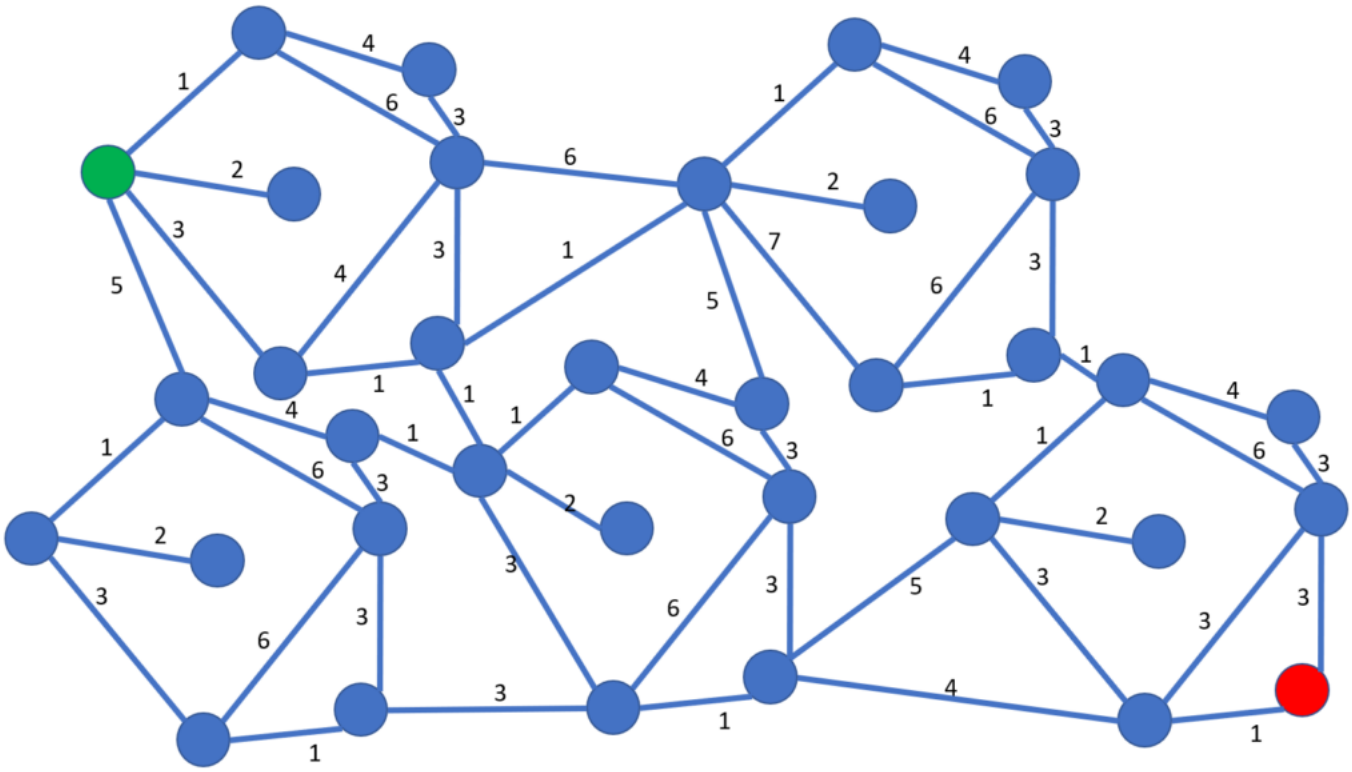
Algorithms: Design & Analysis

Project: Final Report

04 December 2020



(Giraud)

Contents

1 Acknowledgement

The following document has been typed using Overleaf; online LaTeX editor.

The rest of this page is intentionally left blank.

2 Introduction & Logistics

2.1 Topic

Shortest path problem

2.2 Group Members

All names appear in alphabetical order; any other apparent pattern is purely coincidental.

- Maaz Saeed - ms05050
- Syed Ammar Mahdi - sm03691
- Syeda Zainab Fatima - sf05166

2.3 Submitted to

Dr. Shah Jamal Alam - Algorithms: Design & Analysis L3

2.4 Research and Findings

For any intents and purposes, major components such as the project proposal, algorithms studied, and a copy of the presentation can be viewed at our public GitHub repository.
https://github.com/healyyyyyy/CS412-Final-Project

2.5 A brief inquiry

It proves difficult to determine when the *shortest path problem* first arose in history; one could, perhaps, say it is safe to assume that even primitive societies would have come across, and contemplated over the idea.

