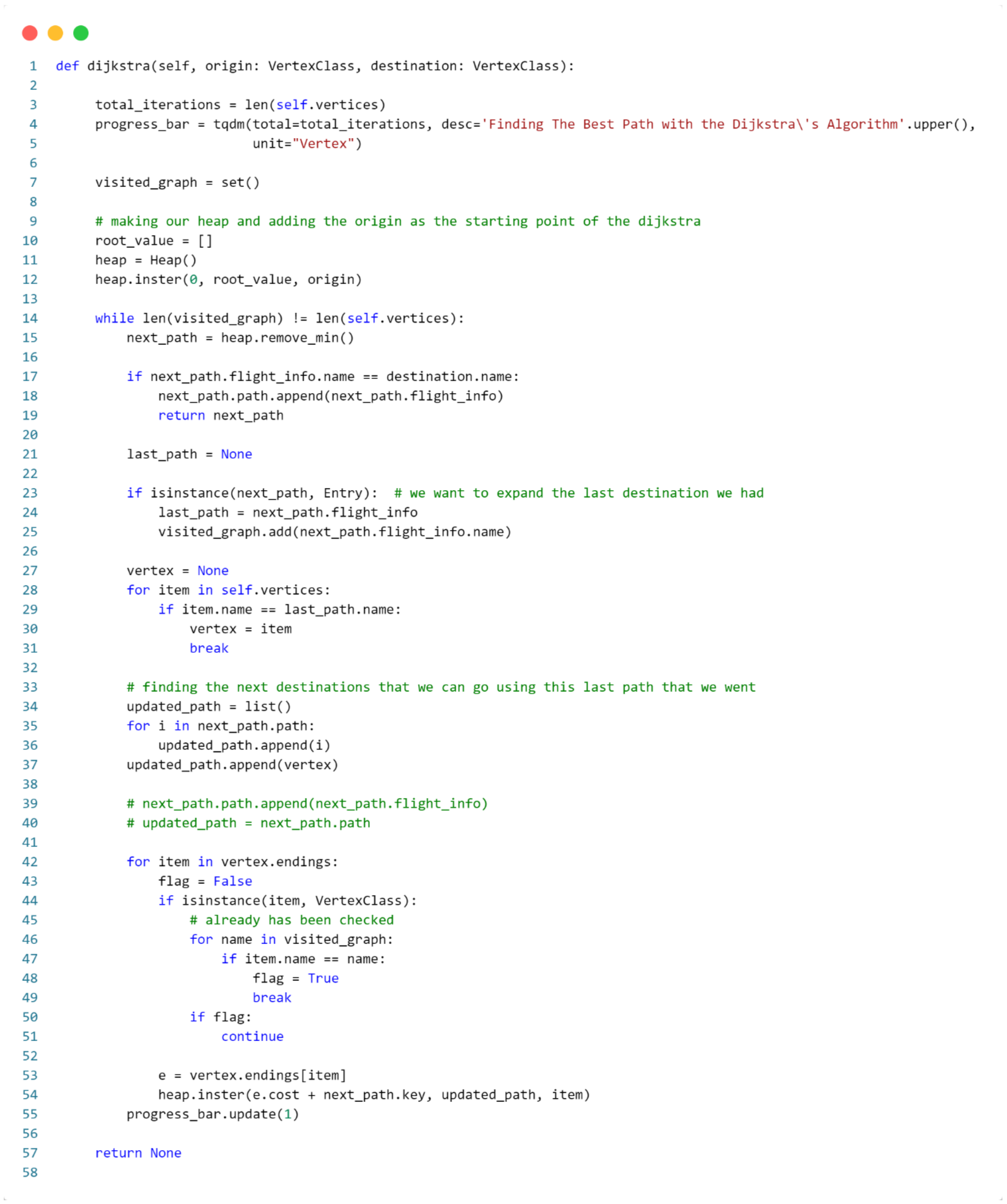
1. The **get_vertex** function finds a vertex based on its airport name from the set of graph vertices.
2. The **add_vertex** function adds a new vertex to the set if it does not already exist.
3. The **get_edge** function, given the start and end vertices, finds the corresponding edge.
4. The **add_edge** function takes start and end vertices and creates an edge with the given cost values. Then, it reciprocally stores the vertex and the corresponding edge in their connected node lists and adds the edge to the set of edges. Note that the vertices given to this function must already exist in the set of graph vertices.



```python
def form_graph(self):
    flights = Flight.get_flights()
    for flight in tqdm(flights, desc='Forming the graph'.upper(), unit='vertex'):
        if self.get_vertex(flight.SourceAirport) is None:
            self.add_vertex(VertexClass(name=flight.SourceAirport, city=flight.SourceAirport_City,
                                        country=flight.SourceAirport_Country,
                                        x_axis=flight.SourceAirport_Longitude,
                                        y_axis=flight.SourceAirport_Latitude,
                                        z_axis=flight.SourceAirport_Altitude))
        if self.get_vertex(flight.DestinationAirport) is None:
            self.add_vertex(VertexClass(name=flight.DestinationAirport, city=flight.DestinationAirport_City,
                                        country=flight.DestinationAirport_Country,
                                        x_axis=flight.DestinationAirport_Longitude,
                                        y_axis=flight.DestinationAirport_Latitude,
                                        z_axis=flight.DestinationAirport_Altitude))

        starting = self.get_vertex(str(flight.SourceAirport))
        ending = self.get_vertex(str(flight.DestinationAirport))
        last_edge = self.get_edge(starting, ending)

        if last_edge is None or last_edge.cost > flight.Cost:
            self.add_edge(starting, ending, duration=flight.Distance, time=flight.FlyTime, price=flight.Price, cost=flight.Cost)

