Notes on the Bellman-Ford-Moore Shortest Path Algorithm and its Implementation in MATLAB

Derek O'Connor
University College, Dublin

September 16, 2012*

The Shortest Path Problem is one of the most fundamental and important in combinatorial optimization. Many practical optimization problems can be cast as shortest path problems. Also, many other algorithms call a shortest path algorithm as a procedure.

Given a directed graph $G = (N, A)$, whose arcs $(u, v) \in A$ have lengths d_{uv}, and a node $r \in N$, find the shortest paths from r to all other nodes in N. Formally, this is called the *Single Source Shortest Path Problem* (SSSP Problem).

History. This problem has been studied continually since the 1950s. It appears in many forms and variations and hundreds of algorithms have been devised to solve it. The problem is easy in the sense that a shortest path in a graph of n nodes and m arcs can be found in $O(mn)$ time.

Despite the simplicity of the problem and the amount of research effort spent on it, many people (some of the best) are still working on it. Table 1 shows the history of *theoretical* improvements that have been made to the original $O(mn)$ algorithm of Bellman[?Bell57].

Two extensions of the standard problem (single source) are common:

- *Single Pair*: Given a source r and a destination t, find the shortest path from r to t.
- *All Pairs*: For every pair of nodes u and v in G find the shortest path from u to v.

All known algorithms for solving the single pair problem must solve all or part of the single source problem. The all-pairs problem can be viewed as n single source problems and solved accordingly. For these reasons the fundamental problem is the single source problem, i.e. find the shortest paths from r to all other nodes.

*Originally written in 1989 as a dissertation proposal for M.Mangt.Sc. students, MIS Dept., University College, Dublin. www.derekroconnor.net

Table 1. THE PROGRESS OF SHORTEST-PATH ALGORITHMS.

Date	Algorithm	Complexity
1957	Bellman, *et al.*	$O(mn)$
1958	Dijkstra	$O(n^2)$
1964	Williams	$O(m \log_2 n)$
1977	Johnson	$O(m \log_d n), \quad d = \max(2, \frac{m}{n})$
1977	Boas, *et al.* (1977)	$O(c + m \log_2 c \log_2 \log_2 c)$
1982	Johnson	$O(m \log_2 \log_2 c)$
1984	Fredman & Tarjan	$O(m + n \log_2 n)$
1985	Gabow	$O(m \log_d c), \quad d = \max(2, \frac{m}{n})$
1989	Tarjan *et al.*	$O(m + n\sqrt{\log_2 c})$

1 Theory

The solution to the shortest path problem is a *spanning tree* $T = (N, A_T)$ of G, which is rooted at r, with $A_T \subseteq A$ and $|A_T| = n - 1$. The unique path from any node $u \in T$ to r, is the shortest path from u to r. The length of this path is the shortest distance between u and r.

The spanning tree may be succinctly represented by a pair of n-vectors (p, D), where $p(u)$ is the parent of u and $D(u)$ is the distance from u to the root r, i.e., the length of the path $u \to p(u) \to p(p(u)) \to \ldots \to r$.[1]

1.1 Spanning Tree Solutions.

The optimum solution to the single-source shortest path problem is a set of $n - 1$ shortest paths and their lengths, which may be viewed as a tree. We call such a tree a *Shortest Path Spanning Tree*. Figure 1 shows an example of a network and its shortest path spanning tree.

Theorem 1. *$T(r) = (p, D)$ is a shortest-path spanning tree of G if and only if for every arc $(u, v) \in A$*

$$D_v \leq D_u + d_{uv}, \tag{1.1}$$

where D_u and D_v are the lengths of the tree paths from r to u and v, respectively □

This theorem allows us to test in $O(m)$ time if a given spanning tree is a shortest path spanning tree.

1.2 Bellman's Equations.

Richard Bellman, in his 1957 paper, *On a Routing Problem* [?Bell57], formally stated and analyzed the existence, uniqueness, and construction of a solution to the shortest path problem.

[1] Where appropriate, we use the MATLAB vector notation throughout these notes.