As the name suggests, the problem is merely the comparison between two or more paths from location to another (or in graph theory, from one node to another). As simple as it may seem, it has caused an abundance of issues in the determining of optimal paths one can take. In this report, we will study, more specifically, it's existence in Graph Theory, and a few computer algorithms that aim to solve said problem.

3 Solutions to the problem

3.1 Dijkstra's Algorithm

Edsger W. Dijkstra, conceived his shortest path algorithm in 1956 (though it was only published after 1959) (Richards).

Dijkstra's algorithm (as it has come to be commonly known), is an iterative algorithm that finds the shortest path from a given node, to all other nodes in a weighted graph.

3.1.1 How it works

The logic behind the algorithm can be divided into five simple steps as listed below:

1. All nodes are marked as un-visited.
2. Pick an initial node, mark it's distance as 0; infinite for the rest.
3. Select a starting node assign it a weight (distance) of 0 from and to itself, mark it as your current node.
4. Repeat step 3, for current node's neighbours. E.g: If you move from A to B with A having a distance of 2 and B having a distance of 1, B is now your current node with distance 2+1 = 3. If 3 is lesser than the distance previously marked for B, change the distance to 3. This step explains the default distance of infinite for all nodes at the start, as any distance we calculate has to be $\leq \infty$.
5. The current node is marked as visited, and removed from the set of un-visited nodes.
6. The loop ends when all nodes have been visited or if the shortest distance between un-visited nodes remains infinite.

3.1.2 Limitations

Although Dijkstra's algorithm has an extremely fast run-time, it does not come without shortcomings.

If one were to summarize the conditions under which Dijkstra's algorithm fails to deliver on it's promise of determining the shortest path, in a single word, that would be *greed*.

The algorithm relies heavily on one simple constraint. It assumes all edges are positively weighed. This leads to the commonly known fact: *Dijkstra's algorithm can not operate with negatively weighted edges.* The algorithm is termed as "greedy", as aforementioned, due to the fact that it looks for a local optimum, rather than a global one.

Case 1: If the algorithm is faced with two or more paths, it will determine the shortest path from only the first node it can reach, in each path. Now there's a possibility that, for example, node A has a weight ¡ node F. Dijkstra will blindly follow the path leading onward from node A. However, it is entirely possible that the path that exists beyond node F is comprised of negative weights which lead to a smaller total distance than the path the algorithm actually chose. This renders the algorithm void as it fails to serve it's very purpose of finding the shortest path.

Case 2: As another example, assume we desire to start our traversal from node A, and reach node G, without needing to visit each node. It is entirely possible that there exists a path from A to G, or even a single and direct edge between the two nodes, that the algorithm will choose to ignore because it sees a better local optimum, i.e a neighbour from it's current node that has a smaller distance than the path that leads to our global optimum.

In a nutshell, it assumes that any vertex it has visited does not need to be checked again and that any path originating from it will have a higher distance than what it has already calculated. This does not hold true when negative weights are involved.

3.1.3 An implementation in Python

The following code iterates once over each vertex in the graph. The order of iteration is determined by a priority queue. The value that determines the order, in the priority queue, is the distance from our starting vertex. Implementing a priority queue ensures that we visit each vertex, one after the other, and that we are always looking at the one with the smallest distance.

```
import heapq

def calculate_distances(graph, starting_vertex):
    distances = {vertex: float('infinity') for vertex in graph}
    distances[starting_vertex] = 0

    pq = [(0, starting_vertex)]
    while len(pq) > 0:
        current_distance, current_vertex = heapq.heappop(pq)

        if current_distance > distances[current_vertex]:
            continue

        for neighbor, weight in graph[current_vertex].items():
            distance = current_distance + weight
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(pq, (distance, neighbor))

    return distances

print(calculate_distances(graph, 'X'))
```