```

In the **form_graph** function, summarily, it reads all records from the list of flights, creates vertices, forms edges between them, and adds them to the graph. It's noteworthy that, since there might be multiple flights between two specific airports, the one with the lower cost is chosen during the graph formation process.

Here, the tqdm library is utilized to display a progress bar while forming the graph.



```python
def dijkstra(self, origin: VertexClass, destination: VertexClass):

    total_iterations = len(self.vertices)
    progress_bar = tqdm(total=total_iterations, desc='Finding The Best Path with the Dijkstra\'s Algorithm'.upper(),
                        unit="Vertex")

    visited_graph = set()

    # making our heap and adding the origin as the starting point of the dijkstra
    root_value = []
    heap = Heap()
    heap.inster(0, root_value, origin)

    while len(visited_graph) != len(self.vertices):
        next_path = heap.remove_min()

        if next_path.flight_info.name == destination.name:
            next_path.path.append(next_path.flight_info)
            return next_path

        last_path = None

        if isinstance(next_path, Entry):  # we want to expand the last destination we had
            last_path = next_path.flight_info
            visited_graph.add(next_path.flight_info.name)

        vertex = None
        for item in self.vertices:
            if item.name == last_path.name:
                vertex = item
                break

        # finding the next destinations that we can go using this last path that we went
        updated_path = list()
        for i in next_path.path:
            updated_path.append(i)
        updated_path.append(vertex)

        # next_path.path.append(next_path.flight_info)
        # updated_path = next_path.path

        for item in vertex.endings:
            flag = False
            if isinstance(item, VertexClass):
                # already has been checked
                for name in visited_graph:
                    if item.name == name:
                        flag = True
                        break
                if flag:
                    continue

            e = vertex.endings[item]
            heap.inster(e.cost + next_path.key, updated_path, item)
        progress_bar.update(1)

    return None

```

In this function, one of the main objectives of this problem is implemented.

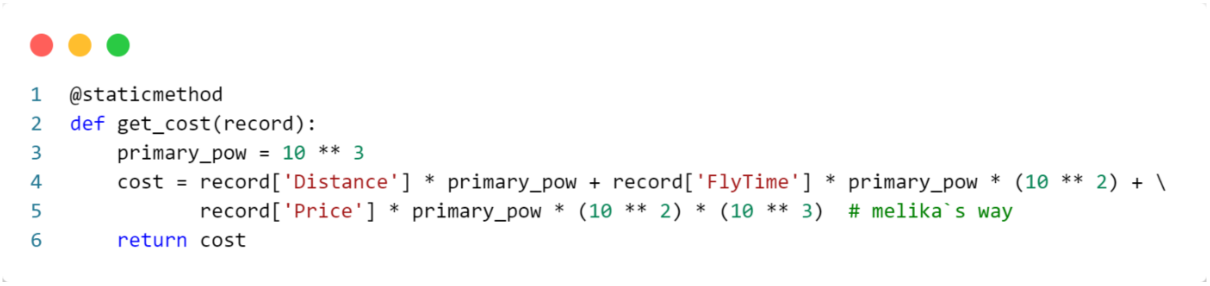
The Dijkstra's function, by taking the source and destination vertices, finds the best path between them as follows:

1. First, the upper limit for iteration in this function is considered, so that, again using the tqdm function, a process bar is displayed during the graph traversal.
2. Then, a set of seen vertices is defined (a set data type is used due to the absence of duplicate nodes).
3. Next, a sample is taken from the heap class (which we will explain later), and a cost of zero for the first node is added to it. Here, the first node includes the source vertex and its cost (note that each node in the heap is an instance of the entry class, which we will explain in the heap section).
4. After that, by entering the loop, we start searching in the graph.

