Dijkstra's Algorithm – Single Source Shortest Path Report

Krish Veera and Jonathan Koch

Abstract

The following report analyses the Dijkstra's algorithm and elaborates on the theoretical and coding implementation of the same using basic data structures like a Heap Queue and a Graph.

Introduction

Dijkstra's Algorithm is a set of rules or a set of instructions to be followed to find the shortest path to search through a weighted graph from the source node to other nodes in the graph. An excellent application of the Dijkstra's algorithm in real life is when the Maps application on people's smart phones calculates the shortest path from location A to location B, or even in social networking applications where the algorithm calculates if individual A might know individual B given the proximity of their locations.

Theoretical Implementation of the Dijkstra's Algorithm

1) The programmer initializes a value called as Dijkstra's value, which essentially keeps a track of the amount of weight that the programmer has gone through on their journey from point A to point B.
2) For the Dijkstra's algorithm the programmer then begins at the source node and assigns it a value of 0, and the distance to the other nodes is marked as ∞ since the programmer has not been through any nodes and thus has not encountered any weightage.
3) The algorithm then peeks at all the edges and the corresponding weightage of those at node that the algorithm is on. It then determines the shortest path by comparing the weightage and choosing the smallest value using a priority queue.
4) Once the shortest path to the source node has been found the node that it is currently on is marked 'visited' and is then added to the path.
5) This process continues until the end node, or the destination has been reached.

Limitation

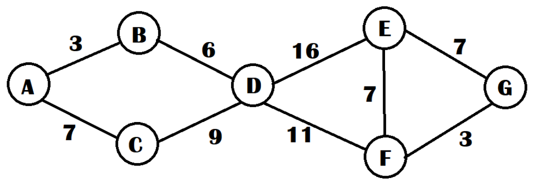
The Dijkstra's algorithm can only work with positive weights of edges. Therefore, it can only be implemented on a non-negative weighted graph.

Time Complexity

O (N + E * log(N)) where N is the number of nodes in the graph, and E is the number of edges in the graph. The reason for the same will be further explained in the next section of the report.

Example of the implementation

Given the following graph:



The distances from Node A to all other nodes is evaluated and initially all the distances are marked as ∞. And A is marked as visited.

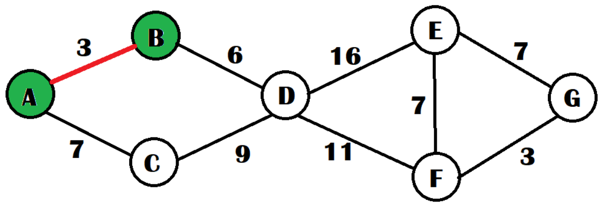
Therefore, the distance table looks something like this:

A	B	C	D	E	F	G
0	∞	∞	∞	∞	∞	∞

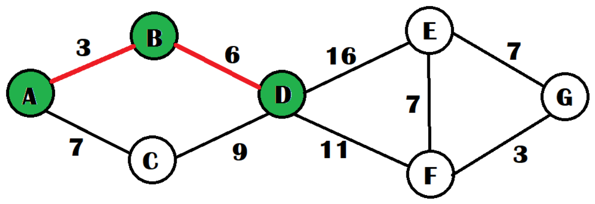
Next the two edges stemming from A are analyzed and are evaluated giving the following:

A	B	C	D	E	F	G
0	3	7	∞	∞	∞	∞

Out of these the smaller of the two, 3 in this case, is taken and is then followed through to the node on the other end of the edge which is B and B is then marked visited giving a graph like below.



Next the edge for B is analyzed and since there is only one edge, that is taken as the next link in the path giving the following distance table and modified graph.

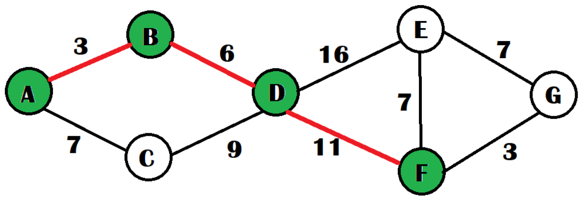


A	B	C	D	E	F	G
0	3	7	3 + 6 = 9	∞	∞	∞

The process repeats and the edges connected to D are analyzed and the distances to the nodes are updated.

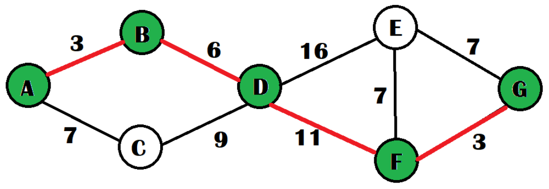
A	B	C	D	E	F	G
0	3	7	9	9 + 16 = 25	9 + 11 = 20	∞

The smaller distance of the two is taken, 20, and the node on the other side of the edge, F, is marked as visited.



