An $O(nm)$ time algorithm for finding the min length directed cycle in a graph*

James B. Orlin† Antonio Sedeño-Noda‡

October 30, 2016

Abstract

In this paper, we introduce an $O(nm)$ time algorithm to determine the minimum length directed cycle (also called the "minimum weight directed cycle") in a directed network with n nodes and m arcs and with no negative length directed cycles. This result improves upon the previous best time bound of $O(nm + n^2 \log\log n)$. Our algorithm first determines the cycle with minimum mean length λ^* in $O(nm)$ time. Subsequently, it chooses node potentials so that all reduced costs are λ^* or greater. It then solves the all pairs shortest path problem, but restricts attention to paths of length at most $n\lambda^*$. We speed up the shortest path calculations to $O(m)$ per source node, leading to an $O(nm)$ running time in total. We also carry out computational experiments comparing the performance of the proposed methods and other state-of-the-art methods. Experiments confirmed that it is advantageous to solve the minimum mean cycle problem prior to solving shortest path problems. Analysis of our experiments suggest that the running time to solve the minimum length directed cycle problem was much faster than $O(n^2)$ on average.

1 Introduction.

We address the determination of the Minimum Length Directed Cycle (MLDC) in a graph $G = (V, A)$ with n nodes and m arcs and with no negative length directed cycles. (Elsewhere, researchers have referred to the MLDC as the minimum weight directed cycle or the minimum cost directed cycle.) Floyd [12] and Warshall [32] showed how to solve this problem in $O(n^3)$ time. An alternative approach is to find shortest paths between all pairs of nodes. In case there are negative length arcs, the first shortest path problem is solved using the label correcting algorithm. Subsequently, one can use reduced costs to transform the problem into an equivalent problem with nonnegative arc lengths. The subsequent $n - 1$ shortest path problems are solved using Dijkstra's Algorithm. Using the shortest path algorithm of Fredman and Tarjan [14], the running time is $O(nm + n^2 \log n)$. Hagerup [16] provides an improved algorithm for the all pairs shortest path problem for sparse graphs running in $O(nm + n^2 \log\log n)$ time. Hagerup's algorithm permitted $O(1)$ time operations on words of size w, where w is the number of bits to express the largest integer in the problem instance. Pettie [26] achieved the same time bound with real valued lengths, assuming that operations on real numbers take $O(1)$ time.

Based on the results of Itai and Rodeh [18], Roditty and Williams [28] introduce an $\tilde{O}(Mn^\omega)$ time algorithm using fast matrix multiplication to solve MLCD in directed graphs with integral weights in the interval $[-M, M]$. Using the results of [6] and [22], matrix multiplication runs in n^ω time, where ω is slightly less than 2.373. The theoretically fast algorithms for matrix multiplication are not effective in practice due to a very large constant factor term.

Roddity and Williams [28] also prove the following very interesting result: they reduce the problem of finding the minimum weight cycle to the problem of finding a minimum weight triangle in a $\theta(n)$-node undirected graph with weights in $\{1, ..., O(M)\}$.

We present an $O(nm)$ time algorithm for solving the MLDC problem. Our algorithm improves upon other combinatorial algorithms and is the best strongly polynomial time bound. Our algorithm's running time matches the time bound for detecting a negative length cycle in a graph using Bellman [4], Ford [13] and Moore [23] algorithms. Our method first computes the Minimum Mean Cycle (MMC) using the $O(nm)$ algorithm of Karp [19]. After using node potentials to modify the lengths, it then carries out n single-source shortest path computations as restricted to paths of length at most $n\lambda^*$, where λ^* is the minimum mean length of a cycle. Each of these n additional single-source shortest path computations requires only $O(m)$ time using a variant of Dial's [10] implementation of Dijkstra's algorithm. Therefore, we obtain an $O(nm)$ time algorithm for the MLDC.

There is a heuristic argument that it would be difficult to solve the MLDC faster than $O(nm)$. Suppose for example that one wanted to establish that a cycle W

*Supported by ONR Grant N000141410073

†Massachusetts Institute of Technology

‡Universidad de La Laguna

was optimal for the MLDC. As part of the proof of the optimality of W, one would need to establish that for each node $j \notin W$, the minimum length cycle containing j has length at least $c(W)$. This appears to require $\Omega(n)$ shortest path computations in general with $\Omega(m)$ steps per shortest path computation. The running time would be $\Omega(nm)$ in total.

The situation for the MLDC contrasts with the MMC. The currently best strongly polynomial time algorithm for the MMC runs in $O(nm)$ time; however, there is a quick demonstration that W is optimal for the MMC problem. Let λ^* denote the mean length of W and let p be the node potentials as described in Lemma 1 of the next section. Then the reduced cost of every arc of W equals λ^*, and the reduced cost of every other arc of G is at least λ^*. This establishes in $O(m)$ time that W is optimal. It also provides some hope that we will eventually develop an algorithm for MMC that is faster than $O(nm)$ time.

The literature on graph algorithms contains numerous studies on the practical behavior of the shortest paths algorithms, including MMC algorithms. Studies on the MMC algorithms have been developed by Dasdan et al. [8], Dasdan [7] and Georgiadis et al. [15]. In this paper, we include an experimental study on the MLDC algorithms. To the best of our knowledge, ours is the first experimental study on MLDC algorithms. Our experimental study showed that the average case performance of our algorithms is far better than the $O(nm)$ worst case analysis would suggest.

Rather than rely on Karp's $O(nm)$ algorithm, we relied on Howard's [17] Algorithm for finding the MMC. Howard's Algorithm is far more efficient in practice. For most of our experiments, solving the MMC first did speed up the overall computation time. Nevertheless, the time to solve the MMC was typically the bottleneck operation. Its running time was greater than the running time to solve all of the shortest path problems.

Several combinatorial optimization algorithms solve the MLDC problem as a subroutine. An example is the problem of determining the K best assignments. The fastest algorithm for this problem runs in $O(Knm)$ time. It uses the proposed MLDC algorithm as a subroutine while relying also on the results in the papers of Chegireddy and Hamacher [5] and Sedeño-Noda and Espino-Martin [29]. Another application which employs the MLDC as a subroutine is the enumeration of the extreme points of some classical combinatorial optimization problems (vertices of the corresponding polyhedron) as ordered by the objective function (Balinski, [3] and Provan, [27]).

The organization of the paper is as follows. Section 2 presents the definitions and initial results needed in the paper. In Section 3, we show how the n shortest path tree computations can be determined in $O(nm)$ time. This section contains pseudo-code of the proposed algorithm, illustrated on a small example. In Section 4, we present the computational experiments and results comparing the performance of the proposed algorithms with other state-of-the-art algorithms. Section 5 contains conclusions.