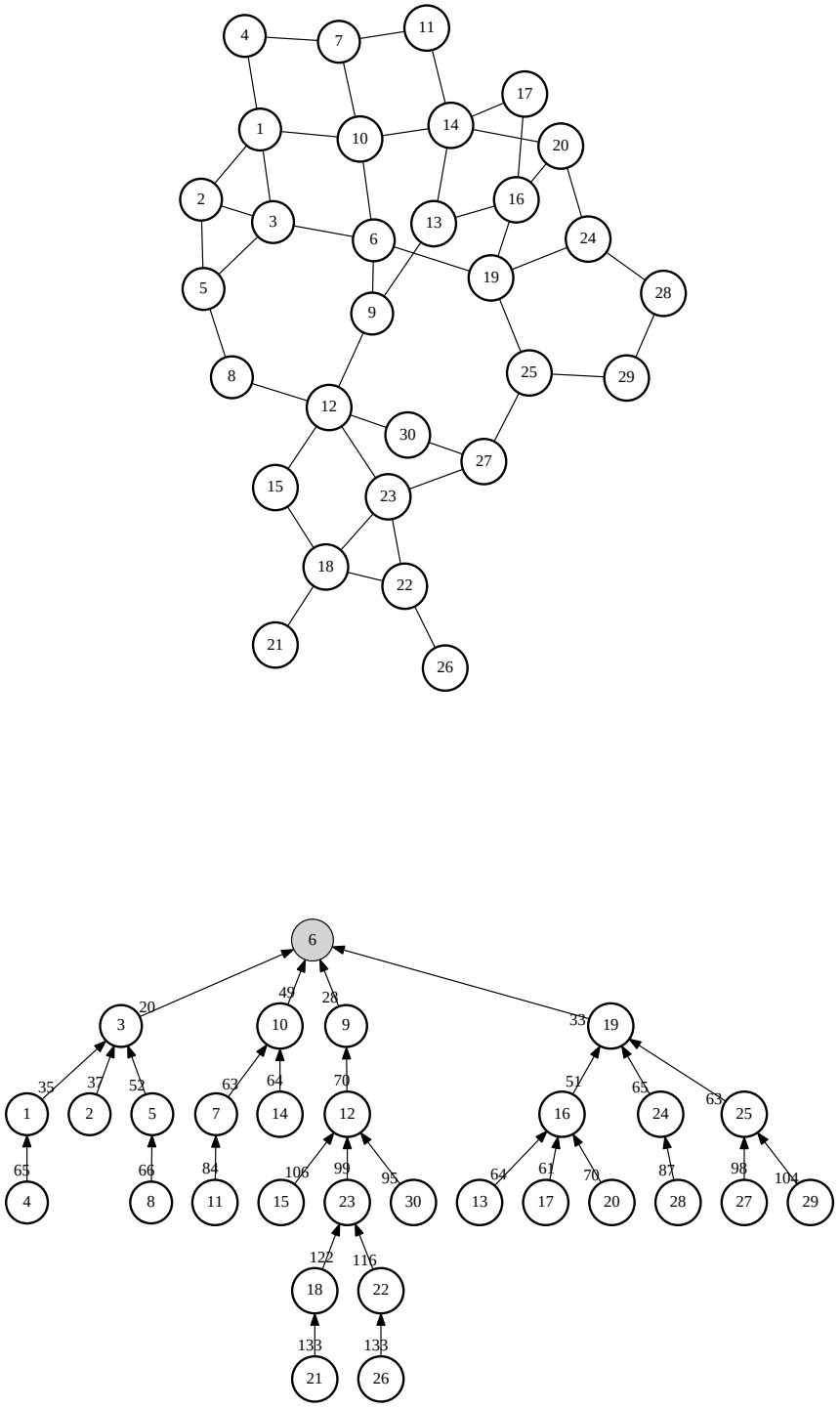

Figure 1. A 30-NODE NETWORK AND ITS SHORTEST PATH SPANNING TREE

Bellman applied his *Principle of Optimality* to the length of the shortest path to any node v. This path must have a final arc (u, v), for some node u. Thus the shortest path from r to v is a path from r to u followed by the arc from u to v, as shown in Figure 2. Bellman's principle says that the path from r to u must also be a shortest path from r to u. This means that the shortest path distances must satisfy the following equations :

$$\begin{aligned} D_r &= 0, \\ D_v &= \min_{u \neq v} \{D_u + d_{uv}\}, \quad v \in N - \{r\}. \end{aligned} \tag{1.2}$$