Each time, the node with the minimum cost value (the sum of costs from the source) is popped from the heap and stored in the variable **next_path**. If the node in **next_path** is the destination node, the algorithm finishes by returning the entire path, which is present in the **next_path** field as an entry. Otherwise, the node in this taken path from the heap is added to the **visited_graph** set. Then, this node is found in the list of graph vertices (because in each node in the heap, we don't have the entire node but only its airport name) and stored in the variable **vertex**. After that, the path, which started from the source vertex and was updated with the current vertex each time, is stored in the variable **updated_path** to move to the next iteration.

5. Then, we move on to the nodes connected to the current vertex that we can travel to.
    - If we reach a vertex connected to the current vertex that we have seen before, we move on to the next connected vertex.
    - Eventually, for each connected vertex that we haven't seen before, we calculate the total cost from the source and add it to the heap along with the traversed path. This continues until the algorithm selects a vertex whose total path cost from the source is the minimum possible at each iteration.

- As soon as it reaches a vertex that is the destination, the algorithm stops, and the entire node (entry) containing the traversed path is returned.
- If it fails to find a destination, it returns **None**.



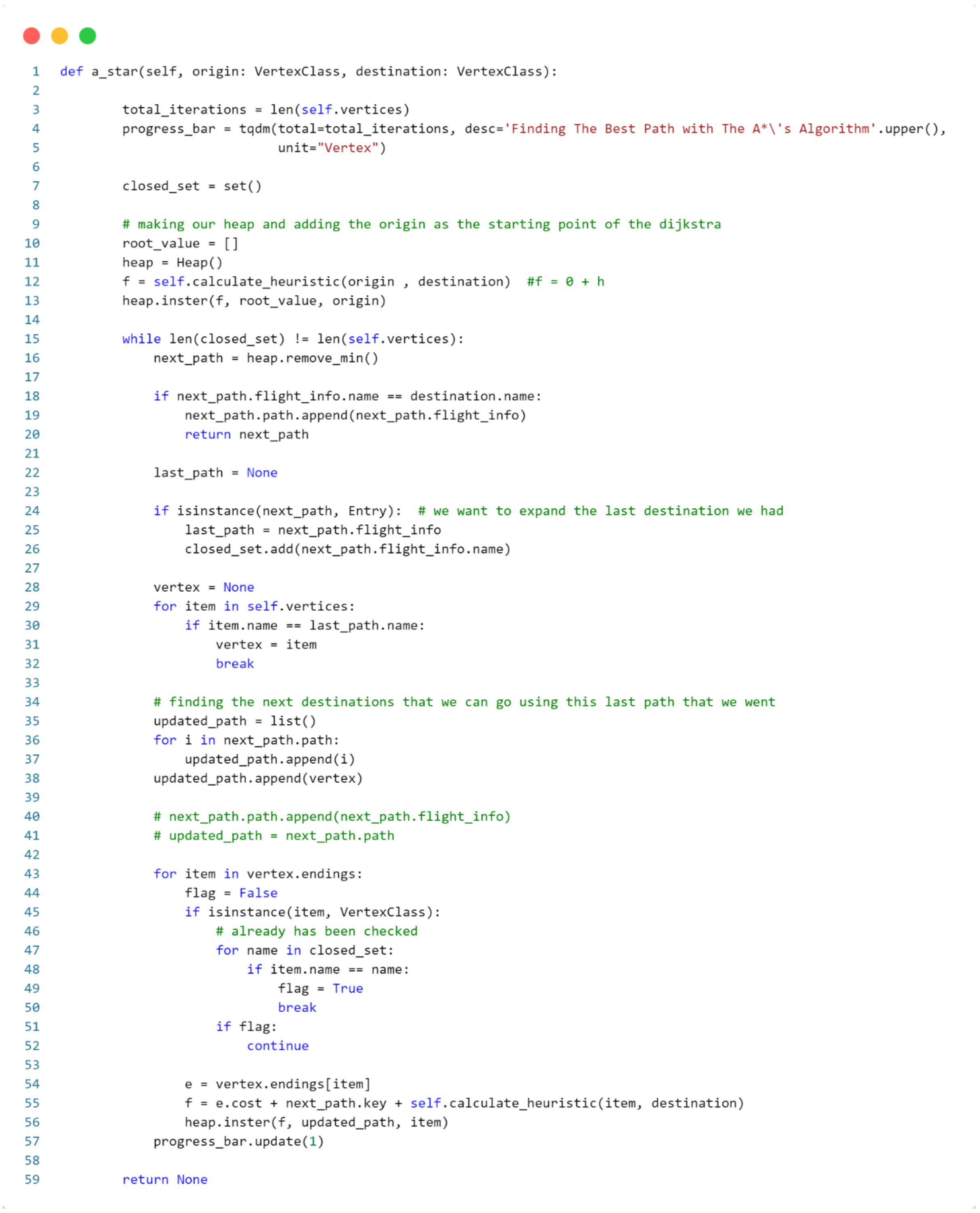
```python
@staticmethod
def get_cost(record):
    primary_pow = 10 ** 3
    cost = record['Distance'] * primary_pow + record['FlyTime'] * primary_pow * (10 ** 2) + \
           record['Price'] * primary_pow * (10 ** 2) * (10 ** 3)  # melika`s way
    return cost
```

The calculated cost for the Dijkstra's algorithm in the above function, located in the Flights module, is determined based on three priority parameters: price, time, and distance.