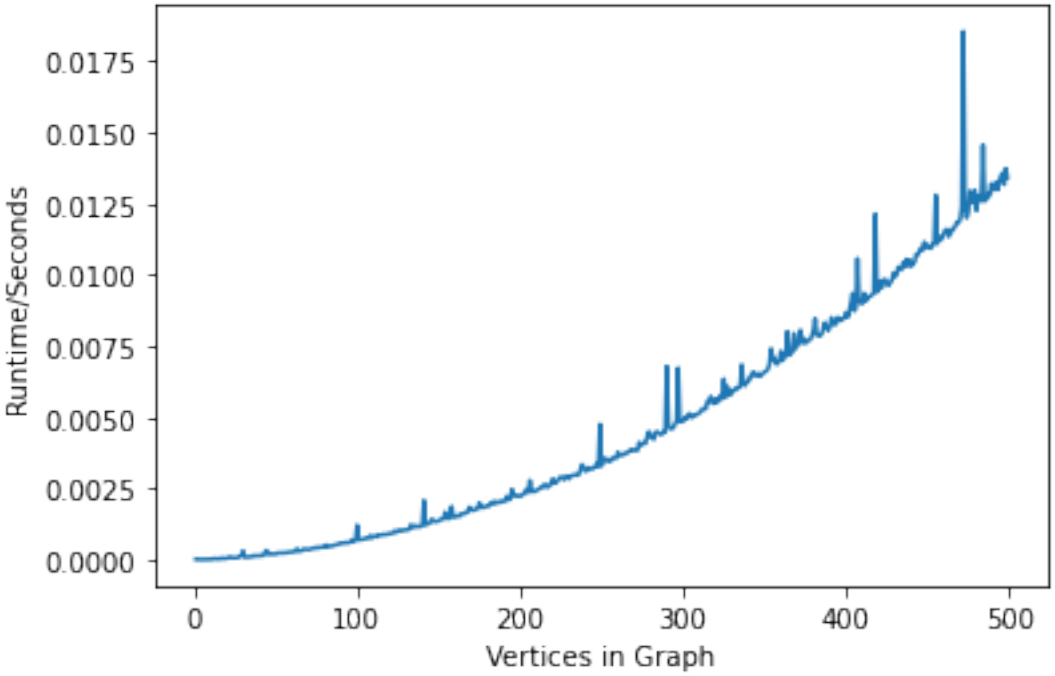
("Shortest Path With Dijkstra's Algorithm", Bradfield)

3.1.4 Theoretical & Empirical Analysis

The aforementioned algorithm boasts a theoretical running time of $O(V + ELogE)$, where V & E are the number of vertices and edges in the graph, respectively. The derivation is as follows:

- Building the dictionary of distances: O(V)
- While loop: O(E)
- For loop: O(E)
- Adding/Removing priority queues: O(Log E)
- Total theoretical running time: O(V + E Log E)

After running the code on our own machine, we obtained the following graph:



3.1.5 Practical Application of Dijkstra's Algorithm

- Application of Dijkstra's Algorithm in Fire Evacuation System

  Traditional methods of fire evacuation through the use of signs to guide people to evacuate a large building during an on-site fire or the use of evacuation lights installed prior to any incident are outdated. They not only fail at being noticed during a crisis but also may sometimes lead people to the source of the fire as they are not smart enough to detect the source on their own.

  For this reason, an intelligent fire evacuation system has been provided as an optimal solution to this problem. One of the most important features of this system is its ability to track and detect the source of the fire, allowing for the following steps to be taken in such a manner that the source of fire is completely avoided.

  Dijkstra's algorithm is used after the fire source is detected, to enable the system to find an optimal path which is also the shortest path. This situation is one of life and death, and obtaining the minimum distance to safety is very important for this system to work efficiently.

  (Xu et al.)

3.2 Bellman Ford Algorithm

The Bellman-Ford algorithm computes, from a single source vertex, the shortest path to all other vertices. It is named after Richard Bellman and Lester Ford Jr., who published it in 1958 and 1956, respectively (Schrijver, 2005). Another (lesser known) variant of the algorithm exists, under the name *Bellman-Ford-Moore*, published in 1959 with contributions from Edward F. Moore (Bang-Jensen and Gutin, 2009).

Interestingly enough, there exists evidence which proposes that it was actually Alfonso Shimbel, who was the first to discover the algorithm in 1955. Though his name was never officially associated with it, like the aforementioned individuals.

3.2.1 How it works

1. Initialize distances from source to all vertices as infinite. Create an array 'dist' that contains these distances.
2. Run a loop $|V| - 1$ times, where V is the number of vertices.
   - For each edge u-v, check if dist[v] > dist[u] + weight of edge u-v.
   - If true, update dist[v] as dist[u] + weight of edge
3. This step checks for negative cycles. If dist [v] > dist[u] + weight of edge u-v then the algorithm reports a negative cycle. This confirms whether the output is, indeed, the shortest distance or is the shortest distance achievable due to the fact that an infinite number of trips around the cycle can infinitely reduce distance.

