A Comparative Study between Bellman-Ford Algorithm and Dijkstras- Algorithms

Rochan Mehrotra, Sarthak Budhiraja
Information Technology
Dronacharya College Of Engineering, Haryana

***Abstract*- Packet switching is an approach used by home computer network protocols to deliver data across local or long distance connection. Packet switching entails packaging data in small unit called packet that are routed from source to destination using network switch & routers. In this article we made a study about the two well known shortest path searching algorithms, which are used in routing. They are Bellman-Ford algorithm and Dijkstra's algorithm. The analysis of the comparison is given briefly.**

I. INTRODUCTION

Today, an internet can be so large that one routing protocol can't handle the task of updating the routing table of all routers. For this reason internet is divided in to an autonomous systems.

The routing inside an autonomous system is called as intra domain routing. And the communication between autonomous systems is called inter domain routing. Bellman-Ford algorithm and the Dijkstra's algorithms are two popular algorithms used in intra domain routing to update the routing tables.

II. BELLMAN-FORD ALGORITHM

The Bellman-Ford algorithm uses relaxation to find single source shortest paths on directed graphs. And it is also contain negative edges. The algorithm will also detect if there are any negative weight cycles (such that there is no solution). If talking about distances on a map, there is no any negative distance. The basic structure of bellman-ford algorithm is similar to dijkstra algorithm. It relaxes all the edges, and does this |V| - 1 time, where |V| is the number of vertices in the graph . The repetitions allow minimum distances to accurately propagate throughout the graph, since, in the absence of negative cycles, the shortest path can only visit each node at most once. Unlike the greedy approach, which depends on certain structural assumptions derived from positive weights, this straightforward approach extends to the general case.

A. Procedure

```
Bellman-Ford (list vertices, list edges, vertex
source)
// This implementation takes in a graph, represented
as lists of vertices
// and edges, and modifies the vertices so that their
distance and
// predecessor attributes store the shortest paths.
// Step 1: Initialize graphe
for each vertex v in vertices:
if v is source then v.distance := 0
else v.distance := infinity
v.predecessor := null
// Step 2: relax edges repeatedly
for i from 1 to size(vertices)-1:
for each edge uv in edges:
 u := uv.source
v := uv.destination
// uv is the edge from u to v
if v.distance > u.distance + uv.weight:
v.distance:= u.distance + uv.weight
v.predecessor:= u
// Step 3: check for negative-weight cycles
for each edge uv in edges:
u := uv.source
v := uv.destination
If v.distance > u.distance + uv.weight:
error "Graph contains a negative-weight
cycle"
```

B. Bellman-Ford Algorithm

```
Input:
Edge edges[], int edgecount, int nodecount,
int source
Output: Routing table
Begin:
 {
int *distance ;// Should be allocated
int i, j;
if (distance == NULL) Then
{
fprintf (stderr, "malloc () failed\n");
exit (EXIT_FAILURE);
}
for (i 0; i < nodecount; ++i)
distance[i] INFINITY;
distance[source] 0;
for (i 0; i < nodecount; ++i)
{
```

```
for (j 0; j < edgecount; ++j)
{
if (distance [edges[j].source] != INFINITY)
{
int new_distance
distance[edges[j].source] +
edges[j].weight;
If (new_distance < distance[edges[j].dest])
distance[edges[j].dest] = new_distance;
}
}
}
for (i 0; i < edgecount; ++i)
{
If (distance[edges[i].dest] >
distance[edges[i].source] + edges[i].weight)
{
puts ("Negative edge weight cycles detected!");
free (distance);
return;
}
}
```

Time complexity
Time a 10*n +18*m+27*n*m+3
n - Node count
m - Edge count
i.e Time a n*m
Time a 27 * Edges * Nodes
Time a (number of nodes) (number of edges)
For a general case of number of edges equals to number of nodes we can write(m=n)
Time a 27*n2 + 28*n + 3
i.e **Time a 27n2**

III. DIJKSTRA'S ALGORITHM

Dijkstra's algorithm, conceived by Dutch computer scientist Edsger Dijkstra in 1959, is a graph search algorithm that solves the single-source shortest path problem for a graph with non negative edge path costs, outputting a shortest path tree. This algorithm is often used in routing. For a given source vertex (node) in the graph, the algorithm finds the path with lowest cost (i.e. the shortest path) between that vertex and every other vertex. It can also be used for finding costs of shortest paths from a single vertex to a single destination vertex by stopping the algorithm once the shortest path to the destination vertex has been determined. Dijkstra Algorithm used the method of increasing node by node to get a shortest path tree which makes the starting point as its root.