2 The minimum length directed cycle problem: initial results.

Given a directed network $G = (V, A)$, let $V = \{1, ..., n\}$ be the set of n nodes and let A be the set of m arcs. For each node $i \in V$, we let $\Gamma_i^+ = \{j \in V | (i, j) \in A\}$ and $\Gamma_i^- = \{j \in V | (j, i) \in A\}$ denote the successors and predecessors adjacent lists of node i. For each arc $(i, j) \in A$, let $c_{ij} \in \mathbb{R}$ be its length. We sometimes refer to c_{ij} as the *cost* of arc (i, j), and will use the terms length and cost interchangeably. A *directed path* P_{st} from node s to node t is a sequence $\langle i_1, (i_1, i_2), i_2, ..., i_{l-1}, (i_{l-1}, i_l), i_l \rangle$ of distinct nodes and arcs satisfying $i_1 = s$, $i_l = t$ and for all $1 \leq w \leq l - 1, (i_w, i_{w+1}) \in A$. The *length* of a directed path is the sum of the lengths of the arcs in the path. A *directed cycle* is a directed path from node i_1 to node i_r together with the arc (i_r, i_1) . We let $c(W) = \sum_{(i,j) \in W} c_{ij}$ denote the length of the directed cycle W. The MLDC problem consists in identifying a directed cycle with minimum length. The MLDC problem is said be *feasible* if G contains at least one directed cycle and G does not contain a negative length directed cycle.

For each cycle W, we let

$$\lambda(W) = \frac{c(W)}{|W|}$$

denote the *mean length* of the cycle W. The minimum mean cycle (MMC) problem consists in determining a directed cycle with minimum mean length. Henceforth, we will let W^* be an optimal MMC and we will let $\lambda^* = \lambda(W^*)$.

A *directed graph* G is *strongly connected* if every two nodes are reachable by a path from each other. A *strongly connected component* of a directed graph G is a maximal strongly connected subgraph of G.

The following lemma is well known, and stated without proof. See for example, Ahuja et al. [1].

LEMMA 2.1. *Let $G = (V, A)$. For each $i \in V$, let $p(i)$ denote the "node potential" of i. For each arc $(i, j) \in A$, let $c_{ij}^p = c_{ij} + p(i) - p(j)$ denote its reduced cost. Then for any cycle W of G, $c(W) = c^p(W)$.*

Lemma 2.2 is well known. We provide its proof for the sake of completeness.

LEMMA 2.2. *Let W^* be an optimum MMC for a strongly connected graph $G = (V, A)$, and let $\lambda^* = \lambda(W^*)$. Let G' be the graph obtained from G by replacing the cost c_{ij} of each arc $(i, j) \in A$ by $c'_{ij} = c_{ij} - \lambda^*$. Let $p(i)$ be the shortest path length from node 1 to node i in G'. For any arc $(i, j) \in A$, the reduced cost of (i, j) is $c^p_{ij} = c_{ij} + p(i) - p(j) = c'_{ij} + p(i) - p(j) + \lambda^*$. Then for any arc $(i, j) \in A$, $c^p_{ij} \geq \lambda^*$. Moreover, for any arc $(i, j) \in W^*$, $c^p_{ij} = \lambda^*$.*

Proof. Consider first the case that $\lambda^* = 0$. Suppose $(i, j) \in A$. Because $p(i)$ and $p(j)$ are the lengths of the shortest paths from node 1 to nodes i and j, $p(j) \leq p(i) + c_{ij}$, and thus $c^p_{ij} \geq 0$. If (i, j) is on the shortest path from node 1 to node j, then $p(j) = p(i) + c_{ij}$, and thus $c^p_{ij} = 0$. Accordingly if $\lambda^* = 0$, then the lemma is true.

If $\lambda^* > 0$, then $c'(W^*) = 0$, and W^* is an optimum MMC with respect to c'. The previous paragraph shows that $p(j) \leq p(i) + c'_{ij}$ for all arcs $(i, j) \in A$. Therefore, $c^p_{ij} = c_{ij} + p(i) - p(j) = c'_{ij} + p(i) - p(j) + \lambda^* \geq \lambda^*$.

For every arc $(i, j) \in W^*$, $c^p_{ij} \geq \lambda^*$. By Lemma 2.1, $c^p(W^*) = c(W^*) = \lambda^*$. Therefore, for each arc $(i, j) \in W^*$, $c^p_{ij} = \lambda^*$.

One can use Karp's [19] algorithm to determine λ^* and p in $O(nm)$ time. By Lemma 1, solving the MLDC problem with original costs c is equivalent to solving the MLDC problem with costs c^p in G, where all arc lengths are at least λ^*.

In order to find a minimum length cycle, for each pair i, j of nodes, we will compute $d^p(i, j)$, which is the shortest length of a path from node i to node j using arc lengths c^p. The shortest length cycle can be determined by finding $min\{d^p(i, j) + c^p_{ji} : (j, i) \in A\}$. Solving MLDC by solving an all pairs shortest path problem is a well known approach. We speed up this standard approach by preprocessing to compute p, and then computing $d^p(i, j)$ as restricted to pairs of nodes i and j such that $d^p(i, j) \leq c^p(W^*) = c(W^*) = \lambda^*|W^*|$.

3 Solving the n shortest path problems in $O(nm)$ time to determine the MLDC.

In order to simplify notation, we assume in this section that $c = c^p$ and $d = d^p$. We consider a graph in which each arc (i, j) has length $c_{ij} \geq \lambda^*$. We let s denote the origin node for Dijkstra's Algorithm, and find a shortest path from node s to each node j as restricted to nodes j such that distance $d(s, j) < (|W^*| - 1)\lambda^*$. There is no need to seek longer paths from node s. If k were a node such that $d(s, k) \geq (|W^*| - 1)\lambda^*$, then every cycle containing arc (k, s) would have length at least $d(s, k) + c_{ks}$, which is at least $c(W^*) = \lambda^*|W^*|$.

We use Dial's [10] version of Dijkstra's [11] Algorithm, using at most $|W^*| - 1$ buckets. We rely on a speed-up of Dial's Algorithm developed by Denardo and Fox [9]. Suppose that at some iteration, we are identifying shortest paths from node s. We describe this implementation next.

Let $d_s(j)$ be the temporary distance label of node j. Thus $d_s(j)$ is the length of some path from node s to node j in G, and $d_s(j) \geq d(s, j)$. Let $pred(j)$ denote the node that precedes j on the path from s to j of length $d_s(j)$.

Initially, $d_s(s) = 0$, and $d_s(j) = +\infty$ for $j \neq s$. The algorithm keeps an array of $|W^*| - 1$ buckets, where $Bucket(k)$ contains any node j such that $k\lambda^* \leq d_s(j) < (k + 1)\lambda^*$. Node s is not placed in any bucket since $d_s(s) = 0$. The steps for inserting node j into the correct bucket when finding shortest paths from node s is described in Procedure Bucket-Update(s, j) in Figure 1.

Procedure Bucket-Update(s, j)
let $k = \lfloor d_s(j)/\lambda^* \rfloor$;
if $((k \leq |W^*| - 1)$ **and** (node j is not in $Bucket(k)))$ **then**
  delete j from its current bucket;
  add j to $Bucket(k)$;

Figure 1. The subroutine for updating buckets.