3.2.2 Limitations

As much as Bellman-Ford is known to be more reliable than Dijkstra's, when it comes to negative weights, it is not entirely a full-proof option either. If one were to apply Bellman-Ford on an **un-directed** graph, with negative edges, the algorithm would fail to find the shortest distance as it would report a negative cycle on any edge with a negative weight. As far as it's concerned, any un-directed edge from u to v, is a negative cycle, going from u to v and back, infinitely (Sryheni, 2020).

3.2.3 An implementation in Python

```
def bellman_ford(graph, source):
    # Step 1: Prepare the distance and predecessor for each node
    distance, predecessor = dict(), dict()
    for node in graph:
        distance[node], predecessor[node] = float('inf'), None
    distance[source] = 0

    # Step 2: Relax the edges
    for _ in range(len(graph) - 1):
        for node in graph:
            for neighbour in graph[node]:
                # If the distance between the node and the neighbour is lower than the current, store i
                if distance[neighbour] > distance[node] + graph[node][neighbour]:
                    distance[neighbour], predecessor[neighbour] = distance[node] + graph[node][neighbou

    # Step 3: Check for negative weight cycles
    for node in graph:
        for neighbour in graph[node]:
            assert distance[neighbour] <= distance[node] + graph[node][neighbour], "Negative weight cyc

    return distance, predecessor

if __name__ == '__main__':
    graph = {
        'a': {'b': -1, 'c':  4},
        'b': {'c':  3, 'd':  2, 'e':  2},
        'c': {},
        'd': {'b':  1, 'c':  5},
        'e': {'d': -3}
    }

    distance, predecessor = bellman_ford(graph, source='a')

    print distance

    graph = {
        'a': {'c': 3},
        'b': {'a': 2},
        'c': {'b': 7, 'd': 1},
        'd': {'a': 6},
    }

    distance, predecessor = bellman_ford(graph, source='a')

    print distance
```

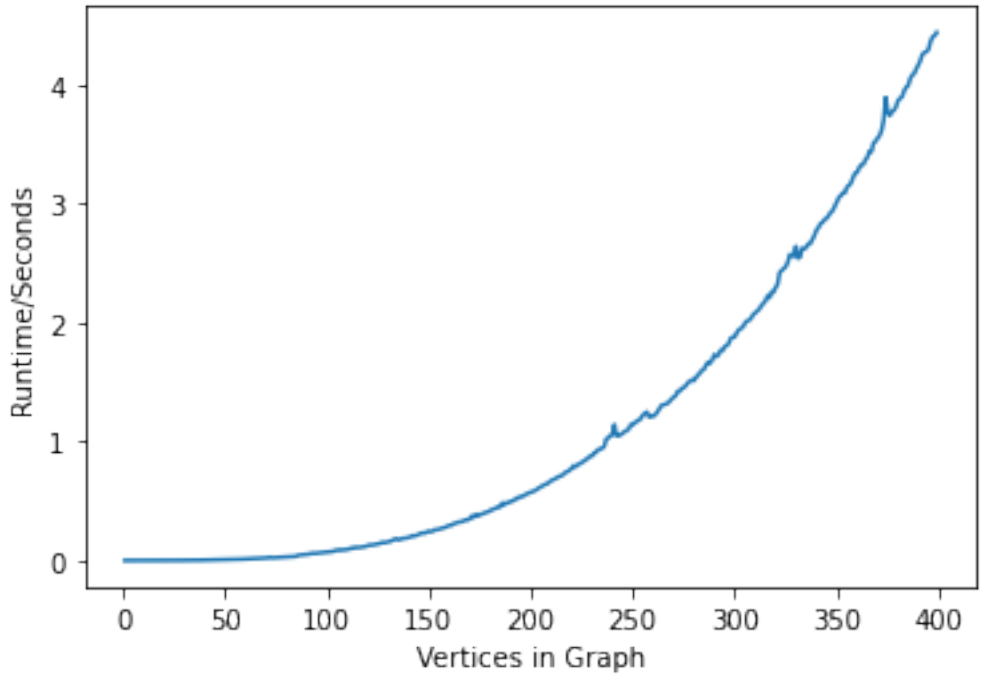
(Ng)

3.2.4 Theoretical & Empirical Analysis

The theoretical analysis of the algorithm comes out to be $O(VE)$, where V and E are the number of vertices and edges in a graph, respectively. The derivation is show below:

- Initialization takes O(V) time
- Inner loop takes O(E (V-1)) time or O(E V) time
- Detection of negative cycles takes O(E) time
- Overall time taken by the algorithm: O(V E)

After running the code on our own machine, we obtained the following graph:



3.2.5 Practical Applications of Bellman Ford's Algorithm

- Route Planning for Electric Vehicle Efficiency

  Previously, route planning problems have been dealt with by the use of Dijkstra's algorithm. This algorithm has proved to very resourceful for engineers as it not only had a low computation complexity but also solved the problem of shortest path correctly for all routes. However, with the technological advancements we see, electric and driver-less cars have entered the scenario.