To obtain a unified value from these three criteria, according to the above function, we first sum the distance criterion with the time criterion multiplied by 10 to the power of 3 (because the maximum distance in the dataset has three digits). Finally, we add the result to the price criterion multiplied by 10 to the power of 3 and further multiplied by 10 to the power of 2 (because the maximum time in the dataset has two digits).

By performing this calculation, we obtain an 8-digit number representing our criteria as follows:

5	6	4	1	6	5	4	3
Price			Fly Time		Distance		

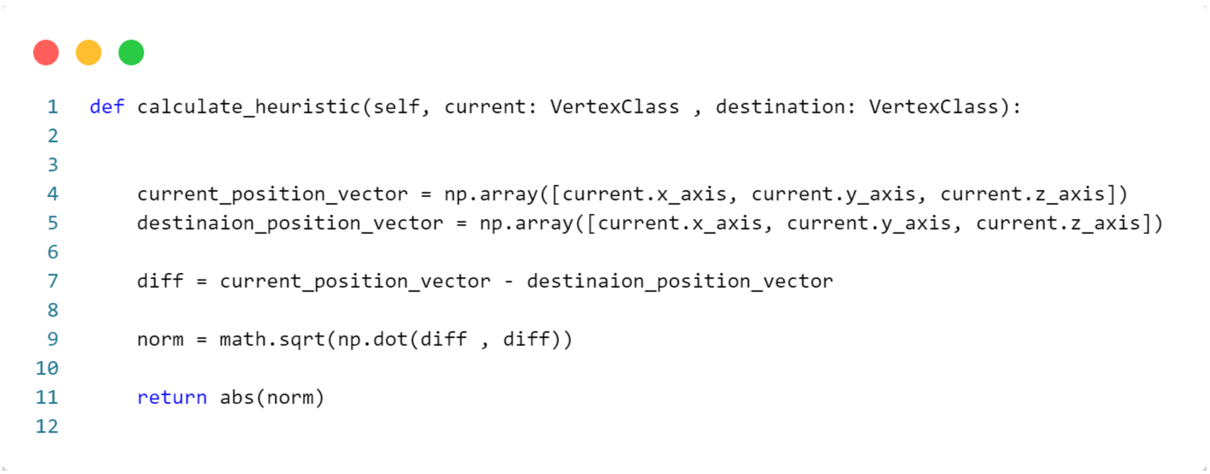


```python
def a_star(self, origin: VertexClass, destination: VertexClass):

        total_iterations = len(self.vertices)
        progress_bar = tqdm(total=total_iterations, desc='Finding The Best Path with The A*\'s Algorithm'.upper(),
                            unit="Vertex")

        closed_set = set()

        # making our heap and adding the origin as the starting point of the dijkstra
        root_value = []
        heap = Heap()
        f = self.calculate_heuristic(origin , destination)  #f = 0 + h
        heap.inster(f, root_value, origin)

        while len(closed_set) != len(self.vertices):
            next_path = heap.remove_min()

            if next_path.flight_info.name == destination.name:
                next_path.path.append(next_path.flight_info)
                return next_path

            last_path = None

            if isinstance(next_path, Entry):  # we want to expand the last destination we had
                last_path = next_path.flight_info
                closed_set.add(next_path.flight_info.name)

            vertex = None
            for item in self.vertices:
                if item.name == last_path.name:
                    vertex = item
                    break

            # finding the next destinations that we can go using this last path that we went
            updated_path = list()
            for i in next_path.path:
                updated_path.append(i)
            updated_path.append(vertex)

            # next_path.path.append(next_path.flight_info)
            # updated_path = next_path.path

            for item in vertex.endings:
                flag = False
                if isinstance(item, VertexClass):
                    # already has been checked
                    for name in closed_set:
                        if item.name == name:
                            flag = True
                            break
                    if flag:
                        continue

                e = vertex.endings[item]
                f = e.cost + next_path.key + self.calculate_heuristic(item, destination)
                heap.inster(f, updated_path, item)
            progress_bar.update(1)

        return None
```

The function **a_star** works exactly like the **dijkstra** function, with the difference that, instead of considering only the cost (function f) for evaluating the suitability of the selected vertex and selecting the vertex with the minimum cost, it considers the value **cost + h** at the end. Here, **h** is the heuristic function that will be explained later.

