# Code Explanation

Sample Input:

```
7
8
0 1 4
0 2 3
1 5 5
2 3 7
2 4 10
3 4 2
4 6 5
5 6 16
0 0 0
1 2 1
2 3 1
3 7 2
4 10 2
5 4 0
6 12 5
0 6
```

- In this the 1st input 7 is the total number of cities use will enter which we had stored in the variable totalCities.
- Then 2nd input 8 is the total number of connections we have in the graph so it will control the total number of lines of the 3rd input.
- Then 3rd input consist of multiple lines each consisting of two cities name and the distance between them. This distance between two cities we had stored in the graph[][] which we will pass to the Dijkstra and A star algorithm so that they can get the cost to travel between two cities, for calculating the shortest path.
- Then 4th input is the city name along with its coordinates which we are storing in the HashMap whose key is the name of the city and the value is the coordinates stored in the form of a list. This HashMap is passed to the A star algorithm for calculating the heuristic distance between the node and the target city using their respective coordinates.
- Then the 5th and the last input contains the source and the target city for which we had to find the shortest path and the shortest distance.

Then in the main we had created objects of the Dijkstra and the AStar class. In the constructor of the Dijkstra, we are passing source city, target city, total number of cities and the graph we had taken as 3rd input. Similarly for the constructor of the AStar algorithm we had passes same arguments as the Dijkstra with one additional HashMap for calculating the heuristic distance between each node and the target city. After the object creation we had called finalResult function of both the classes which print the shortest path between the source and the target city along with the shortest distance after calculating using both the algorithms.

Now the finalResult of both the classes will call their compute functions which will return the classDiskstra type object which will hols the target city along with the shortest Distance if a path exists between the source and the target city otherwise it will return null which will print that all nodes were visited but the target was not found otherwise will call the shortestDistAndPath function to print the shortest distance and the path between the source and the target city.

Now compute function will run till the priority queue is not empty. So basically using priority queue we are creating a table of all nodes with previous (parent nose) and total distance. For the start node we had set its total distance as zero in the constructor. And for all the other nodes instead of adding them into the table and setting their distance as infinity we had used priority queue in which we add only those nodes for which we had found the path. So using this we don't need to add those nodes which are unreachable from the start node. So queue contain all those nodes which we had reached and it rearranges the nodes in the ascending order based on their total distance. So we poll the first element from the queue which is till now closest to the source. And as for this node we had already calculated the shortest path so store it in a HashSet<Integer> which stores the nodes for which the shortest path is already found. We had also saved it as the present node and finds out its neighbours using the graph.

Now if the neighbour city is present in the shorthestPathFound HashSet it means its shortest distance is already calculated so no need to look at it and so look for another neighbour. But if it is not present then calculate its total distance by adding the present node distance and the distance between the present node and the neighbour. Then we check if the neighbour node is present in the HashMap cityObject, which contains all the cities which we had visited once. If it does not contains the neighbour city then we create an object of neighbour city by giving reference to the CityDijkstra class. And then add it in the hash map and the queue. When it is added in the queue it will automatically get rearrange in it depending on its total distance from the start.

And if neighbour already existed in the HashMap then compare the total distance which we had calculated above with the already stored distance in the object of that neighbour. So if this total distance is less then set the distance of the node as the total distance. And then for rearranging this neighbour node in the queue as well we need to remove and then add this node again, because priority queue not automatically rearranges its objects.

The same is the functioning of the AStar algorithm as well, the only difference is that while calculating the total distance for a node along with the distance of the present node with the start and the distance of priority node with the neighbour we also add the heuristic distance of the neighbour node with the target node and then stores this total distance as the present total distance of neighbour from the source if it is less

than already stored distance in the neighbour which is stored in it if we had visited it before as well.

For Above test case, we obtained the following result

**DIJKSTRA ALGORITHM RESULT**

```
city polled from queue 0
neighbor: 1
totalCost: 4.0
object for neighbor doesn't exist, so making a new object
neighbor: 2
totalCost: 3.0
object for neighbor doesn't exist, so making a new object
city polled from queue 2
neighbor: 3
totalCost: 10.0
object for neighbor doesn't exist, so making a new object
neighbor: 4
totalCost: 13.0
object for neighbor doesn't exist, so making a new object
city polled from queue 1
neighbor: 5
totalCost: 9.0
object for neighbor doesn't exist, so making a new object
city polled from queue 5
neighbor: 6
totalCost: 25.0
object for neighbor doesn't exist, so making a new object
city polled from queue 3
neighbor: 4
totalCost: 12.0
object for neighbor exist!
city polled from queue 4
neighbor: 6
totalCost: 17.0
object for neighbor exist!
city polled from queue 6
```

**the shortest Distance calculated using Dijkstra Algorithm between city 0 and city 6 is :17.0**
**the shortest path computed using Dijkstra Algorithm between city 0 and city 6 is :[0, 2, 3, 4, 6]**

**ASTAR ALGORITHM RESULT**

```
city polled from queue 0
neighbor: 1
heuristic: 10.770329614269007
totalCost: 14.770329614269007
object for neighbor doesn't exist, so making a new object
neighbor: 2
heuristic: 9.848857801796104
```

totalCost: 12.848857801796104
object for neighbor doesn't exist, so making a new object
city polled from queue 2
neighbor: 3
heuristic: 5.830951894845301
totalCost: 15.8309518948453
object for neighbor doesn't exist, so making a new object
neighbor: 4
heuristic: 3.605551275463989
totalCost: 16.605551275463988
object for neighbor doesn't exist, so making a new object
city polled from queue 1
neighbor: 5
heuristic: 9.433981132056603
totalCost: 18.4339811320566
object for neighbor doesn't exist, so making a new object
city polled from queue 3
neighbor: 4
heuristic: 3.605551275463989
totalCost: 15.60555127546399
object for neighbor exist!
city polled from queue 4
neighbor: 6
heuristic: 0.0
totalCost: 17.0
object for neighbor doesn't exist, so making a new object
city polled from queue 6
**the shortest Distance calculated using AStar Algorithm between city 0 and city 6 is :17.0**
**the shortest path computed using AStar Algorithm between city 0 and city 6 is :[0, 2, 3, 4, 6]**