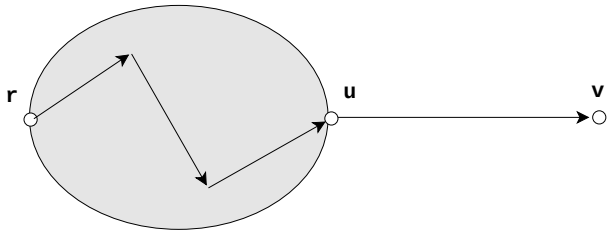

Figure 2. BELLMAN'S PRINCIPLE OF OPTIMALITY FOR A SHORTEST PATH

Each D_v is a non-linear function of v, and Bellman proposed solving these non-linear *functional* equations[2] by the method of *successive approximations*[3]. First choose a set of initial values $D_v^{(0)}$, $v = 1, 2, \ldots, n$, and then iteratively solve these equations, for $k = 1, 2, \ldots$

$$\begin{aligned} D_r^{(k+1)} &= 0, \\ D_v^{(k+1)} &= \min_{u \neq v} \{D_u^{(k)} + d_{uv}\}, \quad v \in N - \{r\}. \end{aligned} \tag{1.3}$$

We will call this the *mathematical* form of Bellman's algorithm to distinguish it from the *algorithmic* form to be discussed later. The mathematical form is convenient for proving the existence and uniqueness of the an optimal solution to the problem.

Bellman showed the following, which we have called a theorem

Theorem 2. *If $D_u^{(0)} = d_{ur}$, $i = 1, 2, \ldots, n$, then the approximation scheme of (1.3) generates a monotone sequence $D_u^{(0)} \geq D_u^{(1)} \geq D_u^{(2)} \geq \cdots$, which converges after $n - 1$ steps with $D_u^{(n-1)} = D_u$, the length of the shortest path from u to r.* □

1.3 An Arc-Scanning Relaxation Algorithm.

The shortest path problem can be solved by the following very general relaxation algorithm. We may view this algorithm as (i) finding a tree $T(r)$ that satisfies Theorem 1.1, or (ii) solving Bellman's equations, or (iii) the *dual Simplex method* applied to the linear program. Most shortest path algorithms are based on the this algorithm.

[2]These two functional equations $f(x+y) = f(x) + f(y)$, and $f(x+y) = f(x)f(y)$, in the unknown function f, have the solutions $f(z) = cz$, and $f(z) = e^{-cz}$, respectively.

[3]Bellman called this *Approximation in Policy Space*, which is a dynamic programming term.

Algorithm SPRelaxBF (G, r) **return** (p, D)

$(p, D) \leftarrow$ Initialize(G, r)
for $i \leftarrow 1$ **to** $n-1$ **do**
  for each arc $(u \to v) \in G$ **do**
    $(p, D) \leftarrow$ Relax(u, v, p, D)
  endfor
endfor i
endalg SPRelaxBF

Algorithm Relax (u, v, p, D) **return** (p, D)

if $D[v] > D[u] + d_{uv}$ then SP Tree p is not Optimal
  $D[v] \leftarrow D[u] + d_{uv}$ Update (p, D)
  $p[v] \leftarrow u$
endif
endalg Relax

Algorithm Initialize (G, r) **return** (p, D)

Set $D[i] \leftarrow \infty$, $i \in N - \{r\}$ and $D[r] \leftarrow 0$
Set $p[i] \leftarrow 0$, $i = 1, 2, \ldots, n$
endalg Initialize

We have implemented this rather vague algorithm as **Algorithm** Bellman-Ford-Moore, the algorithmic form of Bellman's successive approximation scheme (See 1.3), which is shown below.[4]

In this algorithm the spanning tree is represented by two n-vectors p and D, where $p[u]$ is the parent of u and $D[u]$ is the distance from u to r in the tree.