  These electric vehicles come with the ability to recharge themselves through either electric power or solar power converted to electrical energy. This rechargeable quality of electric vehicles has allowed for decreased routing cost, and therefore the energy costs (or weights) for traversing over an area or path can now be negative.This is where the Bellman Ford algorithm comes into play. With negative costs part of the route, it is better to work with Bellman Ford as it not only computes shortest path for negative weights but also in optimal time. This algorithm coupled with an embedded GPU can help effective route planning for different vehicles.

  (Schambers et al.)

- Identification of Optimal Path in Power System Network using BFA

It is a common occurrence for power system networks to undergo outages that result in loss of power and blackouts. These blackouts can prove to be detrimental to a number of industries and systems. In such cases, restoration of power needs to be done via an optimal path that decreases losses and cost, and increases efficiency of the supply in the network. Choosing an optimal path for power transmission is also essential to reconsider the usable components of the power system. Therefore, for this problem Bellman Ford's Algorithm is a perfect choice to find the optimal path for power transmission as it works with the lowest possible costs and losses in the system and allows the network to be built on a system of stability. The running time of this algorithm makes it even more appealing as it is optimal and can be optimized further to yield faster results.

The inputs that need to be considered in this system of power networking is the capacity of transmission line, voltage stability and a balance between demand and supply of power. Since there could be negative costs in this system, for example, negative priority of load at some edge, therefore Bellman Ford works to find an optimal path for power transmission and is also used to find alternate paths.

(Hemalatha, and Valsalal)

3.3 In Conclusion: Dijkstra v/s Bellman Ford

To recap, here is a table of comparison and contrast between the two algorithms.

	Dijkstra	**Bellman Ford**
Greedy algorithm?	Yes, prioritizes local optimum	No, prioritizes global optimum
Works with negative edges	Can not work with negative edges	Can work with negative edges
Works with negative cycles	Can not work with negative cycles	Can not work with, but can detect.
Theoretical Analysis	O(V + E log E) [faster]	O(V E) [slower]
Empirical Analysis (seconds)	0.0079322 [faster]	0.0540228 [slower]

In conclusion, Dijkstra's algorithm is much more efficient than Bellman Ford, however, it is not as applicable as the latter, as negative weighted edges are not an uncommon occurrence in graph theory, and may play a huge role in practical usages.

3.3.1 When to use Dijkstra's

Dijkstra's algorithm is the optimal choice, given that:

- Low time complexity is the main priority.
- It is certain that the graph does not have negative weights.

3.3.2 When to use Bellman-Ford

Bellman-Ford proves to be the better alternative if:

- The given graph may comprise of negative weights.
- The given graph is dense, where E is close to V^2. In this case, Dijkstra's time complexity translates to $O(V + V^2 Log(V))$, which means the total time complexity can reach as high as $O(V^2 + V^3 Log(V))$

3.4 Further Study: All Pairs Shortest Path Problems

The shortest path problem can be tackled with two different approaches. These approaches are the Single-Source Shortest Paths (SSSP) approach and the All-Pairs Shortest Path (APSP) approach. We have already discussed SSSP in detail with the analysis of Dijkstra's algorithm and Bellman Ford's algorithm. Both these algorithms compute the shortest path from one source vertex s to the rest of the vertices in the graph. It should be noted that no shortest path algorithm works on negative cycles, be it SSSP or APSP. However, if we need to find the distance from each node in the graph to every other node, we use the All-Pairs Shortest Path approach. This approach is made specifically to cater to such needs. In this approach, all vertices act as source node and the target node as well.

One example of such a kind of algorithm is the Floyd-Warshall algorithm. This algorithm is the work of 4 individuals Roy, Kleene, Warshall and Floyd. However it is commonly known the name Floyd-Warshall's algorithm. This is because the basis of the algorithm came from Warshall's work whereas Floyd generalized this discovery to give us the algorithm as it is today. This algorithm works by adding a third parameter to the dynamic programming. This parameter is r and signifies the number of vertices present within a path from a source node u and target node v. These vertices are also called intermediate vertices and a for loop is run over all edges, vertices and intermediate vertices in a graph to find the all-pairs shortest path distances from this algorithm. Due to the three for loops that run simultaneously in the algorithm, the average running time of the algorithm is $O(V^3)$.