Here is the concrete method: in the Weighted directed graph, the shortest path node which Starts from the starting point s and reaches the earliest must be the smallest point where all the nodes adjacent to s and its length of arc is
Chord length

A. Procedure

It should be noted that distance between nodes can also be referred to as weight.

1. Create a distance list, a previous vertex list, a visited list, and a current vertex.
2. All the values in the distance list are set to infinity except the starting vertex which is set to zero.
3. All values in visited list are set to false.
4. All values in the previous list are set to a special value signifying that they are undefined, such as null.
5. Current vertex is set as the starting vertex.
6. Mark the current vertex as visited.
7. Update distance and previous lists based on those vertices which can be immediately reached from the current vertex.
8. Update the current vertex to the unvisited vertex that can be reached by the shortest path from the starting vertex.
9. Repeat (from step 6) until all nodes are visited.

B. Dijkstra's Algorithm

1 function Dijkstra (*Graph*, *source*):
2 for each vertex *v* in *Graph*: *// Initializations*
3 dist[*v*]:= infinity *// Unknown distance function from source to v*
4 previous[*v*]:= undefined *// Previous node in optimal path from source*
5 dist[*source*]:= 0 *// Distance from source to source*
6 *Q*: = the set of all nodes in *Graph // All nodes in the graph are unoptimized - thus are in Q*
7 while *Q* is not empty: *//The main loop*
8 *u* := node in *Q* with smallest dist[]
9 remove *u* from *Q*
10 for each neighbour *v* of *u*: *// where v has not yet been removed from Q.*
11 *alt*: = dist[*u*] + dist_between (*u*, *v*)
12 if *alt* < dist[*v*] *// Relax (u ,v)*
13 dist[*v*]:= *alt*
14 previous[*v*]:= *u*
15 return previous []

If we are only interested in a shortest path between vertices *source* and *target*, we can terminate the search at line 10 if *u* = *target*. Now we can read the shortest path from *source* to *target* by iteration:

1 *S* := empty sequence
2 *u* := *target*
3 while defined previous[*u*]
4 insert *u* at the beginning of *S*
5 *u* := previous[*u*]

Time complexity

The running time of Dijkstra's algorithm on a graph with edges *E* and vertices *V* can be expressed as a function of |*Edges*| and |*Vertices*| using the Big-O notation.

The simplest implementation of the Dijkstra's algorithm stores vertices of set *Q* in an ordinary linked list or array, and operation Extract-Min(*Q*) is simply a linear search through all vertices in *Q*. In this case, the running time is $O(|V|2+|E|) = O(|V|2)$.

***O* (|Vertices|2 + |Edges|)**

For a general case of number of edges equals to number of nodes we can write (m=n) i.e. **Time a 2n2** (form the above algorithm)

For a case of n=m we can plot the following graph

Number of **Nodes vs. time**

IV. CONCLUSION

As the analysis shows the Bellman-Ford algorithm soles a problem with a complexity of 27n2 but the Dijkstra's algorithm solves the same problem with a lower running time, but requires edge weights to be non-negative. Thus, Bellman–Ford is usually used only when there are negative edge weights. Both of these functions solve the single source shortest path problem. The primary difference in the function of the two algorithms is that Dijkstra's algorithm cannot handle negative edge weights. Bellman-Ford's algorithm can handle some edges with negative weight. It must be remembered, however, that if there is a negative cycle there is no shortest path.

REFERENCES

[1] en.wikipedia.org/

[2] Jin Y. Yen. "An algorithm for Finding Shortest Routes from all Source Nodes to a Given Destination in General Network", Quart. Appl. Math., 27, 1970, 526-530.

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L.Rivest, and Clifford Stein.*Introduction to Algorithms*, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Section 24.1: The Bellman-Ford algorithm, pp.588–592. Problem 24-1, pp.614–615.

[4] A note on two problems in connexion with graphs. In Numerische Mathematik, 1 (1959), S. 269–271.

[5] "A Study on Contrast and Comparison between Bellman-Ford algorithm and Dijkstra's algorithm" by Thippeswamy.K, Hanumanthappa.J, Dr.Manjaiah D.H.