Bellman's algorithm applies Theorem 1.1 by scanning the entire set of arcs A for each iteration. Also, it always performs $n-1$ iterations without checking to see if the current tree is optimal. This can be wasteful because many arcs will not violate Theorem 1.1. Worse, if the tree becomes optimal at iteration $k < n-1$, then all the iterations after k are pointless. This deficiency can be remedied very easily by putting in a tree-optimality check in the outer iteration loop, as shown below.

[4] Non-programmers and mathematicians can see the correspondence between this algorithm and the successive approximation scheme if they realize that the algorithmic statement, $D[v] := D[u] + d_{uv}$, means that the logical assertion, $D^{(k+1)}(v) = D^{(k)}(u) + d_{uv}$, is true, after this statement has been executed.

Algorithm SPRelaxBF-OT (G, r) **return** (p, D)

```
(p, D) ← Initialize(G, r)
Opt ← false
while ¬ Opt do
    Opt ← true
    for each each arc (u → v) ∈ G do
        (p, D, Opt) ← Relax(u, v, p, D)
    endfor
endwhile ¬ Opt
endalg SPRelaxBF
```

Algorithm Relax (u, v, p, D) **return** (p, D, Opt)

```
if D[v] > D[u] + d_uv  then SP Tree (p, D) is not Optimal
    Opt ← false
    D[v] ← D[u] + d_uv  Update (p, D)
    p[v] ← u
endif
endalg Relax
```

This is an important algorithm, and the attention of young, *shortest-path algorithmists* must be drawn to the following points: (1) the outer loop is fixed at $n-1$, by virtue of Theorem 2; (2) the inner loop is over the *arcs* of G, not the *nodes*. This means that its complexity is $O(mn)$, which becomes $O(n^2)$ if G is *sparse*, and $O(n^3)$ if G is *dense*. Bellman's paper and algorithm of 1957 established this $O(mn)$ upper bound on the complexity of the shortest path problem. Dijkstra came later, in 1959, with his $O(n^2)$ algorithm, which lowered the upper bound.[5]

Finally, we must draw the attention to another important fact about Bellman's algorithm: it is *robust*, i.e., it works, even if G has negative arc weights. Dijkstra's algorithm is not robust because it cannot handle negative arc-weights.

2 The MATLAB Data Structures and Functions

2.1 Sparse Matrices

Most physical and social structures are sparse in the sense that the elements of these structures are loosely connected. For example, the atoms of very large molecules are directly connected only to a few other atoms; towns are directly connected to 2 or 3 other towns; a person in an organization of 10,000 probably communicates with less than 10-20 people in any week.

A mathematical model of these connections is the *Adjacency Matrix* A, where $a_{ij} = 1$ if element i is connected directly to element j, and $a_{ij} = 0$ or ∞ otherwise. A more general model is : $a_{ij} \neq 0$

[5] Dijkstra also came later with his version of Prim's (1957) Minimum Spanning Tree Algorithm.

if element i is connected directly to element j, and $a_{ij} = 0$ or ∞ otherwise. This allows us to represent a road network by an adjacency matrix, where a_{ij} is the distance or travel time between towns i and j directly connected by a road, and $a_{ij} = \infty$ otherwise. Such adjacency matrices occur in many applications and they all have the characteristic that most of their elements are 0 (or ∞).

Despite this, many university courses still present graph algorithms that use full (dense) matrices. This may be mathematically convenient, but it misleads students who, when they have gone out into the real world, are flummoxed when presented with a real graph problem:

> I'm a LAVA programmer at *Gargle* and I'm trying to find the shortest paths in a social network that has 5 million nodes. I've written a really slick 2-line LAVA version of Dijkstra's algorithm but LAVA keeps giving me an out-of-memory error. Please help.

2.2 Sparse Graph Data Structures

An important point to make at the outset is that all graphs are assumed to be and are represented as *directed* graphs. An undirected arc (u, v, d_{uv}) can be represented by two directed arcs (u, v, d_{uv}) and (v, u, d_{uv}).

The simplest and most efficient way to *store* a directed graph is as a list of m triples $G = \{(u, v, w_{uv}), u, v = 1, 2 \ldots, m\}$. This can be implemented in various languages in various ways, but the simplest way for all languages is 3 separate arrays: $u[1 : m], v[1 : m]$, and $w[1 : m]$, with the assumption that the direction of the arc is $u \to v$. These arrays are often called `(tail, head, weight)`, to indicate the direction of the arc.