Dial's Algorithm iteratively selects a node i. The first selected node is node s. Subsequently, the algorithm selects any node i from the minimum non-empty bucket, and deletes i from the bucket. It does not necessarily select the node i with minimum distance label. If all buckets are empty, the algorithm stops looking for shortest cycles containing node s.

After selecting node i, the algorithm checks to see if the cycle consisting of the path from s to i plus arc (i, s) is an improvement over the current shortest cycle. Subsequently, the algorithm scans arcs out of node i. For each scanned arc (i, j), $d_s(j)$ is replaced by $\min\{d_s(j), d_s(i) + c_{ij}\}$, and $pred(j)$ is updated when appropriate. If $d_s(j)$ is decreased, and if $d_s(j) < |W^*|\lambda^*$, then the algorithm checks to see if node j is in its correct bucket; if it isn't, then j is moved to the correct bucket.

The procedure for updating distance labels is described in Figure 2. Figure 3 presents the algorithm for seeking minimum length cycles containing node s. In this algorithm, *mldc* is the length of the minimum length cycle found so far. Figure 4 presents the entire algorithm for finding the min length cycle.

Procedure Distance-Update(s, i)

For all $j \in \Gamma_i^+$ **do**
  If $(d_s(j) > d_s(i) + c_{ij})$ **Then**
    $d_s(j) = d_s(i) + c_{ij}$;
    $pred(j) = i$;
    Bucket-Update(s, j);

Figure 2. The procedure for updating distance labels.

Procedure Min-Length-Cycle(s)

for $k = 1$ **to** $|W^*| - 1$ **do** $Bucket(k) = \emptyset$;
for $i = 1$ **to** n **do** $d_s(i) = +\infty$
$d_s(s) = 0$; $pred(s) = 0$;
Distance-Update(s, s);
while there are any non-empty buckets **do**
    $k = min\{i : Bucket(i) \neq \emptyset\}$;
    let j be the first node in $Bucket(k)$;
    delete j from $Bucket(k)$;
    if $(mldc > d_s(j) + c_{js})$ **then**
      $mldc = d_s(j) + c_{js}$;
      store $W = (s, j)$ to indicate that
        W gives the min length cycle;
    Distance-Update(s, j);

Figure 3. The procedure for seeking a minimum length cycle containing node s.

Algorithm MLDC (**Input**: *a graph G in which there is a path from node 1 to each other node*)

let W^* be the optimal MMC and $c(W^*)$ its length;
set $\lambda^* = c(W^*)/|W^*|$;
store W^* as the current MLDC and set $mldc = c(W^*)$;
let $p(k)$ denote the length of the shortest path from node 1 to any node k in G with respect to the length c', where $c'_{ij} = c_{ij} - \lambda$ for each $(i, j) \in A$;
for each arc $(i, j) \in A$, let $c^p_{ij} = c_{ij} + p(i) - p(j)$;
for $s = 1$ **to** n **do**
    Min-Length-Cycle(s);
Output the MLDC is W with length $mldc$;

Figure 4. The algorithm determining the MLDC.

In the usual implementation of Dial's shortest path algorithm, when selecting a node i from Bucket(k), one selects a node with minimum distance label. In Min-Length-Cycle(s), we select any node from $Bucket(k)$. This more flexible procedure is valid because of the following observation due to Denardo and Fox [9]. We include the proof for the sake of completeness.

LEMMA 3.1. *If j is any node in the minimum non-empty bucket then $d_s(j) = d(s, j)$. Thus, at the time that a node j is selected, $d_s(j) = d(s, j)$.*

Proof. Let B denote the minimum non-empty bucket, and suppose that bucket B contains distance labels in the range $[\alpha, \alpha + \lambda^*)$. Let S denote the nodes that were permanently labeled by Dijkstra's algorithm prior to scanning nodes in B. We assume inductively that $d_s(i) = d(s, i)$ for all nodes $i \in S$.

By the standard proof of Dijkstra's algorithm and by our inductive hypothesis, for all nodes $i \notin S$, $d(s, i) \geq \alpha$. For all nodes $j \in B$, $\alpha \leq d(s, j) < \alpha + \lambda^*$.

Let j be some node in B. Let P be the shortest path from s to j, and let node i be the predecessor of node j in P. Thus $d(s, j) = d(s, i) + c^p_{ij}$. We claim that $i \in S$. Otherwise, $d(s, j) = d(s, i) + c^p_{ij} \geq \alpha + \lambda^*$, which is a contradiction. Because $i \in S$, $d_s(j) = d(s, i) + c^p_{ij} = d(s, j)$, completing the proof.

THEOREM 3.1. *The algorithm Min-Length-Cycle(s) correctly determines in $O(m)$ time whether there are any cycles containing node s of length less than $|W^*|\lambda^*$.*

Proof. By Lemma 3.1, the algorithm correctly identifies shortest paths from node s as restricted to paths of length at most $(|W^*| - 1)\lambda^*$. Accordingly, it determines the shortest cycle containing node s as restricted to cycles of length at most $|W^*|\lambda^*$.

We next show that the running time is $O(m)$. The time to scan arcs and update distance labels and buckets requires $O(1)$ per arc and $O(m)$ in total. The time spent in identifying the minimum non-empty bucket is $O(n)$ in total because the index of the minimum non-empty bucket is non-decreasing from iteration to iteration, and one scans an empty bucket at most once. The time spent extracting nodes from buckets in the select operation is $O(1)$ per node and $O(n)$ in total. Thus the algorithm runs in $O(m)$ time when scanning from node s.

THEOREM 3.2. *Algorithm MLDC computes the minimum length directed cycle in G in $O(nm)$ time.*

Proof. The computation of MMC requires $O(nm)$ time using the algorithm of Karp [19]. The distances p are also obtained in $O(nm)$ time. Finally, each one of the n single shortest path computations runs in $O(m)$ by Theorem 3.1. Therefore, the running time of the proposed MLDC algorithm is $O(nm)$ time.

3.1 Approximating min cycle mean. The $O(nm)$ time algorithm for the min length cycle relied on Karp's algorithm for the min cycle mean. However, it suffices to approximate the min cycle mean. If we were to obtain reduced costs that were at least $\lambda^*/2$, then Min-Length-Cycle(s) could be run in $O(m)$ time by using $2n$ buckets rather than n buckets.

In this subsection, we outline how to approximate the min cycle mean in $O(n^{.5} m \log n)$ time by relying on

the algorithm of Orlin and Ahuja [24] for solving the min cycle mean.

The Orlin-Ahuja algorithm finds the cycle with the min cycle mean in a sequence of scaling phases. Each scaling phase runs in $O(n^{.5}m)$ time. In each scaling phase, the algorithm maintains an interval $[LB, UB]$ that is guaranteed to contain the value $\lambda^*/2$. In addition, there are node potentials such that the reduced cost of each arc is at least LB.

