



NANYANG
TECHNOLOGICAL
UNIVERSITY

Nanyang Technological University
School of Electrical and Electronic Engineering (EEE)

EE 6108 PROJECT REPORT

Name:

Jerald Yik Jia Ler

Date:

31/10/20

1. Introduction

This report will cover brief descriptions of Dijkstra's and Bellman-Ford algorithms, a more in-depth explanation of my implementation of both algorithms, as well as possible variations of the two aforementioned algorithms.

Both Dijkstra's and Bellman-Ford algorithms are single-sourced shortest path algorithms, which in the case of Computer Networks, routing algorithms. The difference between the two algorithms is that Dijkstra's algorithm disallows negative edges. However, in the presence of negative cycle(s), Bellman-Ford algorithm would also be rendered useless. Therefore, do note that should Dijkstra's algorithm & Bellman-Ford algorithms be carried out on the same graph with no negative edges, the outputs should both be identical.

The time complexity of Dijkstra's algorithm is $O((|V|+|E|) \log|V|)$. The time complexity of Bellman-Ford algorithm is $O(|V||E|)$. In the event of a fully connected graph, i.e. $|E| = O(|V|^2)$, then the time complexity would be $O(|V|^3)$.

2. Explanation of Implementation

The language I have used to code my implementation in is `Javascript`, more specifically `Typescript` for the implementations of the algorithms. The commands to run the test and driver scripts can be found in `package.json`. Before running any of the scripts, please ensure that you have installed the relevant packages by running `npm install`.

The driver script is structured in the way, such that user can input their selections and data on command line, according to the specific instructions. The driver script imports the main algorithm scripts and executes based on the user's input.

2.1 Explanation of testing case

The test scripts contain hard-coded graphs (in which a visualisation can be found in *sample-graphs.pdf*). The test scripts contain an example each as a general proof-of-work, as well as added edges to display the limitations of the respective algorithm. For example, in the case of Dijkstra's algorithm:

- A sample graph is introduced with no negative edges.
- The shortest-path algorithm is ran, and a shortest path (with a given start vertex and end vertex), together with the corresponding total weight are gathered.
- After that, an edge with negative weight is introduced and the algorithm will throw an error.

Next, in the case of Bellman-Ford algorithm:

- A sample graph is introduced with some negative edges, but with no negative cycles.
- The shortest-path algorithm is ran and likewise produces a shortest path and its corresponding total weight.
- After that, a negative edge is introduced, such that a negative cycle is present in the graph. The shortest-path algorithm is ran again, and an error would be thrown.

2.2 Explanation of the algorithms

The explanation of my implementation of Dijkstra's and Bellman-Ford algorithms can both be found as comments embedded in the code itself, as well as flowcharts present in the directory, labelled *Dijkstras-Flowchart.pdf* & *Bellman-Ford-Flowchart.pdf*. The comments in the code are structured as such:

- On the top of each code file, a pseudocode of the algorithm is sketched out. The pseudocodes are taken from *Algorithms (PDV) Textbook*.
- All of the important lines of code have comments above them. Each line of the pseudocodes mentioned are also included as comments.

3. Variations of Dijkstra's & Bellman-Ford algorithms

3.1 Variation of Dijkstra algorithm

One of the variations we can make to the Dijkstra's algorithm is to add some form of heuristics, to add to the weight of each node to the start node. Such an example would be called the A* algorithm, which is actually a generalisation of the Dijkstra's algorithm¹. For example, one heuristic could be the Euclidean distance from the start node to the target node. As such, a node with a very high Euclidean distance from the start node could end up with a very high overall weight, even though it has a low absolute weight, as calculated via the edges. Also, do note that the heuristic is encouraged not to produce a negative value, as it could potentially lead to a negative edge after calculation, which is shown further below.

Such an implementation could be beneficial in a way, whereby it could potentially cut down on the size of the subgraph that have to be explored, since Dijkstra's algorithm aims to cover every possible node, given the edges, in order to find the best shortest path out of all combinations. In the case of calculating Euclidean distance from the start vertex, the shortest path chosen could very well not have much deviation, should a straight line be drawn from the start node to the end node. This works quite well in the context of Computer Networks, as such a heuristic could shorten propagation bandwidth.

