Dijkstra Algorithm Application: Cheapest Airline Fare Between Departure and Arrival Cities

Muhammad Naufal Dzakki Rauf, Don Rui Tornado Rosa, Rizky Ramadhan, Muhammad Syauqi Abdurrahman
Department of Computer Science and Electronics, Faculty of Mathematics and Natural Sciences, Universitas Gadjah Mada
Sekip Utara Bulaksumur, Kotak Pos 21, Senolowo, Sinduadi, Kec. Mlati, Kabupaten Sleman, Daerah Istimewa Yogyakarta, Indonesia
mnaufaldr01@mail.ugm.ac.id
don.rui.t.r@mail.ugm.ac.id
rizkyramadhan@mail.ugm.ac.id
syauqiabdurrahman@mail.ugm.ac.id

***Abstract*- Finding the shortest path from one point to the destination within a graph is solved by using Dijkstra's Algorithm. In our daily lives we can find many problems that can be classified as a shortest path problem, for example deciding which plane route between departure and arrival cities we should take that will cost the least airline fare. This paper is focused on the basics of Dijkstra's Algorithm and how this problem can be solved by implementing Dijkstra's Algorithm.**

***Keywords*- Airline Fare, Plane Route, Dijkstra's Algorithm, Graph, Shortest Path Problem**

I. Introduction & Problem

Executing a task with as little resources as possible is one of the most important aspects of achieving efficiency within doing one's work. As it allows the achievement of more while using less, this problem is often encountered in businesses, most businesses try to have the maximum profit while still minimizing the cost. This problem can be classified as one of the classic problems in graph theory, that is the shortest path problem.

The shortest path problem, as the name implies, is the problem of determining the path between two specified vertices, where one acts as the source and the other as the destination, within a weighted graph with the least sum of weight of edges taken. Shortest Path problems can be found within our daily lives, as usually the weight of the edges are used to represent time, price, and distance. Problems such as choosing which route to take to work in order to not be late, determining which route that cost the least, and as the name implies, the shortest distance are all classified as shortest path problems. This paper will focus on a problem of deciding which plane(s) from one city to another will have the least cost of the whole trip.

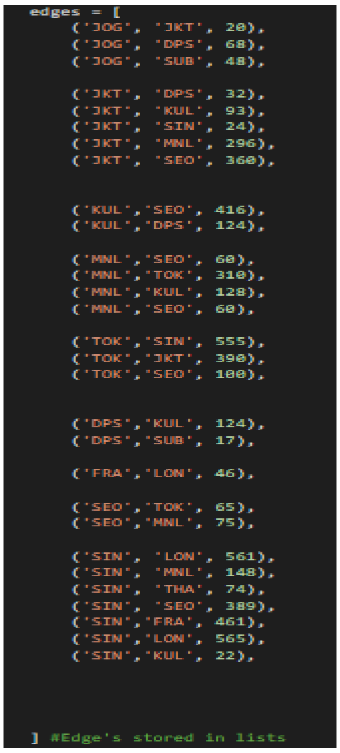
On holidays, people often travel to have a unique experience that is different from what they go through in their daily lives to get a new and refreshing experience, as holidays allow people to "switch off" from their normal habits and lets them do whatever they want with the time they have, may it be going off to see new places, pick up new skills, or just simply enjoying the freedom so that they can come home with pleasant memories (Krippendorf, 1987; Carmouche & Kelly, 1995).

Airplane is one of the most used modes of transportation for travelling, as it allows overseas trips that enables us to experience more new places, although it also comes with the appropriate cost of such service as well. Nowadays, purchasing airline tickets has become much easier with the presence of websites that allow the users to book their homes from the comfort of their own home or wherever they may be. However, a new problem arises as often those websites don't always recommend the cheapest possible flights from departure to arrival cities. That's why this paper will discuss an algorithm that may help the travellers to get the best possible price for their trips.

The algorithm that is used to solve this problem is the Dijkstra's Algorithm. Dijkstra algorithm is an algorithm used to find the shortest path between two vertices in a graph, first conceived in 1956 by computer scientist Edsger Dijkstra. The Dijkstra algorithm is an iterative greedy algorithm, which makes the choice based on the optimum solution at each iteration that looks best at the moment. The main idea of Dijkstra's algorithm is to maintain a table of all current shortest paths from a source vertex to all vertices, and after finishing, the table will have the shortest path from the source vertex to all other vertices. Within every iteration it will find a vertex whose edge between them and the source vertex possesses the least weight. Using this algorithm solves the shortest path problem with the time complexity of $O(|E| + |V| \log |V|)$ (Barbehenn, 1998), but with the condition that there is no negative weight cycle present within the graph, as if there is even a single negative weight cycle present in the graph then it will not be able to detect the shortest path.