For a network with non-negative arc lengths, the initial value of LB is 0, and the initial value of UB is $c_{\max}$. At each subsequent phase, LB or UB is modified. Suppose, for example, that LB is replaced by LB' and that UB is replaced by UB'. The replacements are carried out so as to guarantee that $UB' - LB' < .75(UB - LB)$.

To use the Orlin-Ahuja Algorithm to find an approximate value of $\lambda^*/2$, we carry out the following steps, starting from a network with non-negative arc costs. We refer to this approach as the Truncated O-A Algorithm.

1. Determine the minimum value K such that there is a directed cycle W_0 in G such that every arc of W_0 has length at most K.
2. Let $A' = \{(i,j) \in A : c_{ij} \leq K\}$.
3. Use the Orlin-Ahuja Algorithm starting with A' and terminating after the first phase for which $UB - LB < K/2n$.

LEMMA 3.2. *The Truncated O-A Algorithm runs in $O(n^{.5}m \log n)$ time. At termination, each arc has a reduced cost that is at least $\lambda^*/2$.*

Proof. We first note that $\lambda^*/2 \leq K$ because $c(W_0)/|W_0| \leq K$.

The first step of the algorithm can be carried out by sorting the arc lengths, and then using binary search to determine K. The time for the first step of the Truncated O-A Algorithm is $O(m \log n)$. Step 2 takes $O(m)$ time. The number of scaling phases in Step 3 is $O(\log n)$ because the algorithm starts with $UB = K$, and $LB = 0$, it ends with $UB - LB > K/4n$, and $UB - LB$ is decreasing geometrically. Therefore, Step 3 takes $O(n^{.5}m \log n)$ time.

We next note that $\lambda^*/2 \geq K/n$ because every cycle of G contains an arc (i,j) with $c_{ij} \geq K$. Accordingly, at the end of the Truncated A-O Algorithm, $UB \geq K/n$, and $LB \geq UB - K/2n \geq .5UB \geq .5\lambda^*/2$. This completes the proof.

3.2 Speed-ups used in our experiments. In the next section, we will describe our experimental results. In order to speed up the cycle detection in our experiments, we include a variety of practical improvements to the algorithms. None of the improvements affects the worst case running time except for use of Howard's [17] Algorithm to solve the MMC, which does increase the worst case running time of the overall algorithm.

Speeding up the search for the MMC in practice. The fastest algorithm in practice for finding the minimum mean cycle is due to Howard [17], even though it's time complexity is greater than the $O(nm)$ time bound achieved in this paper. We also carried out preliminary tests on the scaling network simplex algorithm of Ahuja and Orlin [2], which runs in $O(nm \log nC)$ time and is similar in spirit to Howard's Algorithm. In our initial tests,
Howard's Algorithm was faster than the scaling simplex algorithm, and so we restricted our testing to Howard's Algorithm. Nevertheless, in each of our initial tests, Howard's Algorithm was faster by a factor less than 2. Perhaps with some additional heuristic speedups, the scaling simplex algorithm could dominate Howard's Algorithm empirically.

Using integer arithmetic. In computing the reduced costs for MLDC, we used distance labels for the problem instance in which c_{ij} is replaced by $c_{ij} - \lambda^*$, where λ^* is the length of the minimum mean cycle. In our experiments, the original arc lengths are all integer valued, and thus λ^* is a rational number whose denominator is less than n. Rather than use fractional arc lengths, we first multiply all original arc lengths by the denominator of λ^*. This creates an equivalent problem in which all data is integral and all calculations use integer arithmetic.

Stopping the search after computing the MCC. If the minimum mean cycle W^* has only two arcs or if its total length is 0, then it is also a minimum length cycle. If W^* has only three arcs, then it is either the minimum length cycle or there else the minimum length cycle has only two arcs. In this latter case, it suffices to check all cycles with two arcs, which can be done in $O(m)$ time. (It can be sped up to $O(n)$ time with additional data structures.) We did not rely on this particular speed-up in our experiments, but it could be useful in practice.

Strongly connected components. Initially, the graph is partitioned into its strongly connected components. This preprocessing leads to a speed-up because searches can be restricted to the strongly connected components. Because computing the strongly connected components takes $O(m)$ time, one would not want to recompute them after each shortest path computation.

Deleting nodes with index less than s. We

seek shortest paths from nodes 1, 2, 3, ..., n. After the algorithm caries out Min-Length-Cycle(s) for some $s \in [1, n]$, it (permanently) deletes node s as well as its incident arcs from G. By "permanently", we mean that the algorithm deletes the nodes and arcs in all subsequent shortest path computations.

Deleting source nodes and sink nodes. The algorithm keeps track of the indegree and the outdegree of each node. It (permanently) deletes any node whose indegree is 0 or outdegree is 0.

Deleting arcs with large length. Let $mlcd$ be the length of the shortest cycle found at some iteration. The algorithm (permanently) deletes any arc (i, j) whose length is at least $mldc - \lambda^*$ since the arc cannot be in a shortest cycle.

Initializations of the vector of the distance labels. In our initial implementations, we created a distance vector d for each source node s. This initialization took $O(n^2)$ time in total, and was the computational bottleneck of our algorithm. We eliminated this bottleneck and sped up the computations by using a single vector of distances labels d for the n shortest path computations. Prior to the shortest path computations from the first source node, each position of this vector is set to $mldc - \lambda^*$. When finding shortest paths from a given source node, a dynamic vector $ModifiedLabels$ contains the label of each node whose distance label is less than $mldc - \lambda^*$. Prior to selecting the next source node, the distance labels of the nodes in $ModifiedLabels$ are reset to $mldc - \lambda^*$ and $ModifiedLabels$ becomes empty.

3.3 An example. In this section, we show the execution of the MLDC algorithm on the graph given in Figure 5a. The MMC is $3 \to 4 \to 6 \to 5 \to 3$ with length 6. The value of λ is 3/2. Figure 5b gives the shortest path distances p from node 1 with respect to lengths $c - \lambda$. The arcs out of the shortest path tree appear in dotted lines. We did not multiply lengths by 2 in our example and instead used fractional arc lengths.

Figure 5c displays the reduced cost $c^p_{ij} = c_{ij} + p(i) - p(j)$ for each arc $(i, j) \in A$. Note that the reduced cost of any arc in the MMC is 3/2. Arcs $(2, 3)$ and $(4, 5)$ appear in dotted lines indicating that they will be deleted prior to finding shortest paths, as per the speedup suggested above (see "Deleting arcs with large length"). Figure 5d displays the shortest path trees from nodes 1 and 3, as well as the directed cycles induced by the shortest path trees. When computing shortest paths from node j, we first delete nodes 1 to $j-1$ as per the speedup suggested above (See "Deleting nodes with index less than s").