In general, the heuristic function provides an estimate of the future path from the current vertex to the destination vertex, allowing for better decision-making. The fundamental difference between this algorithm and the previous one (Dijkstra's) is that it can, simultaneously considering both past and future states (unlike Dijkstra's, which only considers the past), avoid selecting many vertices by taking into variables cost and heuristic variables. This is because a good estimate of these vertices to the destination has not been provided, thus finding the optimal path more quickly.

It's worth noting that in short paths, the A* algorithm may perform worse in terms of time compared to Dijkstra's because it has to calculate the heuristic function each time. However, in long paths, by avoiding the selection of many vertices that do not provide a good estimate of the future path, A* can significantly outperform Dijkstra's in terms of time.



```python
def calculate_heuristic(self, current: VertexClass , destination: VertexClass):


    current_position_vector = np.array([current.x_axis, current.y_axis, current.z_axis])
    destinaion_position_vector = np.array([current.x_axis, current.y_axis, current.z_axis])

    diff = current_position_vector - destinaion_position_vector

    norm = math.sqrt(np.dot(diff , diff))

    return abs(norm)

```

To calculate the heuristic function in the A* algorithm, it is necessary to leverage specific information about the problem. In the above heuristic function, by considering the three-dimensional geographical coordinates of each airport, we calculate the Euclidean distance between the source and destination airports.

Certainly, this distance is not equal to the sum of the distances traveled between different airports in the optimal path of this graph. However, it provides a good approximation of the selected vertex's situation, which is noteworthy.

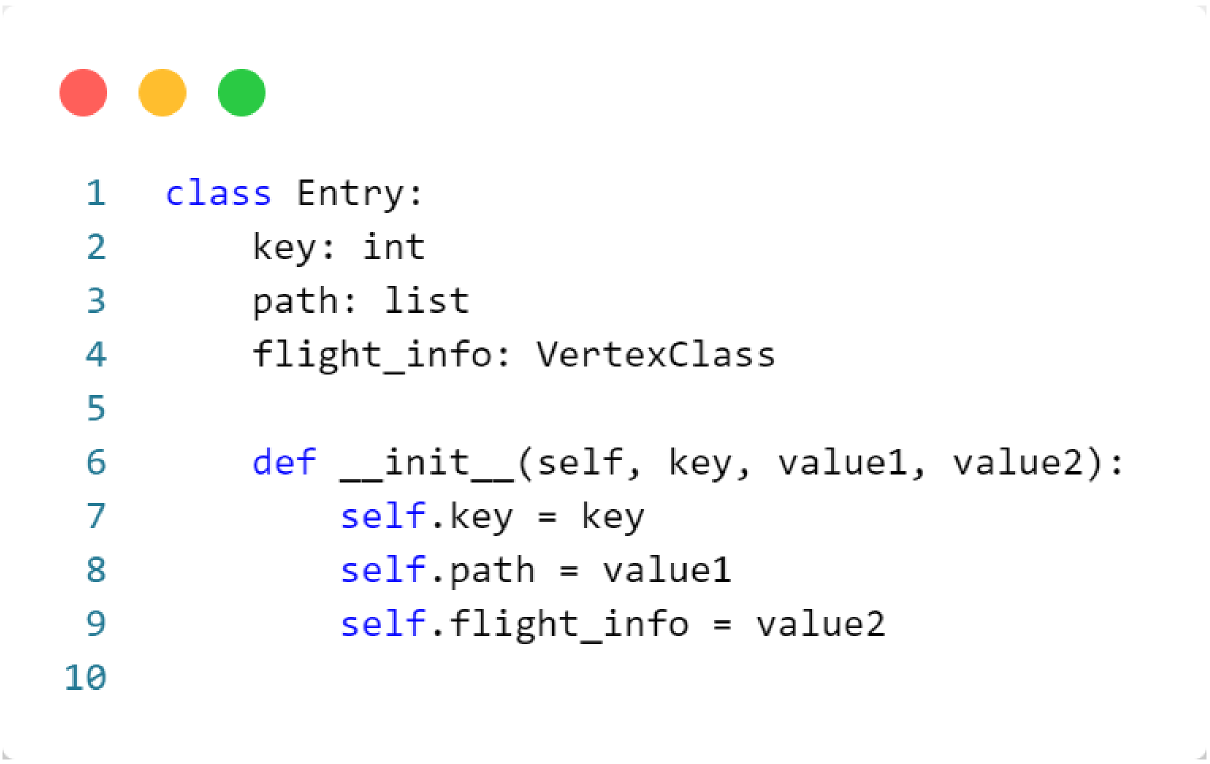
To calculate the Euclidean distance, we use the Euclidean norm of the difference between two vectors representing the geographical coordinates of these two airports, using the NumPy library for faster computations.