We can add this heuristics in the condition when comparing between the current weight of the node from the start, with the current weight of the previous node + the weight of the edge. Below is a screenshot of the current implementation:

```
// calculateWeight = dist(u) + l(u,v)
const calculateWeight: number =
  | currentVertex.weightFromStart + dest.weightOfEdge;
// if dist (v) > dist(u) + l(u,v); calculateWeight < dist(v)
if (
  | calculateWeight <
  | pool[dest.nameOfDestVertex].vertex.weightFromStart
) {
```

¹ Madkour, A., Aref, W., Rehman, F., Rahman, M., & Basalamah, S. (2017, May 8). A Survey of Shortest-Path Algorithms. <https://arxiv.org/pdf/1705.02044.pdf>

We can change the `calculateWeight` to include the heuristic, e.g.

```
calculateWeight = currentVertex.weightFromStart +  
dest.weightOfEdge + currentVertex.heuristicValue
```

To make such an implementation work, the heuristic chosen must be monotone (admissible), i.e. estimated distance must always be less than or equal to the actual distance.

One disadvantage with such a variation is that firstly, it is difficult to find a perfect heuristic to apply (as the name 'heuristic' suggests), that could not only reduce the time complexity of searching for the actual shortest path, but also instead not to find a longer actual path. This is because should the heuristic chosen and calculated give a weight to a node that is wildly different from the actual weight calculated from weights on edges, then the algorithm might not produce the optimal path. It is then better to stick to the original Dijkstra's algorithm implementation.

Another disadvantage could be the calculation of the heuristic taking too long, therefore increasing the time complexity of the overall implementation. Heuristics like Euclidean distance should not be too complex to calculate, given the appropriate data, especially in Computer Networks, whereby routers have their easily-obtainable physical locations. There could be other forms of heuristics that might not only provide somewhat accurate information, that complement the actual weights derived from edges, but are difficult to obtain and calculate. As such, there exist a trade-off and hence before implementation, one could consider and analyse carefully on the type of heuristic to utilise.

3.2 Variation of Bellman-Ford algorithm

In my implementation, before running the algorithm, I chose a start node to calculate the weights of all other edges from. However, since Bellman-Ford is traditionally used as a Distance Vector Routing Protocol, for example the Routing Information Protocol (RIP), whereby each vertex holds a table of smallest weights from itself to all other nodes, this drastically increases the space complexity required. In other words, the actual implementation of Bellman-Ford would have $|V|$ times the amount of space complexity as my implementation. As such, Bellman-Ford is not necessarily a scalable algorithm, both space-wise and even time-wise (remember it has time-complexity of $O(|V|^3)$).

As such, one variation would be to set a certain threshold to the weights a vertex should have from another vertex¹. For example, when vertex B has calculated a weight from vertex A that is higher than the threshold, then this entry in the table of vertex B could be dropped. On first glance, this looks like the space-complexity could be improved, since during the processing of the algorithm, memory could be freed up from "impossible" routes. However, time-complexity could also be improved, since calculations would not have to be made for that certain combination of nodes.

Referring back to the screenshot of the conditional:

```
// calculateWeight = dist(u) + l(u,v)
const calculateWeight: number =
  currentVertex.weightFromStart + dest.weightOfEdge;
// if dist (v) > dist(u) + l(u,v); calculateWeight < dist(v)
if (
  calculateWeight <
  pool[dest.nameOfDestVertex].vertex.weightFromStart
) {
```

This variation could be demonstrated in my implementation by adding a line of conditional before the if statement in the screenshot:

```
if (calculateWeight > threshold){ remove from pool }
```

However, the disadvantage of such implementation is glaringly obvious, in that time perhaps further down the iteration, the weight calculated could be lowered due to cascading effects from neighbouring nodes. This means that the route is wrongfully dropped and could actually be reached.

4. Conclusion

In conclusion, Dijkstra's and Bellman-Ford algorithms do definitely have their merits; that is why they are so widely used and in the context of Computer Networks, Dijkstra's algorithm being used as a Link State Routing Protocol, and Bellman-Ford algorithm being used as a Distance Vector Routing Protocol. Over the years, extensive research has been carried out to further improve existing algorithms, to formulate more space and time efficient solutions. However for now, Dijkstra's and Bellman-Ford algorithms would still be used as the standard shortest-path searching algorithm in the context of Computer Networks.