After finding the shortest paths from node 1, we delete node 1. At this point, we delete node 2 because it has no incoming arcs (See "Deleting source nodes and sink nodes"). We next compute shortest paths from node 3, and then delete node 3. At this point, node 4 has no incoming arc and is deleted. Subsequently node 5 is deleted because it has no outgoing arc. And finally, node 6 is the only node that remains, and is deleted. The minimum length cycle is $3 \to 4 \to 6 \to 3$ with

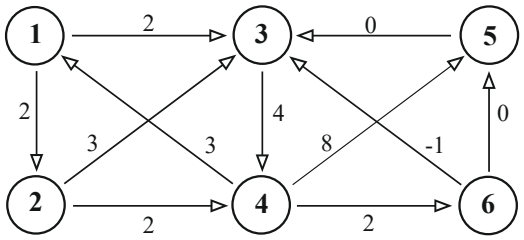

Figure 5a. Example graph.

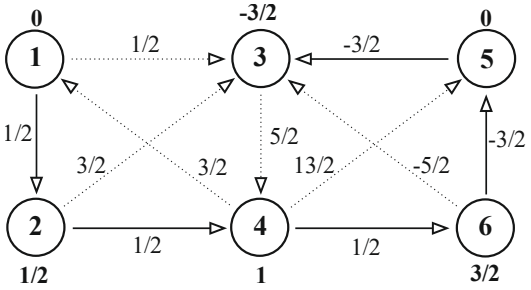

Figure 5b. Shortest path distances p.

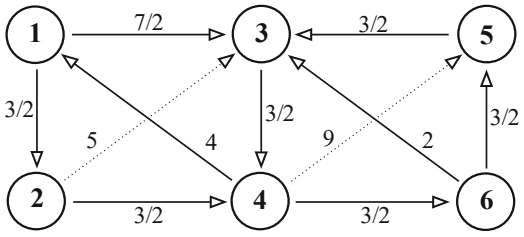

Figure 5c. Reduced costs c^p

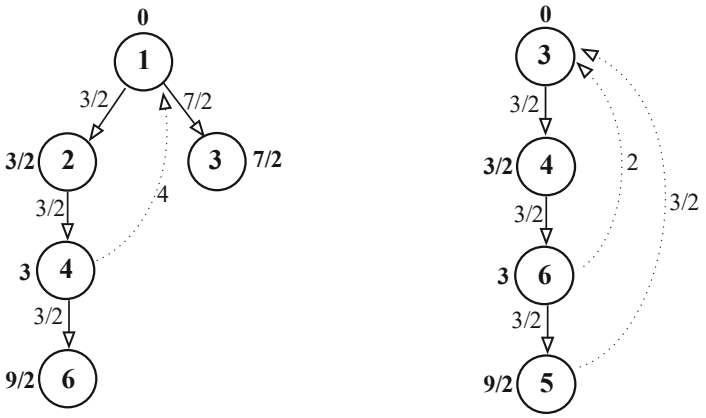

Figure 5d. The shortest path trees in G with reduced costs c^p.

length 5, which was determined when finding shortest paths from node 3.

4 Experimental Results.

4.1 Algorithms. All algorithms in this experiment were written in C and compiled with gcc ver. 5.2 with the option -O. The tests were performed on an Intel Xeon (R) CPU ES-1620 v2 3.7 GHz x 8 processor and 64GB of RAM running Ubuntu 14.04 LTS.

All test instances use integer arc lengths. Each of the implementations of the algorithms employs only fixed-point arithmetic. The graphs are not necessarily connected. In each case, we first identify the strongly connected components (*scc*) using the algorithm of Tarjan [30]. Then, each algorithm in the experiment is executed separately in each scc. When computing the scc's, we relabel nodes in depth first search order as per Tarjan's Algorithm. Running times do not include reading the graph, but do include the computation of the ssc.

Floyd and Warshall's Algorithm (FW). We have included the Floyd [12] and Warshall [32] Algorithm in the experiment. For the FW Algorithm, $d^k[i,j]$ denotes the shortest path length from i to j as restricted to nodes in $\{i,j\} \cup \{1,2,...,k-1\}$. At the k-th iteration, the algorithm computes $d^k[i,j]$ for all nodes i and j, including that case in which $i = j$. One can compute $d^{k+1}[i,j]$ using the following recursive relationship: (see Ahuja et al., [1], page 148) $d^{k+1}[i,j] = \min\{d^k[i,j], d^k[i,k] + d^k[k,j]\}$.

Suppose that one computes $d^{k+1}[i,j]$ for a fixed k and i and letting j vary. One can speed up calculations in practice by avoiding computations when $d^k[i,j] \geq mldc$.

Binary heap algorithm (BHEAP). We have included a binary heap implementation of Dijkstra's shortest path algorithm in the experiment. In the case that the graph contains any arc with negative length, initial optimal distances are calculated using Pallotino's [25] Algorithm, which is a variant of the label correcting algorithm. The remaining shortest path trees are computed using a binary heap implementation of Dijkstra's Algorithm. All heap operations take log(size(heap)) time. Each shortest path computation from node s only tries to reach nodes in the same ssc as s, and restricted to nodes with an index greater than s, and restricted to arcs whose distance label is inferior to *mldc* (the current best value for the MLDC found by the algorithm).

Additionally, we have considered bidirectional variants of Dijkstra's Algorithm. That is, two simultaneous executions of Dijkstra's Algorithm. Since, we are searching for a cycle containing node s, both executions start from node s. The first examines the successor nodes and the second examines the predecessor nodes of each recent permanently labeled node. The implementation alternates one step in the forward direction with a step in the backward direction. Once a node different from node s is permanently labeled in both directions, the algorithm ends. It would also end if the value of the sum of the minimum keys in the two heaps is greater than the current value of the *mldc*. The value of the *mldc* is updated whenever a shorter cycle is determined. This occurs when the sum of the distance labels in both direction of a node is smaller than *mldc*. We code this algorithm as BBHEAP.

These two implementations incorporate the practical improvements commented in Section 3.1. with the exception of "Stopping the search after computing the MCC".

We developed six different variants of the algorithm (referred to as OSN) that we described in Sections 2 and 3. All of them incorporate Howard's [17] Algorithm to solve the MMC problem. A detailed description of Howard's [17] Algorithm adapted to solve the MMC can be found in Dasdan [7]. Howard's [17] Algorithm is quite simple and is as fast or faster in practice than other proposed algorithms in the literature. (Dasdan et al. [8], Dasdan [7] and Georgiadis et al.[15].) In addition, Howard's Algorithm provides the vector p of node potentials, which is needed for our algorithm. More precisely, Howard's Algorithm ends with a vector π, where $\pi = -p$.

All the arithmetic operations are made in fixed-point (integer arithmetic) with the exception of the operation calculating λ. If λ is fractional, we multiply all original arc lengths by $|W|$, which is its denominator. Subsequently, all calculations can be carried out using integer arithmetic. At termination of the shortest path algorithms, we divide all values again by $|W|$.

In our first implementation of Dial's bucket-based algorithm, referred to as OSNB, we store buckets using doubly linked lists so that it supports the operation of deletion in $O(1)$ time. A deletion from $Bucket(k)$ occurs in one of two different situations. It can occur when $Bucket(k)$ is the least index non-empty bucket and we are deleting the first element of $Bucket(k)$. It also occurs when a node j is moved from $Bucket(k)$ to a lower index bucket. Only in the latter case are doubly linked lists needed. A singly linked list is adequate for an $O(1)$ deletion in the former case.