We could also use the L1 norm (Manhattan distance) for the heuristic function, which might give us a better approximation given the real distances in the dataset. However, our measurements showed that this change does not significantly affect finding the optimal path.

The calculation of the Euclidean norm of two vectors is as follows:

$$d(\mathrm{x}\,,\mathrm{y}) = ||x - y|| \; = \sqrt{\sum_{i=1}^{n} (x_i - \, y_i)^2}$$

Graph Module:



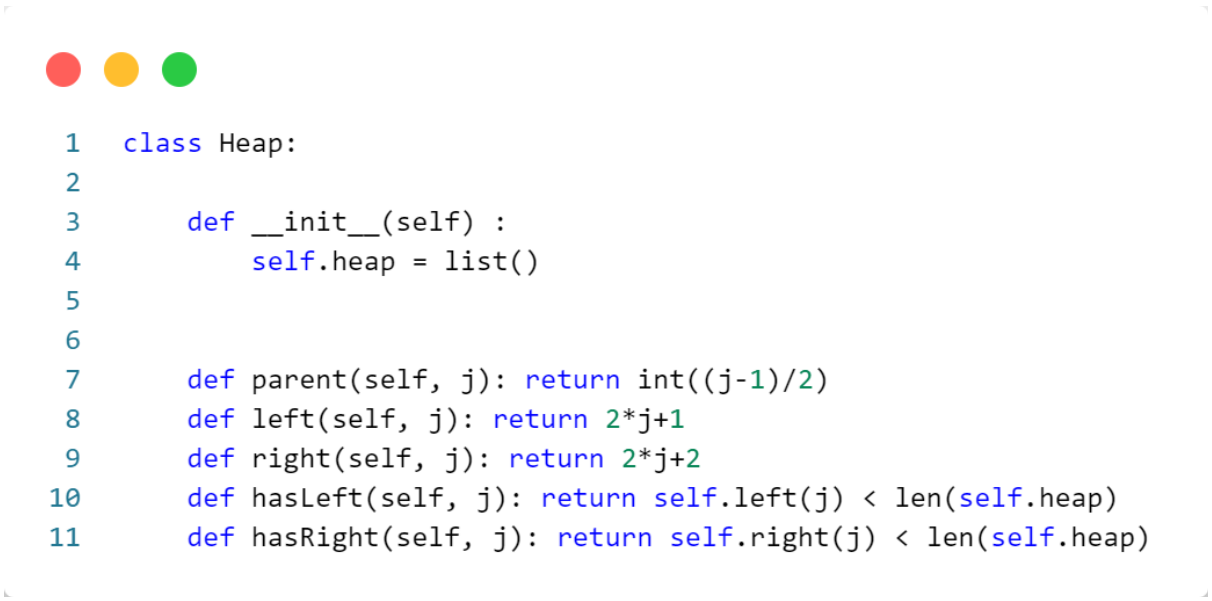
```python
class Entry:
    key: int
    path: list
    flight_info: VertexClass

    def __init__(self, key, value1, value2):
        self.key = key
        self.path = value1
        self.flight_info = value2

```

The **entry** class is created to implement a min-heap tree.

Each object of the **entry** class serves as a node in the tree and contains the following variables:

