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 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 contains 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.

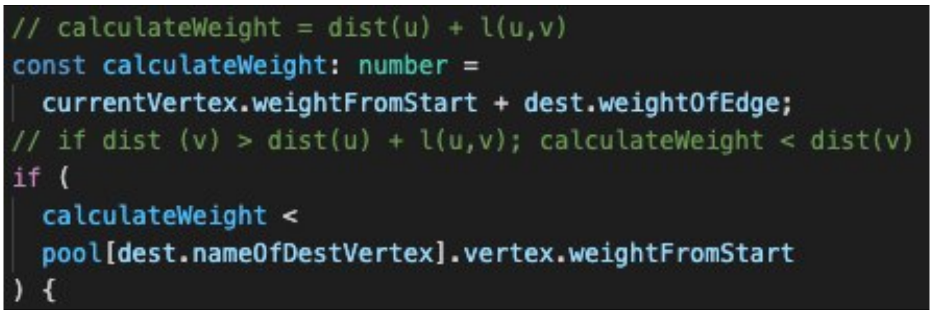
3. Variations of Dijkstra's & Bellman-Ford algorithms

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 algorithm[1]. 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/. 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

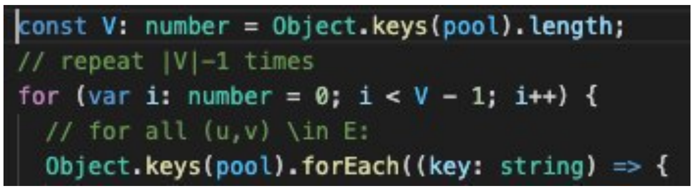
With every iteration of the Bellman-Ford algorithm, an arbitrary linear ordering of vertices is used for re-calculating each of the weights of the vertices from the start node. While this is a very standard approach in determining the order of iteration, one proposal would be to utilise a random ordering[1].

A random ordering in every iteration could potentially result in a fewer number of iterations. With a linear ordering of iterating through the nodes, changes are trickled down via neighbours, which might take many iterations, especially if the changes are bunched up at a small portion of the graph, or if the graph is large and sparse. With a random ordering of vertices, changes can be propagated throughout the graph at a much faster rate.

Not only that, we can limit the number of iterations by setting an arbitrary threshold[1], in which if the number of vertices in each iteration that has their weights and previous node variables changed, then the algorithm is terminated, and the shortest path between the start node and the determined end node is found. This is similar to the RIP Operation, which utilises Bellman-Ford algorithm. Instead of terminating and determining the shortest path, RIP sets the weight of that node to be infinite at the 16th hop, hence referring to that node as 'unreachable'. The total weight of the found 'shortest' path would most probably not be extremely accurate. However, in the context of Computer Networks, we are usually only trying to find the shortest path, and not the corresponding total weights. Of course, the algorithm can only be terminated if all nodes have been explored, which would be case after at least 1 iteration.

With the two aforementioned variations, the number of iterations carried out by the algorithm may be able to reduce drastically. Utilising such an approach would almost always reduce the number of iterations required (fewer than |V|-1 iterations). Of course, we should still be wary of the fact that the shortest path returned might not be the actual shortest path, given that the algorithm is now structured in a less precise fashion.

The screenshot below shows the portion of my code implementation, encapsulating both the necessary |V|-1 times of iteration, and the set linear fashion of iterating through every node:



```
const V: number = Object.keys(pool).length;
// repeat |V|-1 times
for (var i: number = 0; i < V - 1; i++) {
  // for all (u,v) \in E:
  Object.keys(pool).forEach((key: string) => {
```

For randomising the iteration of each node, we can first retrieve the keys of `pool` as an array and randomise it, and then using this randomised array of keys, iterate through the `pool` object:

```
const keys = Object.keys(pool);
// using a meta-function
keys.shuffle();
keys.forEach(key => { ... }); // perform algorithm
```

What we can do for setting a threshold, would be to set a counter at the start of every iteration, right below the `for loop`, and increment the counter every time a change of weight and previous node is made. When the counter exceeds the threshold, exit the iteration. One might notice that the checking of a negative cycle is not carried out in this variation. This is one glaring disadvantage of this variation, however in the context of Computer Networks, there should most likely not be a negative edge, hence this issue should not cause much of a problem.

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.