IMPERIAL COLLEGE LONDON, DEPARTMENT OF MATHEMATICS

Scientific Computing (M3SC)

Peter J. Schmid

January 31, 2017

1 SHORTEST PATH IN GRAPHS (BELLMAN-FORD ALGORITHM)

Dijkstra's algorithm is commonly used in finding the shortest path in weighted graphs. However, the weight assigned to edges, connecting nodes, have to be positive for the algorithm to converge. In many applications, the positivity of the edge values cannot be guaranteed. For example, when modeling the workflow through graphs from manufactoring, resource management or supply chains, certain parts of a path can imply a net gain, rather than a net cost. Dijkstra's algorithm fails in this case. Instead, an alternative algorithm – the Bellman-Ford algorithm – can be used. In contrast to Dijkstra's algorithm, which can be categorized as a greedy algorithm by search for the locally optimal way forward, the Bellman-Ford algorithm is a relaxation algorithm. It updates the travel costs through the graph by repeatedly updating the edges, until the optimal solution can be extracted.
The algorithm was invented in 1958 by Richard Bellman and Lester Ford. Due to its applicability to graphs with negative weights, it is more versatile than Dijkstra's algorithm, but this versatility comes at a cost of increased run times.

1.1 ALGORITHM

Identical to the Dijkstra algorithm, the Bellman-Ford method starts by initializing the distance to the source node to zero and the distance to all other nodes to infinity. In a loop over all edges, we check whether the distance to the end-node of the edge is greater than the distance to the start-node plus the edge value. Mathematically, we have that if

$$d[\texttt{start}] + edge[\texttt{start}-\texttt{to}-\texttt{end}] < d[\texttt{end}] \tag{1.1}$$

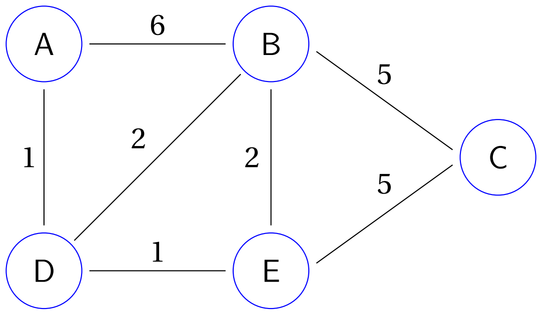

Figure 1.1: graph

with $d[\cdot]$ denoting the stored current distance to a node, we update the distance to the end-node by the lower value $d[\texttt{start}] + edge[\texttt{start} - \texttt{to} - \texttt{end}]$. We sweep through all edges of the graph. This constitutes one iteration of the Bellman-Ford algorithm. Since the longest possible path through the graph can have at most $|V| - 1$ segments, we have to perform $|V| - 1$ iterations to ensure convergence towards the shortest path from a starting to an ending node in the graph. As a consequence, the Bellman–Ford algorithm runs in $\mathcal{O}(|V| \cdot |E|)$ time, where $|V|$ and $|E|$ are the number of vertices and edges, respectively. This run time is longer than the one for Dijkstra's algorithm.

	A	B	C	D	E	A	B	C	D	E
$d[\cdot]$	0	∞	∞	∞	∞	0	6	∞	1	∞
DE = 1	0	∞	∞	∞	∞	0	6_A	∞	1_A	2_D
DB = 2	0	∞	∞	∞	∞	0	3_D	∞	1_A	2_D
BE = 2	0	∞	∞	∞	∞	0	3_D	∞	1_A	2_D
BC = 5	0	∞	∞	∞	∞	0	3_D	8_B	1_A	2_D
AD = 1	0	∞	∞	1_A	∞	0	3_D	8_B	1_A	2_D
EC = 5	0	∞	∞	1_A	∞	0	3_D	7_E	1_A	2_D
AB = 6	0	6_A	∞	1_A	∞	0	3_D	7_E	1_A	2_D
	1st iteration					2nd iteration				

Figure 1.2: Bellman-Ford tableau.

2 Implementation

The python code implementing the Bellman-Ford algorithm is given below.

```python
import numpy as np
import scipy as sp

def BellmanFord(ist,isp,wei):
    #----------------------------------
    #   ist:    index of starting node
    #   isp:    index of stopping node
    #   wei:    adjacency matrix (V x V)
    #
    #   shpath: shortest path
    #----------------------------------

    V = wei.shape[1]

    # step 1: initialization
    Inf     = 1e300
    d       = np.ones((V),float)*Inf
    p       = np.zeros((V),int)
    d[ist] = 0

    # step 2: iterative relaxation
    for i in range(0,V-1):
        for u in range(0,V):
            for v in range(0,V):
                w = wei[u,v]
                if (w != 0):
                    if (d[u]+w < d[v]):
                        d[v] = d[u] + w
                        p[v] = u

    # step 3: check for negative-weight cycles
    for u in range(0,V):
        for v in range(0,V):
            w = wei[u,v]
            if (w != 0):
                if (d[u]+w < d[v]):
                    print('graph has a negative-weight cycle')

    # step 4: determine the shortest path
    shpath = [isp]
    while p[isp] != ist:
        shpath.append(p[isp])
        isp = p[isp]
    shpath.append(ist)

    return shpath[::-1]
```

The associated main code is listed below.

```
if __name__ == '__main__':

    # indices of starting and stopping vertices
    ist = 4
    isp = 3

    # randomly generated adjacency matrix
    #N   = 10
    #ma  = np.around(np.random.uniform(0,1.2,(N,N)))
    #wei = ma*np.random.uniform(0,30,(N,N))
    #wei = np.tril(wei,-1) + np.triu(wei,1)

    # adjacency matrix
    wei = np.array([[ 0, 20,  0, 80, 0,  0, 90,  0],
                    [ 0,  0,  0,  0, 0, 10,  0,  0],
                    [ 0,  0,  0, 10, 0, 50,  0, 20],
                    [ 0,  0, 10,  0, 0,  0, 20,  0],
                    [ 0, 50,  0,  0, 0,  0, 30,  0],
                    [ 0,  0, 10, 40, 0,  0,  0,  0],
                    [20,  0,  0,  0, 0,  0,  0,  0],
                    [ 0,  0,  0,  0, 0,  0,  0,  0]])

    shpath = BellmanFord(ist,isp,wei)
    print ist,' -> ',isp,' is ',shpath
```