In general, the arc-list-of-triples is not a very useful *internal* representation of G because it makes standard traversals and graph modification operations very cumbersome and inefficient. Try writing a *Lowest Common Ancestor* algorithm using this representation, or *Depth First Search*.

Fortuitously, the arc-list-of-triples is a perfect match for the original Bellman-Ford-Moore Algorithm: it repeatedly scans a list of arcs, and with the 3-array representation we can access each arc in $O(1)$ time, i.e., random access time. Hence, testing for optimality can be done in $O(m)$ time.

The only other data structure we need is the spanning tree $T(r)$ of G which, after convergence to optimality, will be the shortest path spanning tree of G, rooted at r. Again, there are many ways to represent a tree but, again, fortuitously, the simple array of parent "pointers" $p[1 : n]$ is perfect, along with an array $D[1 : n]$, where $D[u]$ is the distance from u to the root r, in the current tree. The path from u to the root r is $u, p[u], p[p[u]], \ldots, r$, whose length is $D[u]$.

We will avoid the arcana of type sizes by assuming each number is stored in a box of fixed size. Hence the total storage for our problem and solution is $3m + 2n$ boxes. Without fancy and expensive packing and unpacking, you cannot get lower.

In the early 1990s MATLAB added excellent sparse matrix operations that used their sparse matrix data structure. I have never become an expert in the use of MATLAB's sparse matrices so I stuck to the arc-list-of-triples representation and crudely transformed a MATLAB sparse adjacency matrix into an arc-list-of-triples. This is done in the `[m,n,p,D,tail,head,W] = Initialize(G)` function.

2.3 The Original Bellman-Ford-Moore Algorithm

Here is the MATLAB implementation of the original BFM algorithm with the added optimality test.

```
1
2   % ------------------------------------------------------------------
3       function [p,D,iters] = BFMSpathOT(G,r)
4   % ------------------------------------------------------------------
5   %   G is a sparse nxn adjacency matrix. r is the root of SP Tree
6   %   Derek O'Connor, 19 Jan 2012.
7
8     [m,n,p,D,tail,head,W] = Initialize(G);
9     p(0) = 0; D(r) = 0;
10    iters = 0;
11    opt = false;
12    while ~opt                          % Converges in ≤ n-1 iters if
13        iters = iters+1;                % no negative cycles exist in G.
14        opt = true;
15        for arc = 1:m                   % O(m) optimality test.
16            u = tail(arc);
17            v = head(arc);
18            duv = W(arc);
19            if D(v) > D(u) + duv;       % Sp Tree not optimal
20               D(v) = D(u) + duv;       % Update (p,D)
21               p(v) = u;
22               opt = false;
23            end
24        end % for arc
25    end % while ~opt
26  % ---------------- End BFMSpathOT ---------------------------------
27  %
```

```
1
2   % ------------------------------------------------------------------
3       function [m,n,p,D,tail,head,W] = Initialize(G)
4   % ------------------------------------------------------------------
5   % Transforms the sparse matrix G into the list-of-arcs form
6   % and intializes the shortest path parent-pointer and distance
7   % arrays, p and D. Derek O'Connor, 21 Jan 2012
8
9     [tail,head,W] = find(G); % Get arc list {u,v,duv, 1:m} from G.
10    [~,n] = size(G);
11    m = nnz(G);
12    p = zeros(n,1);           % Shortest path tree of parent pointers
13    D = Inf*ones(n,1);
14
15  % ---------------- End Initialize ---------------------------------
16  %
```

The optimality test has been added to the original form. We will see below that this simple addition can speed up this function dramatically for certain network categories.

3 Testing the Algorithm

Shortest path algorithms are considered so important that DIMACS[6] held a competition to see who could write the fastest algorithm. This was called *9th DIMACS Implementation Challenge - Shortest Paths*: October 2005 – December 2006 and was held in Rome.[7] Lots of information and data is available at the website below.

3.1 Test Data

Proper testing requires that the algorithms be tested on as wide a range of problem type as possible. Randomly generated problems can only test general features of the algorithm. Ultimately any algorithm must be tested on problems it is intended to solve, not on synthetic problems.