3.4.1 Floyd-Warshall Algorithm

```
floyd-warshall(V,E,w):
    for all vertices u
        for all vertices v
            dist[u,v,0]<- w(u->v)
    for r <- 1 to V
        for all vertices v
            for all vertices u
                if dist[u,v,r-1]< dist[u,r,r-1]+ dist[r,v,r-1]
                    dist[u,v,r]<- dist[u,v,r-1]
                else
                    dist[u,v,r]<- dist[u,r,r-1]+dist[r,v,r-1]
```

("Floyd-Warshall Algorithm — Brilliant Math Science Wiki")

3.4.2 Floyd-Warshall vs Bellman Ford's

If we try to use the Bellman Ford algorithm to find the all-pairs shortest path then we will have to run the Bellman Ford's algorithm on every single vertex. By running this algorithm over all vertices, we can make all vertices our source as well as target vertex. This means that we will need to run Bellman Ford's V times. Therefore, the running time of using Bellman Ford to compute all-pairs shortest path is $O(EV^2)$. Compared to the running time of Floyd-Warshall's algorithm, Bellman Ford proves to work faster when used to solve the all-pairs shortest path therefore this algorithm is an optimal algorithm when working with any set of input except for negative cycles.

The reason for choosing Bellman Ford's to solve all-pair SP instead of Dijkstra's is that Dijkstra is limited by constraints and is only suitable for use when you're sure that there are no negative weights present. In case Dijkstra is made to run over an input with negative weights it might increase the running time complexity to an exponential function or the algorithm will fail to work completely.

This is why negative edges and cycles have been stressed upon so much in our report. The wrong input for any algorithm can either make it fail and not give any result at all or it can take a really long time to compute the shortest paths. Therefore it is always wise to know your input before you choose an algorithm to find the shortest path problem.

4 References

- Bang-Jensen, J. and Gutin, G., 2009. Digraphs. London: Springer.

- "Floyd-Warshall Algorithm — Brilliant Math Science Wiki". Brilliant.Org, 2020, https://brilliant.org/wiki/floyd-warshall-algorithm

- Giraud, Vincent. Shortest Path. 2019, https://datascience.lc/2019/10/26/shortest-path-dijkstra-algorithm/. Accessed 4 Dec 2020.

- Hemalatha, S., and P. Valsalal. "Identification Of Optimal Path In Power System Network Using Bellman Ford Algorithm". Modelling And Simulation In Engineering, vol 2012, 2012, pp. 1-6. Hindawi Limited, Accessed 7 Dec 2020.

- Ng, Daniel. "Bellman-Ford Algorithm In Python". Github, 2013, https://gist.github.com/ngenator/6178728. Accessed 7 Dec 2020.

- "Overleaf, Online Latex Editor." Overleaf.com. Web. 4 Dec. 2020.

- Richards, Hamilton. EDSGER WYBE DIJKSTRA. ACM, 2019, https://amturing.acm.org/award_winners/dijkstra_1053701.cfm. Accessed 5 Dec 2020.

- Schambers, Adam et al. "Route Planning For Electric Vehicle Efficiency Using The Bellman-Ford Algorithm On An Embedded GPU". 2018 4Th International Conference On Optimization And Applications (ICOA), 2018. IEEE, Accessed 7 Dec 2020.

- Schrijver, A., 2005. On The History Of Combinatorial Optimization (Till 1960). [ebook] Available at: https://homepages.cwi.nl/ lex/files/histco.pdf [Accessed 7 December 2020].

- "Shortest Path With Dijkstra'S Algorithm". Bradfieldcs.Com, https://bradfieldcs.com/algos/graphs/dijkstras-algorithm/. Accessed 30 Nov 2020.

- Sryheni, S., 2020. Dijkstra's Vs Bellman-Ford Algorithm — Baeldung On Computer Science. [online] Baeldung on Computer Science. Available at: ¡https://www.baeldung.com/cs/dijkstra-vs-bellman-ford¿ [Accessed 7 December 2020].

- Xu, Yuanzhe et al. "The Application Of Dijkstra's Algorithm In The Intelligent Fire Evacuation System". 2012 4Th International Conference On Intelligent Human-Machine Systems And Cybernetics, 2012. IEEE, Accessed 7 Dec 2020.