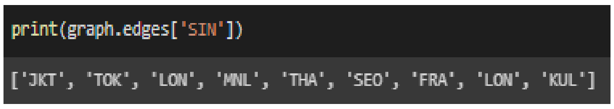
II. Methods

There are many ways to implement Djikstra's algorithm. The application of Dijkstra potentially involves numerous data that could represent either nodes, edges or vertices. Despite the potential of numerous data, the implementation of simple data structures could provide further simple simulation to traverse the nodes in the graph. List or array and dictionary are examples of the data structures that seem suited for the task. A list or array is a simple data structure suited for storing data in a single variable to store edges present in the graph. While dictionary is a unique data structure that could be accessed by a key and is suited to store neighboring vertices that a node has and also weights of each edge.



```
edges = [
    ('JOG',  'JKT', 20),
    ('JOG',  'DPS', 68),
    ('JOG',  'SUB', 48),

    ('JKT',  'DPS', 32),
    ('JKT',  'KUL', 93),
    ('JKT',  'SIN', 24),
    ('JKT',  'MNL', 296),
    ('JKT',  'SEO', 360),


    ('KUL','SEO', 416),
    ('KUL','DPS', 124),

    ('MNL','SEO', 60),
    ('MNL','TOK', 310),
    ('MNL','KUL', 128),
    ('MNL','SEO', 60),

    ('TOK','SIN', 555),
    ('TOK','JKT', 390),
    ('TOK','SEO', 100),


    ('DPS','KUL', 124),
    ('DPS','SUB', 17),

    ('FRA','LON', 46),

    ('SEO','TOK', 65),
    ('SEO','MNL', 75),

    ('SIN',  'LON', 561),
    ('SIN',  'MNL', 148),
    ('SIN',  'THA', 74),
    ('SIN',  'SEO', 389),
    ('SIN','FRA', 461),
    ('SIN','LON', 565),
    ('SIN','KUL', 22),



] #Edge's stored in lists
```

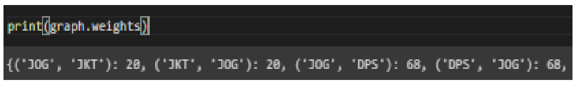
Fig 1 Lists of Edges



```
print(graph.edges['SIN'])
```

```
['JKT', 'TOK', 'LON', 'MNL', 'THA', 'SEO', 'FRA', 'LON', 'KUL']
```

Fig 1.1 Dictionary with The Key 'SIN' or abbreviation of Singapore Contains Other Adjacent Nodes or Cities

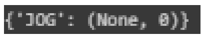


```
print(graph.weights)
```

```
{('JOG', 'JKT'): 20, ('JKT', 'JOG'): 20, ('JOG', 'DPS'): 68, ('DPS', 'JOG'): 68,
```

Fig 1.2 Dictionary Containing Weight of Edges Present in the Graph

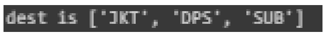
These datas will then be processed by a function that simulates Dijkstra's Algorithm. To implement it, additional variables are required to store additional information such as shortest paths (dictionary), current node, visited nodes, current weight, final path, etc. The current node is also initialized as an 'initial' variable to be able to utilize it as a placeholder to proceed to the next set of nodes and inserted to the shortest path variable.



```
{'JOG': (None, 0)}
```

Fig 1.3 Initial Variable stored in 'shortest_path' variable

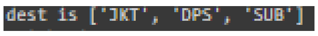
'Visited' variable is used to store the nodes the algorithm selected to be the next initial node. 'Destinations' variable is used to store the neighboring nodes of each current node.



```
dest is ['JKT', 'DPS', 'SUB']
```

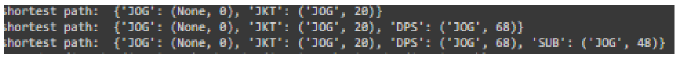
Fig 1.4 'destination' variable storing the current node's neighbors

The algorithm will mostly run on a while loop, where it will keep running until the current node reaches the destination node. Firstly it will insert the initial node to the 'visited' data container. Then it will try to detect the initial node's neighbors as 'destinations' variable. Then it will try to find the shortest path by inserting the neighbors into the 'destinations' variable then checking the weights of each edge to find which neighbor would produce the shortest path. It will then select the next node to visit by assigning the variable 'currNode' to select the node present in the 'shortestPaths' that has the minimum weight. This process will continue until the current node is the destination wanted.