The following problem parameters should be considered when testing Shortest Path algorithms.

1. Large or small arc lengths
2. Real or Integer arc lengths
3. Negative arc lengths
4. Sparse or dense graph, i.e., $m = O(n)$ or $O(n^2)$.
5. Special structure, e.g., random, lattice, Euclidean.

3.2 Sparse Random Graphs

Random problems are useful in testing code to see if it works and to get a general idea of its running times for various sizes of problems. Other than that, they are useless because, by definition, *random* problems are *atypical* problems.

A sparse random graph with n nodes and represented by a sparse $n \times n$ adjacency matrix is particularly easy to construct in MATLAB. The general MATLAB statement that constructs a sparse $m \times n$ adjacency matrix with a given density is `R = sprand(m,n,density)`. We want to generate a sparse $n \times n$ adjacency matrix with `nnzr` non-zeros per row. We do this as follows:

```
G = sprand(n,n,nnzr/n);
```

The density of a matrix is defined as $nnz/(m \times n) = nnz/n^2$ for a square matrix. By definition, a sparse $n \times n$ matrix has $O(n)$ non-zeros, or $O(1)$ non-zeros per row. Notice in the MATLAB statement above that with fixed `nnzr`, the density goes down as n increases. This is typical of real road networks.

[6] The Center for Discrete Mathematics and Theoretical Computer Science, Rutgers.
[7] http://www.dis.uniroma1.it/challenge9/

Table 2. BFMSpath0T ON SPARSE RANDOM NETWORKS.

n nodes $\div 10^5$	m arcs $\div 10^5$	iters to opt	Time secs
1.00	6.00	16	1.29
1.47	16.15	17	3.72
2.15	6.46	20	1.95
3.16	15.81	18	4.52
4.64	41.77	19	14.17
6.81	47.69	19	18.17
10.00	50.00	20	21.74
14.68	102.75	20	46.75
21.54	129.27	20	61.77
31.62	126.49	22	68.00

Time in secs on LENOVO THINKPAD X220,
Intel CORE i7-2640M CPU @ 2.80GHz, 8GB RAM..

An interesting result for random networks is that the number of iterations to optimality is fairly constant, $16 - 22$, over a wide range of $m = 500,000 - 12,670,000$. As we will see, this is not the case for real road networks.[8]

3.3 Real Road Networks

The algorithms were tested on sets of real networks of the USA, all of which had non-negative integer arc lengths. These were downloaded from the website below.

We can see from Table 3 and the graph in Figure 3 below, and can confirm by MATLAB's `polyfit`, that this code runs in essentially $O(m)$ time, i.e., linear in the number of arcs. Theoretically this means that this shortest path algorithm is optimal, at least for the problems we have solved, because just to read a graph problem requires $O(m)$ time. We will see below that this algorithm is far from optimal in the practical sense.

3.4 Detailed analysis of BFMSpath0T

The theoretical upper bound on the running time of this algorithm is $O(mn)$. This is the worst possible that can happen and is fine for theoretical work, but can be very misleading in practical applications, as we saw in the previous paragraph. Doing a detailed theoretical analysis of any algorithm can be very difficult and in many cases pointless because the resulting formulas would be so complicated they would give no real insight.

[8] *Beware of Random Networks!*

Code Profiling.

An alternative is to meter the code by counting the number of times each statement is executed. Even one run of the code on one problem can often reveal much about the behaviour of the algorithm.

A profile of `BFMSpath0T` running on a sparse random graph is shown in Table 4. It is pleasing to see that there are no obvious *hot-spots* in this code, i.e., one or two statements that consume most of the running time. What is not so pleasing to see are the *cold-spots*: the three tree-update statements are executed 5314469 out of a total of 99999720, or just 5% of the time. This means that this algorithm (code) is wasting a lot of time scanning arcs that will not be updated because the algorithm blindly checks every arc in A at each iteration.

4 Conclusion

We have seen that the original Bellman-Ford-Moore Shortest Path Algorithm, with an optimality test, is simple to implement in MATLAB and runs quite fast on random and real problems. We have also seen that it is far from optimal because it wastes a lot of time scanning arcs that will not be updated.