In an implementation referred to as OSNBL, we implemented the buckets using singly linked lists. (The L stands for "lazy.") Singly linked lists supports the operation of deletion of the first element in a bucket in $O(1)$ time. But it would be inefficient to delete j from $Bucket(k)$ following a decrease in $d_s(j)$. Instead,

we carried out "lazy deletions." If j is moved from $Bucket(k)$ to a lower index bucket, we add j to the lower index bucket, but do not delete it from $Bucket(k)$. Dijkstra's Algorithm will permanently label node j prior to $Bucket(k)$ becoming the lowest index non-empty bucket. When the algorithm finds node j in $Bucket(k)$, it deletes j from $Bucket(k)$ at that time. This singly-linked implementation with lazy deletions uses space more efficiently, and is faster in practice, but only by a small amount, as revealed in our experiments.

In an implementation that we refer to as OSNH, we implement the shortest path algorithms using binary heaps. This increases the worst case running time of the shortest path computations, although it does not appear to affect the performance in practice. OSNH differs from BHEAP in that OSNH first uses Howard's Algorithm to solve the MCC and modify the distances. BHEAP does not solve the MCC.

Our final three implementations are the same as the previous three except that we implement the bidirectional version of Dijkstra's Algorithm. Accordingly, we refer to these three implementations as BOSNB, BOSNBL and BOSNH, respectively.

All six implementations of Dijkstra's Algorithm incorporate the practical improvements described in Section 3.1. For each instance of the MLDC that we solved, we kept track of the CPU time of Howard's Algorithm. For each problem instance and for each of the six different implementations of the shortest path algorithm, we kept track of the total CPU time in seconds (including Howard's CPU time).

4.2 Test instances. We considered using the instances used in previous computational studies of the MMC problem. Dasdan [7] and Georgiadis [15] used real-world problem families representing clocking problems on circuits. Their experiments showed that these instances are quickly solved by most algorithms. On the other hand, synthetic families originally used in the evaluation of algorithms for the shortest path problem are more difficult to solve and require additional computational effort. For this reason, we focused our experiments on synthetic instances, as generated by network generators.

Our first set of instances was created using the NETGEN generator, which was developed by Klingman et al. [20]. We generated random graphs with $n \in \{8000, 16000, ..., 256000\}$ and $m \in \{2n, 4n, ..., 128n\}$. The largest of these graphs contains approximately 32 million arcs. This is close to the maximum number of arcs that NETGEN permits, which is 40 million. The following parameters were fixed for all problems: $sources = 1$, $sinks = 1$, $Tsources = Tsinks = 0$, $mincost = 1$, $maxcost$ =10,000 and $\%highcost = 5$. We generated ten replications for each combination of the parameters n and m, resulting in 420 (6 x 7 x 10) different instances.

Our second set of instances was created using the GRIDGEN generator developed by Lee and Orlin [21]. We present results for random graphs with n varying from 8,000 to 256,000 (as in the previous set of instances) and the parameter $width \in \{2, 4, ..., 128\}$, where *width* is the width of the network. As the width of the network increases, the "length" decreases accordingly so that the number of nodes is not affected. The following parameters were fixed for all problems: $sources = 1$, $sinks = 1$, $average\_degree = width$, $mincost = 1$ and $maxcost$ = 10,000. In this case, we generated eleven replications for each combination of the n and *width*, resulting in 462 instances.

We developed our own generator in order to create the third set of instances. This generator, which we call HPGEN first constructs a *Hamiltonian* path in which arc length are chosen uniformly at random of the interval [1,10]. Each of the remaining $m - n + 1$ arcs is incident to a randomly selected pair of nodes. The length of each of these arcs is fixed uniformly at random in the interval $[1, 10, 000]$. We generate random graphs with n and m taking the same values as in our first collection of problem instances. As before, we created ten replications for each combination of these parameters, resulting in 420 instances.

All arc lengths are nonnegative in our experiments. Accordingly, the
BHEAP and BBHEAP algorithms do not need to carry out a label correcting algorithm in order to transform the costs into equivalent nonnegative costs.

4.3 Results. Conclusions from computational tests are limited in nature. They are limited by the network generators that are selected and by bounds on the parameters of the test cases. Nevertheless, there are some conclusions that are robust in the context of our computational experiments. We will state the conclusions and offer a partial justification for the conclusion. In general, we present graphs that demonstrate the conclusions.

Conclusion 1. Floyd-Warshall was not competitive with the other algorithms. In our initial experiment, we observe that the Floyd-Warshall Algorithm was not competitive with the other algorithms. For example, FW needed 183 seconds to solve instances with 50,000 nodes and 95,000 arcs while the other algorithms used at most 0.41 seconds. Because FW is orders of magnitude less efficient in our experiments, and because FW's running grows as $O(n^3)$, we carried out limited testing, and do not present further results

on FW.

Conclusion 2. The bidirectional version of Dijkstra's Algorithm did not substantively improve the running times. We implemented three versions of the OSN algorithm using Dijkstra's Algorithm, and three with bidirectional Dijkstra's Algorithm. In addition, BHEAP used an implementation of Dijkstra's Algorithm, whereas BBHEAP used the bidirectional version. In Figure 6, we graphed the ratio of the CPU time of the original version divided by the CPU time of the bidirectional version. In each case, we graphed the ratio as a function of n, the number of nodes, measured in 1000s. Each data point reflects the average of 70 instances (seven different values of m/n, and ten replications for each parameter setting). The graphs reveal that the bidirectional versions of OSNB and OSNBL were faster than the original versions, but only by very modest amounts. The bidirectional version of OSNH was slower than the original version. For BHEAP, the results were mixed.

It was initially surprising to us that there was not a more substantive improvement for using bidirectional versions Dijkstra. In usual shortest path algorithms, bidirectional Dijkstra leads to a more substantive speed-up because it greatly reduces the number of nodes that are made permanent. In the case of road networks, the speed-up is typically by a factor between 1.5 and 2. Intuitively, if the distance from a source node s to a sink node t is r, then Dijkstra's Algorithm will permanently label all nodes of distance at most r from s. And bidirectional Dijkstra, will permanently label any node of distance at most $r/2$ from either s or t. In a planar network with randomly placed nodes, the improvement is by nearly a factor of two in the number of nodes that are labeled permanently. We conjecture that we did not see a larger speedup because the speed-ups given in Section 3.1 dramatically reduce the number of nodes that are made permanent, and the additional speed-up from using bidirectional Dijkstra is not so great.

Conclusion 3. BOSNB and BOSNBL are the fastest of the six OSN implementations and also faster than both implementations of the BHEAP algorithm. We illustrate this in Figure 7, where we graph the following ratios: CPU(BOSNBL)/CPU(BOSNB), CPU(OSNH)/CPU(BOSNB), and CPU(BHEAP)/CPU(BOSNB) for the three network generators. In each case, we graphed the ratios as a function of n, measured in 1000s.