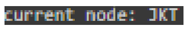
```
dest is ['JKT', 'DPS', 'SUB']
```

Fig 1.5 Initial node's neighbors



```
hortest path:  {'JOG': (None, 0), 'JKT': ('JOG', 20)}
hortest path:  {'JOG': (None, 0), 'JKT': ('JOG', 20), 'DPS': ('JOG', 68)}
hortest path:  {'JOG': (None, 0), 'JKT': ('JOG', 20), 'DPS': ('JOG', 68), 'SUB': ('JOG', 48)}
```

Fig 1.6 Finding which neighbor will result the shortest path



```
current node: JKT
```

Fig 1.7 Current node is assigned to a new node with shortest distance

The overall flow of the program can be visually represented by the following flowchart of Fig 1.8.

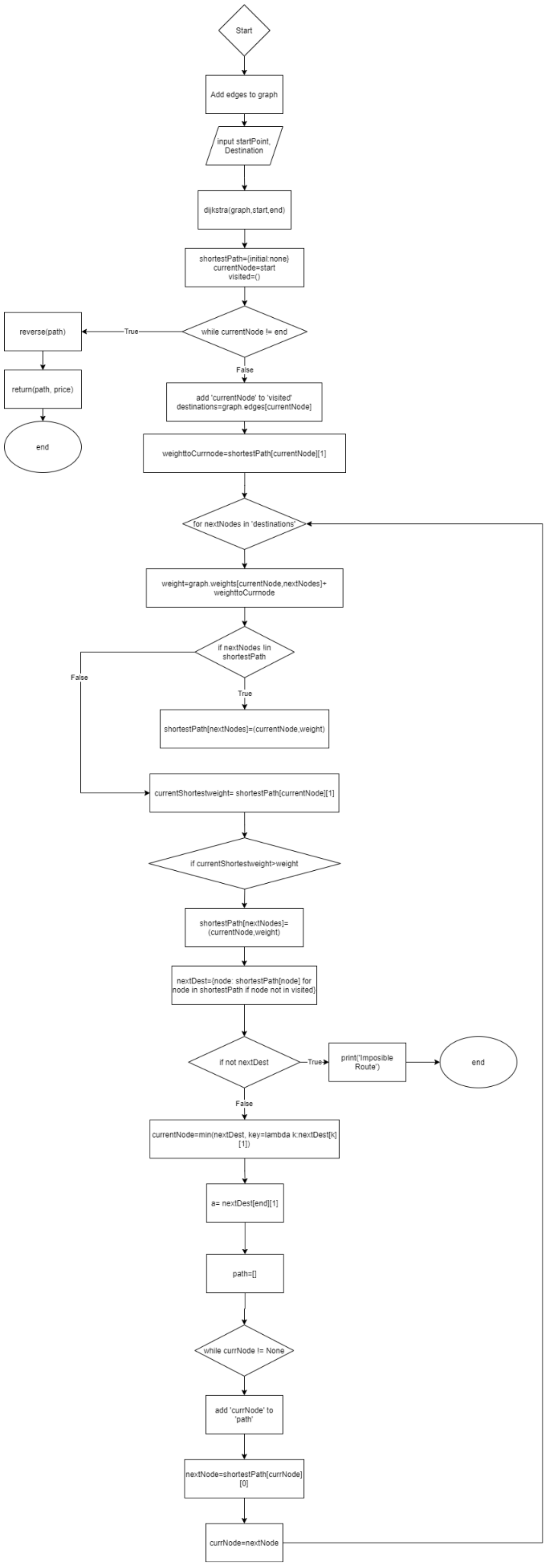
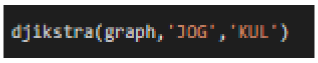

