5 Shortest & 5 Longest Path Algorithms

Daniel E. Hernández - 20180077

2019-04-4

Abstract

The following documents describes multiple algorithms to achieve a total sum of weights of paths from one node to either another one or all other nodes. It contains four *Shortest Path Algorithms* and one *Longest Path Algorithm*.

Part I
Shortest Path Algorithms

1 ... in a Directed Acyclic Graph *(DAG)*

1.1 Explanation

Given that it has a linear time, we use topological sort to find the distance of a signle source to all other nodes[2].

1.2 Complexities

$O(m+n)$

1.3 Algorithm Steps

1. Initialize $dist[] = INF, INF, ...$ and $dist[s] = 0$ where s is the source vertex and INF is ∞
2. Create a topological order of all vertices.
3. Do the following for every vertex u in topological order.
    - (a) Do the following for every adjacent vertex v of u
        - i. if $(dist[v] > dist[u] + weight(u, v))$
            - A. $dist[v] = dist[u] + weight(u, v)$

1.4 Advantages

- Linear time (can me a disadvantage too, depends on usage)
- Easy to code

1.5 Applications

- Scheduling
- Data processing networks
- Causal structures
- Genealogy and version history
- Citation graphs
- Data compression

2 Bellman Ford's Algorithm

2.1 Explanation

Bellman Ford's algorithm is used to find the shortest path from the source vertex to all other vertices in a weighted graph. It depends on the following principle: the shortest path from any given vertex to another, must have at most *n - 1* edges (where n is the total number of *vertex*). That is because the *shortest path* shouldn't have a cycle. The graph may contain negative weight edges. This algorithm is built upon the relaxation principle which means that it will be changing the shortest distance value from a less accurate to a more accurate one once it satisfies the need to do so[1].

2.2 Complexities:

$O(V \cdot E)$ in case $E = V^2$

2.3 Algorithm Steps

1. Traverse in an outer loop from 0 to *n - 1*
2. Loop over all edges
3. If the next node distance > current node distance + edge weight:
    (a) Update the next node distance to current node distance + edge weight

2.4 Advantages

- It may be improved in practice (although not in the worst case) by the observation that, if an iteration of the main loop of the algorithm terminates without making any changes, the algorithm can be immediately terminated, as subsequent iterations will not make any more changes.
- It can have negative weights.

2.5 Applications

- Distance-vector routing protocol

3 Dijkstra's Algorithm

3.1 Explanation

The more common Dijkstra's algorithm sets a node as the *source* node and finds the shortest paths from said source to all other vertices in the graph by adding up their edges' weights and assigning said sum to the next node and so on[1].

3.2 Complexities

$O(V^2)$ but with min-priority queue it drops down to: $O(V + E \cdot logV)$

3.3 Algorithm Steps

1. Set all vertices distances to ∞ except the source vertex, said vertex is set to 0.
2. Push the source vertex in a min-priority queue in the form (distance, vertex), as the comparison in the min-priority queue will be according to vertices distances.
3. Pop the vertex with the minimum distance from the priority queue (at first the popped vertex = the source).
4. Update the distances of the connected vertices to the popped vertex in case of "current vertex distance + edge weight < next vertex distance".
5. Then push the vertex with the new distance to the priority queue.
6. If the popped vertex is visited before, just continue without using it. Apply the same algorithm again until the priority queue is empty.

4 Floyd–Warshall's Algorithm

4.1 Explanation

A single execution of the algorithm will find the summed lengths of the shortest path between all pairs of vertices.

4.2 Complexities

$O(V^3)$

4.3 Algorithm Steps

1. Initialize the shortest paths between any 2 vertices with ∞.
2. Find all pair shortest paths that use 0 intermediate vertices, then find the shortest paths that use 1 intermediate vertex and so on.. until using all N vertices as intermediate nodes.
3. Minimize the shortest paths between any 2 pairs in the previous operation.
4. For any 2 vertices *(i,j)*, one should actually minimize the distances between this pair using the first K nodes, so the shortest path will be: $min(dist[i][k] + dist[k][j], dist[i][j])$.

4.4 Advantages

- It is a good idea to use this algorithm with dense graphs (i.e. when almost all pairs are connected by edges)

4.5 Applications

- Finding a regular expression denoting the regular language accepted by a finite automation.
- Inversion of real matrices.
- Optimal routing. In this application one is interested in finding the path with the maximum flow between two vertices. This means that, rather than taking minima as in the pseudocode above, one instead takes maxima. The edge weights represent fixed constraints on flow. Path weights represent bottlenecks; so the addition operation above is replaced by the minimum operation.
- Fast computation of Pathfinder networks.
- Computing the similarity between graphs

Part II

Longest Path Algorithm

5 ... in a Directed Acyclic Graph *(DAG)*

5.1 Explanation

The longest path problem for a general problem is harder to find because is does not have an optimal solution. However, for a directed graph it does have an optimal solution and it is linear.

5.2 Complexities

$O(m + n)$

5.3 Algorithm Steps

1. Initialize $dist[] = NINF, NINF, ...$ and $dist[s] = 0$ where s is the source vertex and *NINF* is $-\infty$
2. Create a topological order of all vertices.
3. Do the following for every vertex u in topological order.
   (a) Do the following for every adjacent vertex v of u
      i. if $(dist[v] < dist[u] + weight(u, v))$
         A. $dist[v] = dist[u] + weight(u, v)$

References

[1] *Shortest Path Algorithms Tutorials & Notes | Algorithms*, en, `https://www.hackerearth.com/practice/algorithms/graphs/shortest-path-algorithms/tutorial/`.

[2] *Shortest Path in Directed Acyclic Graph*, en-US, `https://www.geeksforgeeks.org/shortest-path-for-directed-acyclic-graphs/`, May 2013.