Figure 7a shows that BOSNB is comparable in running time to BOSNBL. Both of these implementations includes the running time of Howard's Algorithm. If we were to subtract off this running time, BOSNB would still be comparable to BOSNBL, but the relative difference would be greater.

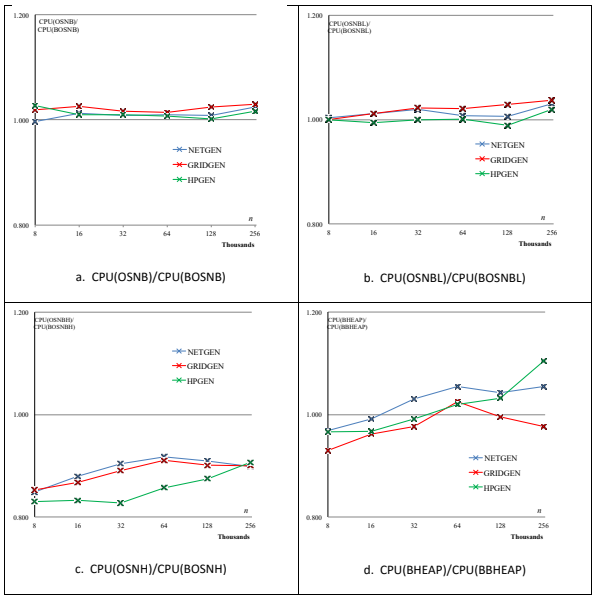

Figure 6. The CPU time of the implementation with Dijkstra's Algorithm divided by the CPU time of the implementation using the bidirectional version of Dijkstra's Algorithm, graphed as a function of n (in 1000s).

Figure 7b shows that BOSNB is 10% to 30% faster than OSNH for all values of n and all network generators. As before, this ratio would increase if we were to subtract off the time of Howard's Algorithm from BOSNB and OSNH.

Figure 7c shows that BOSNB is faster than BHEAP for the parameter settings that we tested; however, the ratio is decreasing as a function of n, suggesting that BHEAP may be asymptotically faster.

In Figure 8, we graph the CPU times of BOSNB, BHEAP, and Howard's Algorithm as a function of n for all three generators.

Conclusion 4. Running Howard's Algorithm leads to a speedup for solving the MLDC for the instances in our experiments. Nevertheless, Howard's Algorithm is the bottleneck operation. We have already seen in Figure 7 that BOSNB is faster than BHEAP (and BBHEAP), which implies that it was advantageous to run Howard's Algorithm to solve the MMC prior to solving shortest path problems. In Figure 9, we show that Howard's Algorithm is the bottleneck operation. We first graph the proportion of the time of BOSNB taken by Howard's Algorithm. As shown in Figure 9a, this time varies from 64% to 85%. We then subtracted the CPU time for Howard's Algorithm (denoted as CPU(H)) from the CPU time

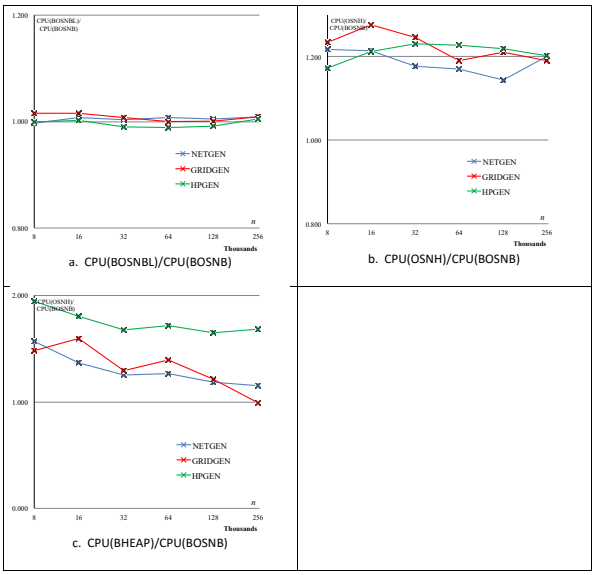

Figure 7. The ratio of the CPU time of BOSNBL, BOSNH, and BHEAP to the CPU time of BOSNB, graphed as a function of n (in 1000s).

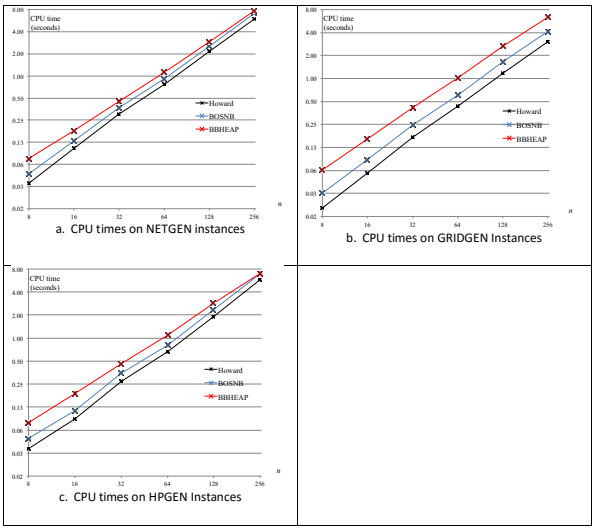

Figure 8. The CPU times for Howard's Algorithm, BOSNB, and BHEAP as a function of n (in 1000s) for all three network generators.

of BOSNB. We denote the difference as CPU(BOSNB - H). In Figure 9b, we graphed the ratio of CPU(H) to CPU(BOSNB - H). We see that the CPU time for running Howard's Algorithm to solve the MMC is two to six times greater than the time to solve all $O(n)$ subsequent shortest path problems to find the MLDC.

Conclusion 5. BHEAP is faster than BOSNB for small values of d but is slower than BOSNB for larger values of d. In the previous tables and graphs, we averaged the CPU times over all graphs with

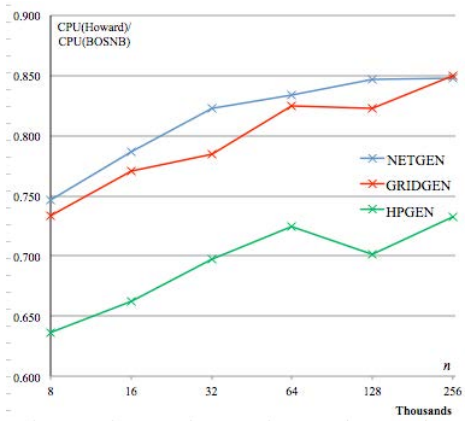

Figure 9a. The CPU times for Howard's Algorithm divided by the CPU time for BOSNB, which includes the time for Howard's Algorithm.