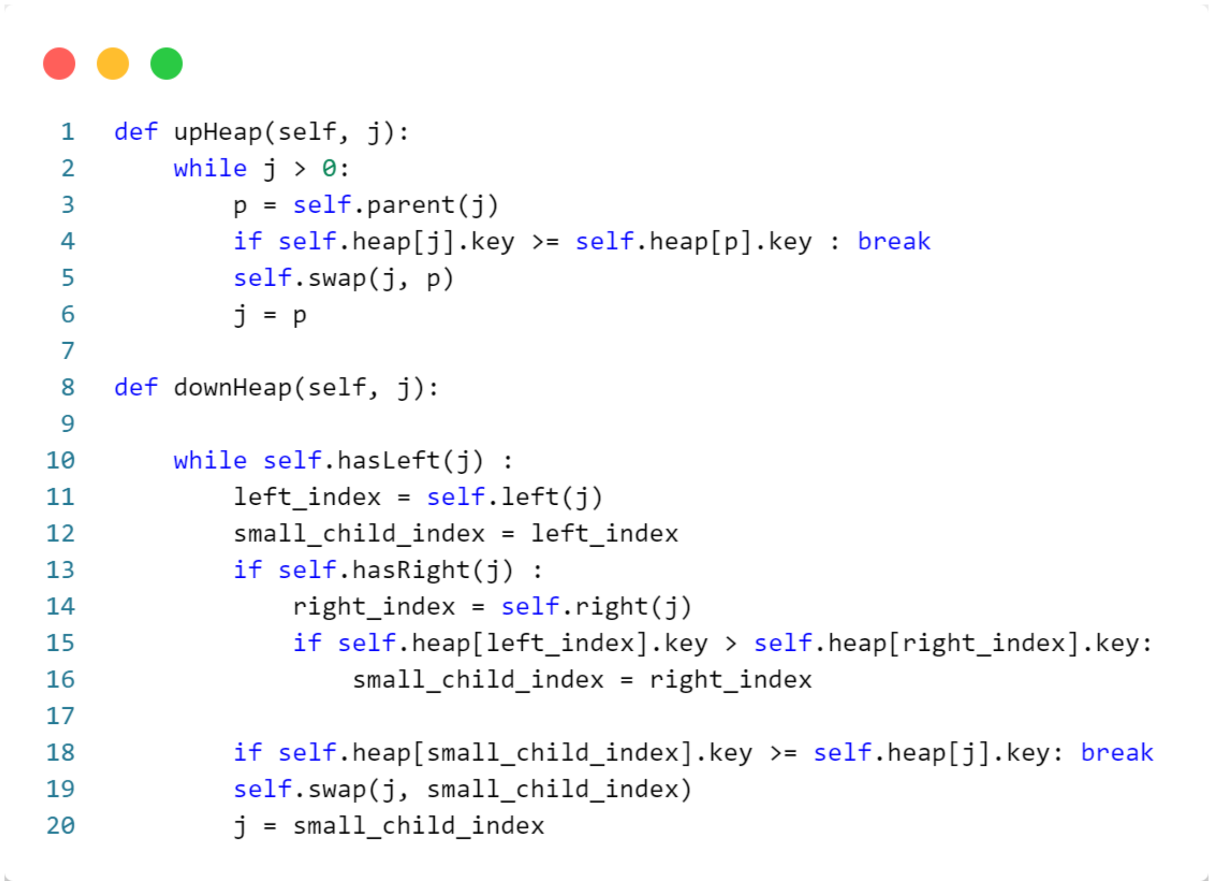
- **key**: Represents the cost of each flight for selecting the vertex with the minimum cost.
- **path**: The entire path traversed to the current vertex.
- **flight_info**: Information about the current vertex.

Continuing with the essential parts of the Heap class:



```python
class Heap:

    def __init__(self) :
        self.heap = list()


    def parent(self, j): return int((j-1)/2)
    def left(self, j): return 2*j+1
    def right(self, j): return 2*j+2
    def hasLeft(self, j): return self.left(j) < len(self.heap)
    def hasRight(self, j): return self.right(j) < len(self.heap)
```