Start
Add edges to graph
input startPoint, Destination
dijkstra(graph,start,end)
shortestPath={initial:none}
currentNode=start
visited=()
while currentNode != end
True
reverse(path)
return(path, price)
end
False
add 'currentNode' to 'visited'
destinations=graph.edges[currentNode]
weighttoCurrnode=shortestPath[currentNode][1]
for nextNodes in 'destinations'
weight=graph.weights[currentNode,nextNodes]+ weighttoCurrnode
if nextNodes !in shortestPath
False
True
shortestPath[nextNodes]=(currentNode,weight)
currentShortestweight= shortestPath[currentNode][1]
if currentShortestweight>weight
shortestPath[nextNodes]= (currentNode,weight)
nextDest={node: shortestPath[node] for node in shortestPath if node not in visited}
if not nextDest
True
print('Imposible Route')
end
False
currentNode=min(nextDest, key=lambda k:nextDest[k][1])
a= nextDest[end][1]
path=[]
while currNode != None
add 'currNode' to 'path'
nextNode=shortestPath[currNode][0]
currNode=nextNode

Fig 1.8 Flowchart of the program

III. Result

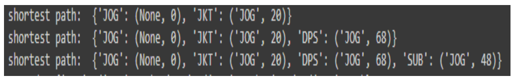
As an example, try to find the shortest flight fare from Yogyakarta to Kuala Lumpur. In this program, our limitation is the unavailability of sources that would allow us to collect data of flight prices. Therefore, the datas collected will be static data but from real occurring flights.



```
djikstra(graph,'JOG','KUL')
```

Fig 2.0 Calling the Dijkstra's Algorithm

'JOG' or short for Yogyakarta is stored at the 'visited' variable as in *Fig* 1.3. It will then try to find its neighbors and store them in the 'destinations' variable as in *Fig 1.4* with initial weight is 0. The current node ('JOG') will proceed to a for loop to find which of Yogyakarta's neighbors has the shortest weight between them.

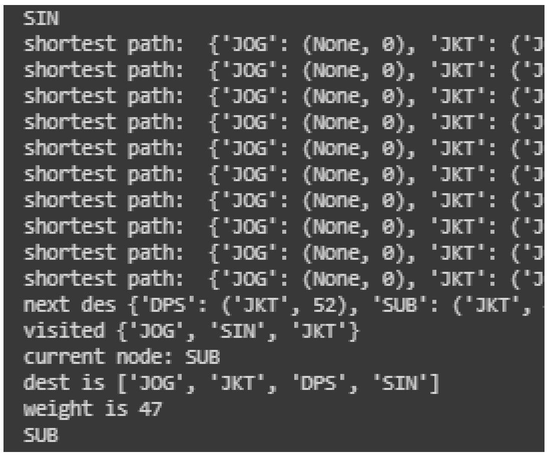


```
shortest path:  {'JOG': (None, 0), 'JKT': ('JOG', 20)}
shortest path:  {'JOG': (None, 0), 'JKT': ('JOG', 20), 'DPS': ('JOG', 68)}
shortest path:  {'JOG': (None, 0), 'JKT': ('JOG', 20), 'DPS': ('JOG', 68), 'SUB': ('JOG', 48)}
```

Fig 2.1 Yogyakarta's neighbors are inserted to 'shortestPath' variable

The 'shortestPath' variable is where the nodes of neighbors will be contained and compared. Each time a node has a new neighbor that was not in the 'shortestPath' variable early on, will be inserted into it. If the same node is present in the 'shortestPath' variable, the total price will be compared. If the previous one is still cheaper, it will not update the price. The price update could affect the 'nextNode' selection later on.

After inserting the 'currentNode' neighbors, the program will select the edge that has the minimum weight. In this case the next minimum weight is from Yogyakarta to Jakarta which costs $20. It will then do the same for Jakarta, storing and updating (if there are new shortest route) the 'shortestPaths' variable and selecting the next edge from Jakarta that has the minimum edge. From Jakarta, it will select to go to Singapore with a cost of $24 which made the total cost to be $44. After Singapore, it will head for Surabaya first due to its total price being calculated as the cheapest option costing a total of $47 despite being able to go directly to Kuala Lumpur.