Finally, edges are again analyzed and the lower of the two is taken, the distances are updated, and the node is added to the path after being marked visited.

A	B	C	D	E	F	G
0	3	7	9	25	20	20 + 3 = 23



Therefore, the shortest path from A to G is:

A -> B -> D -> F -> G

And this path had the total weight of 23.

Algorithm Implementation and Explanation

```
unsigned long Graph::shortestPath(std::string startLabel, std::string endLabel,
std::vector<std::string> &path) {
    GraphNode* startingNode = this->getVertex(startLabel);
    GraphNode* endingNode = this->getVertex(endLabel);
    /* node does not exist in list */
    if (startingNode == nullptr || endingNode == nullptr) return INFINITY;
```

The above code checks whether there is an appropriate graph to implement the algorithm on, if either the start or end node are NULL, it returns infinity.

```
HeapQueue<Path*, Path::PathComparer> paths;
std::map<std::string, unsigned long> dijkstraKey;
for (auto it = this->nodes.begin(); it != this->nodes.end(); ++it) { /* O(N)*/
   if (it->first != startLabel) {
       dijkstraKey[it->first] = INFINITY;
      }
}
dijkstraKey[startLabel] = 0;
paths.insert(new Path(startingNode));
```

This initializes the all the distances from the source node to the other nodes as infinity and initializes a HeapQueue that will store the shortest path that will be encountered.
Time Complexity: Since there is a for loop that goes through all the nodes in the graph, it is O(N)

```
while (paths.min()->current()->id != endLabel) {
        Path* shortestPath = paths.min();
        paths.removeMin();
        for (auto const& [currID, currLink]: shortestPath->pathsFrom()) { /* Each Edge in the
Current Visiting Node O(Edges) */
            if (! shortestPath->nodeVisited(currID) ) {/* Dont Revisit old nodes*/
                Path* newLocation = new Path(*shortestPath); /* Copy the old Path */
                newLocation->visit(currLink->to); /* ... And visit the new Node for that
Path  */
                unsigned long previousDistance = dijkstraKey[currID];
                /* Update the key along the way, and add the node if weve found a shorter
path to another node. */
                if (newLocation->pathWeight() < previousDistance ) {
                dijkstraKey[currID] = newLocation->pathWeight();
                paths.insert(newLocation); /* O(log(N))*/
                }
            }
        }
    }
```
The while loop iterates till it has reached the destination or the end node and within each iteration it implements a for loop that iterates over every single edge that the node the algorithm is currently on. It then checks which of the edges or links has the lowest weight and then adds this as the new path for the algorithm and also marks the node as visited so that later on it isn't visited again and this node is added back to the heap queue that was implemented as a means to keep track of the nodes present on the shortest path.

```
Path* finalPath = paths.min();
paths.removeMin();
path = finalPath->toString();
return dijkstraKey[endLabel];
}
```
The final path that was taken is the shortest path from the source node to the destination node, it is then converted to a path string, a vector of IDs, and the final weight encountered which is the total weight of the shortest path in the DijkstraKey is returned.

The time complexity of this entire algorithm as given before is O (N + E*log(N)) where N is the number of nodes in the weighted graph and E is the number of edges in the weighted graph, because the first for loop that initializes all the distances from the source node to infinity takes the time complexity of O (N) and this is added to the time complexity of the while loop which takes O (E * log(N)). This is because using the greedy approach you end up visiting all the edges in the log(N) nodes, where log(N) is the time that it takes to remove from the Heap Queue.

References

3.6 Dijkstra Algorithm - Single Source Shortest Path - Greedy Method. (n.d.). Www.youtube.com. Retrieved November 20, 2022, from https://youtu.be/XB4MIexjvY0

Dijkstra's algorithm in 3 minutes. (n.d.). Www.youtube.com. Retrieved November 20, 2022, from https://www.youtube.com/watch?v=_lHSawdgXpI&t=1s&ab_channel=MichaelSambol

GeeksforGeeks. (2018, November 19). *Dijsktra's algorithm*. GeeksforGeeks. https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/

Korzhova, Valentina. Dijkstra's Algorithm. 2022. University of South Florida. https://usflearn.instructure.com/courses/1728058/assignments/12404589?module_item_id=26745047

Navone, Estefania Cassingena. "Dijkstra's Shortest Path Algorithm - a Detailed and Visual Introduction." *FreeCodeCamp.org*, 28 Sept. 2020, www.freecodecamp.org/news/dijkstras-shortest-path-algorithm-visual-introduction/.