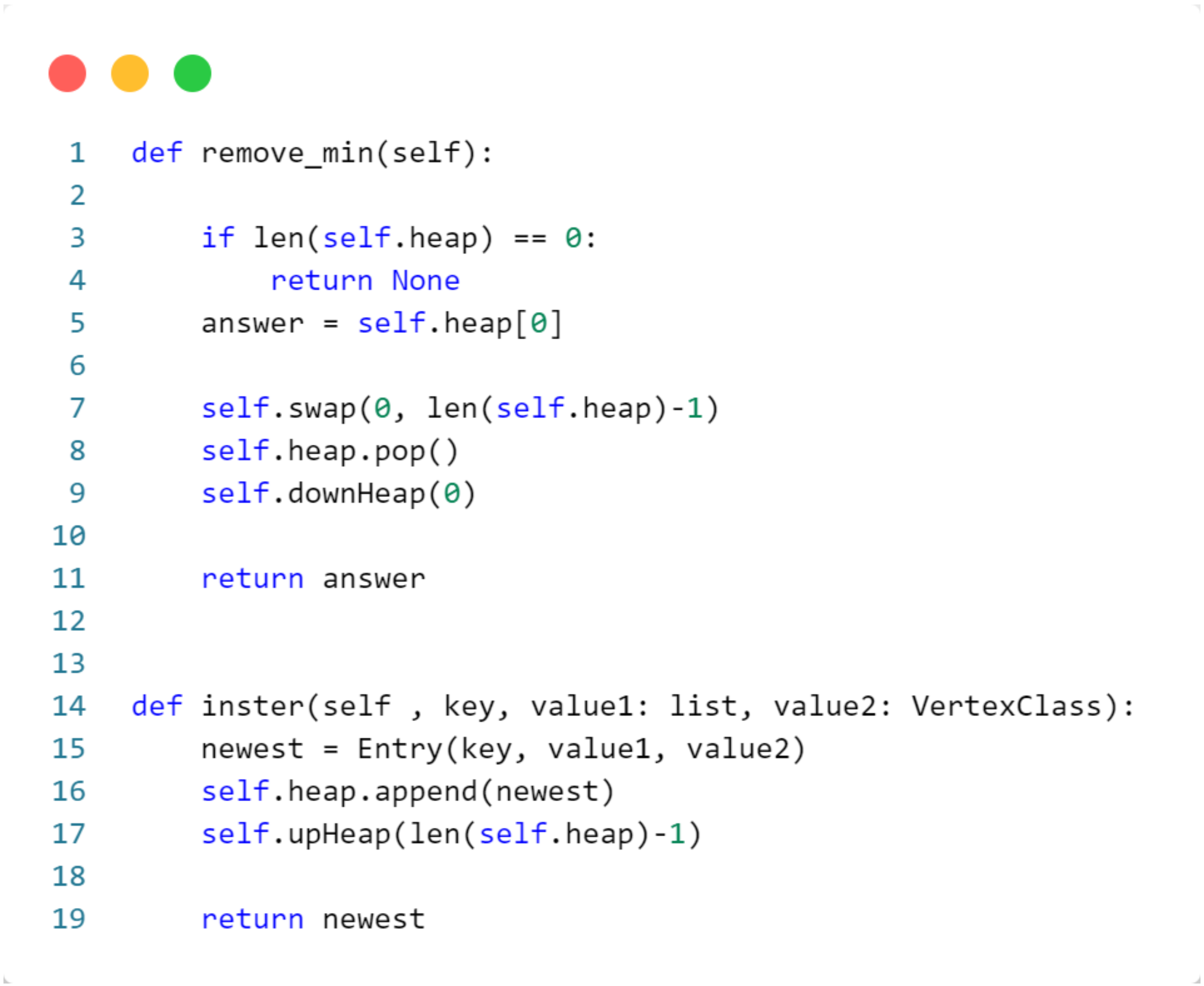
The **Heap** class has a heap list where tree nodes are stored. The functions parent, left, and right, given an index, return the corresponding parent, left child, and right child indices from the list. In heap tree implementations, parent is typically stored at index 2/(j-1), left at (2j+1), and right at (j2+2).



```python
def upHeap(self, j):
    while j > 0:
        p = self.parent(j)
        if self.heap[j].key >= self.heap[p].key : break
        self.swap(j, p)
        j = p

def downHeap(self, j):

    while self.hasLeft(j) :
        left_index = self.left(j)
        small_child_index = left_index
        if self.hasRight(j) :
            right_index = self.right(j)
            if self.heap[left_index].key > self.heap[right_index].key:
                small_child_index = right_index

        if self.heap[small_child_index].key >= self.heap[j].key: break
        self.swap(j, small_child_index)
        j = small_child_index
```

The **upHeap** function is utilized to restore the **heap property** after inserting an element. It compares the element at index **j** with its parent and, if necessary, swaps them until the heap property is satisfied.

The **downHeap** function is employed to restore the **heap property** after removing the minimum element. It compares the element at index **j** with its children and swaps with the smaller one.



```python
def remove_min(self):

    if len(self.heap) == 0:
        return None
    answer = self.heap[0]

    self.swap(0, len(self.heap)-1)
    self.heap.pop()
    self.downHeap(0)

    return answer


def inster(self , key, value1: list, value2: VertexClass):
    newest = Entry(key, value1, value2)
    self.heap.append(newest)
    self.upHeap(len(self.heap)-1)

    return newest
```

The **remove_min** function removes and returns the minimum element from the heap. It swaps the first element (the minimum) with the last one, removes the last element, and then performs a down-heap operation to maintain the **heap property**.

The **insert** function adds a new element with the specified key, value1, and value2 to the heap. It creates an entry object, appends it to the end of the heap list, and then performs an up-heap operation to maintain the **heap property**.

Usage

Script Usage:

1. Clone the Repository:

```
git clone https://github.com/UIAI-4021/search-pandas.git
cd scripts
```

2. Install Dependencies:

```
pip install -r requirements.txt
```

3. Run the Script:

```
python run_best_flight.py
```

Parameters:

- param1: "SourceAirLine - DestinationAitLine"

Example Usage:

```
python run_best_flight.py "Imam Khomeini International
Airport - Raleigh Durham International Airport"
```

References

- OpenAI. "ChatGPT." https://www.openai.com/
- GeeksForGeeks. https://www.geeksforgeeks.org/
- Stack Overflow. https://stackoverflow.com/
- W3Schools. https://www.w3schools.com/
- Russell, Stuart, and Norvig, Peter. "Artificial Intelligence: A Modern Approach." (Book)