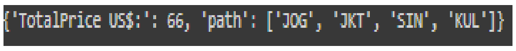


```
SIN
shortest path:  {'JOG': (None, 0), 'JKT': ('J
shortest path:  {'JOG': (None, 0), 'JKT': ('J
shortest path:  {'JOG': (None, 0), 'JKT': ('J
shortest path:  {'JOG': (None, 0), 'JKT': ('J
shortest path:  {'JOG': (None, 0), 'JKT': ('J
shortest path:  {'JOG': (None, 0), 'JKT': ('J
shortest path:  {'JOG': (None, 0), 'JKT': ('J
shortest path:  {'JOG': (None, 0), 'JKT': ('J
shortest path:  {'JOG': (None, 0), 'JKT': ('J
shortest path:  {'JOG': (None, 0), 'JKT': ('J
next des {'DPS': ('JKT', 52), 'SUB': ('JKT',
visited {'JOG', 'SIN', 'JKT'}
current node: SUB
dest is ['JOG', 'JKT', 'DPS', 'SIN']
weight is 47
SUB
```

Fig 2.2 From Singapore to Surabaya

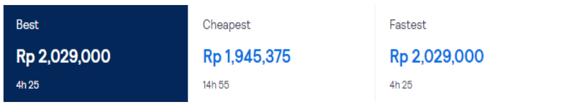
From Surabaya, it will then visit Denpasar due to the low total cost of $52 compared to all other destinations in the 'nextDes' variable. After Denpasar, there is an edge to Kuala Lumpur. But the algorithm decides that there were lower options made previously when the current node was Singapore. From Singapore, the total cost would be $66 to Kuala Lumpur compared to Denpasar to Kuala Lumpur alone could cost is $124. Therefore it would reach Kuala Lumpur, the destination with a total cost of $66.

The total cost is $66 from Yogyakarta to Kuala Lumpur with the path Yogyakarta, Jakarta, Singapore, Kuala Lumpur. Compared to results from skyscanner.co.id, we could achieve a cheaper flight in total.



```
{'TotalPrice US$:': 66, 'path': ['JOG', 'JKT', 'SIN', 'KUL']}
```

Fig 2.3 Result of the Dijkstra's Algorithm



Best	Cheapest	Fastest
Rp 2,029,000	Rp 1,945,375	Rp 2,029,000
4h 25	14h 55	4h 25

Fig 2.4 Flight from Yogyakarta to Kuala Lumpur

IV. Conclusion

The implementation of Djikstra is definitely helpful for travelers that might want to take travel at a relatively low cost. Sadly there are a couple of things to consider before using this method. Firstly, the time for each flight. The time gap in between could vary depending on the available flights in a city. Regional flights occur almost daily, therefore at most a waiting time of up to a day is possible using this method. Secondly is the price changes. The difficulty to find sources of flights is quite high to use this method and usually only provided by airlines to a reputable ticket seller. If implementers could find a source to get flight details from the source (airlines), it could definitely help to find better flights using this method. But in all, the results show that getting a cheaper flight is possible.

References

Javaid, Adeel. (2013). Understanding Dijkstra Algorithm. SSRN Electronic Journal. 10.2139/ssrn.2340905

Deepa, G., Kumar, P., Manimaran, A., Rajakumar, K., & Krishnamoorthy, V. (2018). Dijkstra Algorithm Application: Shortest Distance between Buildings. *International Journal of Engineering & Technology, 7*(4.10), 974-976. doi:http://dx.doi.org/10.14419/ijet.v7i4.10.26638

A Fitriansyah et al 2019 J. Phys.: Conf. Ser. 1338 012044

Steyn, Sonja & Saayman, Melville & Nienaber, Alida. (2004). The impact of tourist and travel activities on facets of psychological well-being : research article. South African Journal for Research in Sport, Physical Education and Recreation. 26. 10.4314/sajrs.v26i1.25880.

Krippendorf, J. (1987). The holiday makers. London: Heinemann.

Carmouche, R. & Kelly, N. (1995). Behavioural studies in hospitality management (1st ed.). London: Chapman & Hall.

Michael Barbehenn. 1998. A Note on the Complexity of Dijkstra's Algorithm for Graphs with Weighted Vertices. IEEE Trans. Comput. 47, 2 (February 1998), 263. DOI:https://doi.org/10.1109/12.663776

Ur Rehman, Atta & Awuah-Offei, Kwame & Baker, D. & Bristow, Douglas. (2019). EMERGENCY EVACUATION GUIDANCE SYSTEM FOR UNDERGROUND MINERS.