The only way to speed up this algorithm is to find clever ways of choosing arcs that violate Theorem 1.1, i.e., arcs (u, v) where $D_v > D_u + d_{uv}$. Much time and effort has been spent on this problem over the past 50 years and continues today. Node-scanning has been used along with more sophisticated node-arc selection data structures which have yielded great improvements.

Research on Shortest Path Algorithms is alive and well.

Table 3. `BFMSpathOT` ON REAL ROAD NETWORKS, USA.

No.	Name	m arcs	n nodes	iters	Time
1	spusaNY	733,846	264,346	632	45
2	spusaBAY	800,172	321,270	668	53
3	spusaCOL	1,057,066	435,666	908	92
4	spusaFLA	2,712,798	1,070,376	1431	118
5	spusaNW	2,840,208	1,207,945	1976	167
6	spusaNE	3,897,636	1,524,453	1316	149
7	spusaCAL	4,657,742	1,890,815	2383	330
8	spusaLKS	6,885,658	2,758,119	2865	586
9	spusaE	8,778,114	3,598,623	2633	735
10	spusaW	15,248,146	6,262,104	2600	1241
11	spusaCTR	34,292,496	14,081,816	–	–
12	spusaFULL	58,333,344	23,947,347	–	–

Time in secs on LENOVO THINKPAD X220, Intel CORE i7-2640M CPU @ 2.80GHz, 8GB RAM..

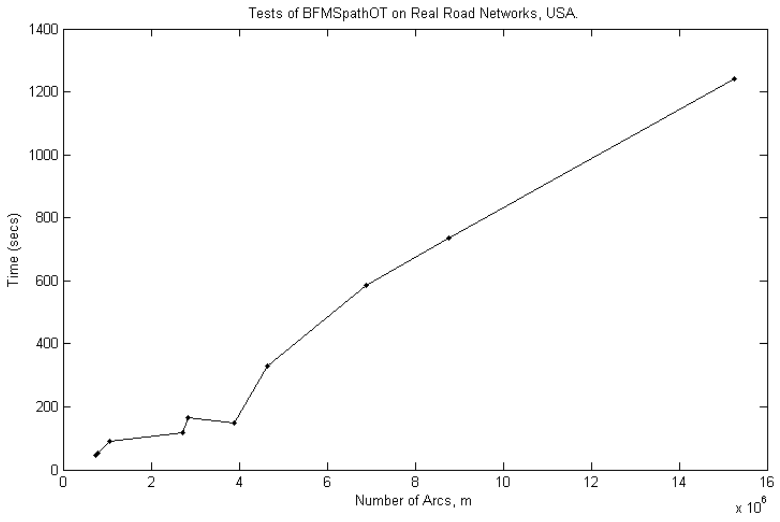

Figure 3. TESTS OF `BFMSpathOT` ON 10 ROAD NETWORKS

Table 4. PROFILING BFMSpath0T ON A SPARSE RANDOM NETWORK.

```
 1 ──────────────────────────────────────────────────────────────────────
 2  n=10^6; G=sprand(n,n,5/n); r=1;
 3  [p,D,iter] = BFMSpathOT(G,r);  nnz = 4,999,986
 4 ──────────────────────────────────────────────────────────────────────
 5    Time    Count    Statement
 6 ──────────────────────────────────────────────────────────────────────
 7    0.24        1   [m,n,p,D,tail,head,W] = Initialize(G);
 8  < 0.01        1   p(r)=0; D(r)=0;
 9  < 0.01        1   for iter = 1:n-1
10  < 0.01       20       optimal = true;
11  < 0.01       20       for arc = 1:m      % Start of O(m) optimality test.
12    4.25 99999720           u = tail(arc);
13    4.50 99999720           v = head(arc);
14    4.49 99999720           duv = W(arc);
15    8.86 99999720           if D(v) > D(u) + duv;
16    0.33  5314468              D(v) = D(u) + duv;
17    0.23  5314468              p(v) = u;
18    0.19  5314468              optimal = false;
19    0.20  5314468           end
20   14.00 99999720       end % for arc     % End of optimality test.
21  < 0.01       20       if optimal
22  < 0.01        1           return
23                         end
24  < 0.01       19   end % for iter
```