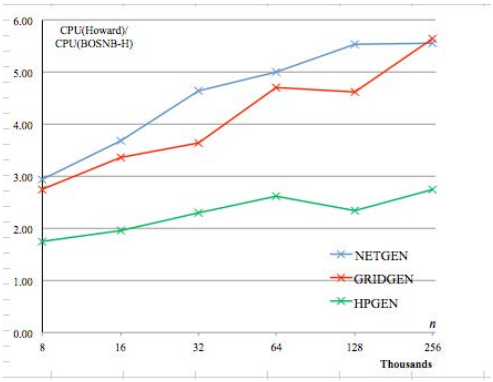

Figure 9b. The CPU times for Howard's Algorithm divided by the CPU time for BOSNB - H, which does not include the running time of Howard's Algorithm.

n nodes. Here for fixed d, we average the CPU times over all values of n. Although this type of averaging is not guaranteed to reveal structure in computational tests, in our case it leads to some useful insights. Figure 10 shows the average CPU time for each value of the density of the graph $d = m/n$. As in many of the previous figures, the CPU times of the BOSNB includes the running time for Howard's Algorithm. Figure 10 reveals that increasing d leads to an increase in the running time. However, doubling d does not lead to a doubling of the CPU time for any of the algorithms. Moreover, Howard's Algorithm is far less sensitive to increases in d as is BHEAP.

For example, for Howard's Algorithm on NETGEN instances, as d increases from 2 to 128, the average running time increases from 0.8 seconds to approximately 3 seconds, which is an increase of less than a factor of 4.

The sensitivity to d was large for the BHEAP

algorithm on NETGEN instances but not for other generators. As d increased from 2 to 128, the CPU time for the BHEAP algorithm increased by a factor of 13.4 for NETGEN instances, 3.67 for GRIDGEN instances, and 5.23 for HPGEN instances. We do not have an explanation for why BHEAP was so sensitive to the parameter d on NETGEN instances.

The sensitivity of BHEAP to d impacts on which algorithm is faster. For small values of d, BHEAP was faster than BOSBN. For larger values of d, BOSBN was the faster of the two algorithms. This was true for all three network generators, as can be seen in Figure 10.

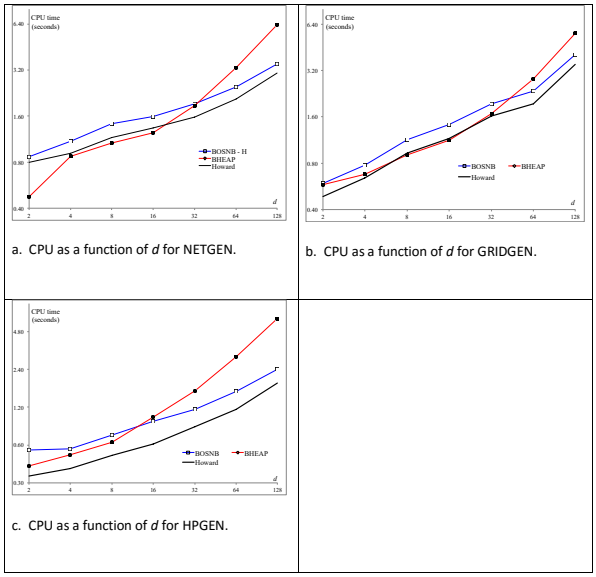

Figure 10. The CPU times as a function of d for the three generators.

Conclusion 6. Regression analysis provides a good estimate for the running time of Howard's Algorithm and for BOSNB - H, and gives an excellent estimate for the running time of BHEAP.

For three algorithms and for each of the network generators, we ran a nonlinear regression to estimate the CPU time as a function of the number of nodes and arcs of the network. The three algorithms that we considered were (1) Howard's Algorithm, (2) BOSNB - H, and (3) BHEAP. This led to nine regressions in total. Each regression was based on 42 points, representing 42 possible choices of n and d. The CPU times were averaged over the 10 instances for each setting of the parameters for NETGEN and HPGEN and over 11 instances for GRIDGEN. For each regression, we estimated the running time in terms of the parameters n and $d = m/n$. We used the form $\alpha n^{\beta} d^{\gamma}$ which is equal to $\alpha n^{\beta-\gamma} m^{\gamma}$ We considered multiplying by $\log_2(n)$ for estimating BHEAP to take into account the log n time taken for a heap operation. However, this did not have much impact on the regression analysis. And there is a simple reason why one would not expect it to have an impact. As n varies from 8,000 to 256,000, $\log_2(n)$ is approximately equal to $5.54n^{0.095}$. In fact, within this range of values of n, $5.54n^{0.095}$ differs from $\log_2(n)$ by less than 1%.

The results of the regressions are summarized in Table 1. We graph the results of the regressions in Figure 10.

	NETGEN	GRIDGEN	HPGEN
BOSNB - H	$9.22 \times 10^{-8} n^{1.25} d^{0.25}$ $R^2 = .73$	$1.61 \times 10^{-8} n^{1.36} d^{0.36}$ $R^2 = .75$	$6.13 \times 10^{-8} n^{1.28} d^{0.28}$ $R^2 = .70$
Howard's Algorithm	$1.32 \times 10^{-7} n^{1.33} d^{0.33}$ $R^2 = .80$	$3.52 \times 10^{-9} n^{1.56} d^{0.56}$ $R^2 = .61$	$1.35 \times 10^{-8} n^{1.43} d^{0.43}$ $R^2 = .86$
BHEAP	$2.90 \times 10^{-8} n^{1.36} d^{0.73}$ $R^2 = .97$	$7.19 \times 10^{-8} n^{1.26} d^{0.76}$ $R^2 = .97$	$2.11 \times 10^{-8} n^{1.33} d^{0.87}$ $R^2 = .98$

Table 1: Regression analysis for estimation of CPU times.

The regression analysis offers some useful insights. For example, if we contrast the regression estimate for Howard's Algorithm and the BHEAP for NETGEN, we would conclude that the breakeven point is when d is approximately 44, and that BHEAP would have a lower CPU time for lower values of d. The CPU times do differ from the estimated times, and the breakeven point seems to be around 20 for our test cases; but the fact that there was a breakeven point depending on d was born out by our tests.

The regression estimates for GRIDGEN are less informative, in part because the estimate for the CPU time for Howard's Algorithm (called Est(H)) is not as good. Also, Est(BHEAP)/Est(H) $\approx 20d^{.2}/n^{.3}$ which does not lend itself as well to our graphical visualization. In any case, it does suggest that BHEAP will be faster if n is large and d is not too large. For example, if n =10,000,000, then Est(BHEAP) < Est(H) whenever $d <$ 9,000.

For HPGEN, the regression estimates suggest that Howard's Algorithm will dominate BHEAP except for extremely large values of n and very small values of d. However, the running time of BOSNB will be greater than that of BHEAP for $n <$ 256,000 and small values of d. For example, if n = 64,000, then the estimated time for BHEAP is less than the estimated time for BOSNB whenever $d <$ 13. This is consistent with Figure 10c.

We show the graphs of the estimated CPU times and the real CPU times of BOSNB-H, Howard's Algorithm, and BHEAP in Figure 11 to 13. In each of the 12 graphs of each figure, we fix the value of d at 2, 8,

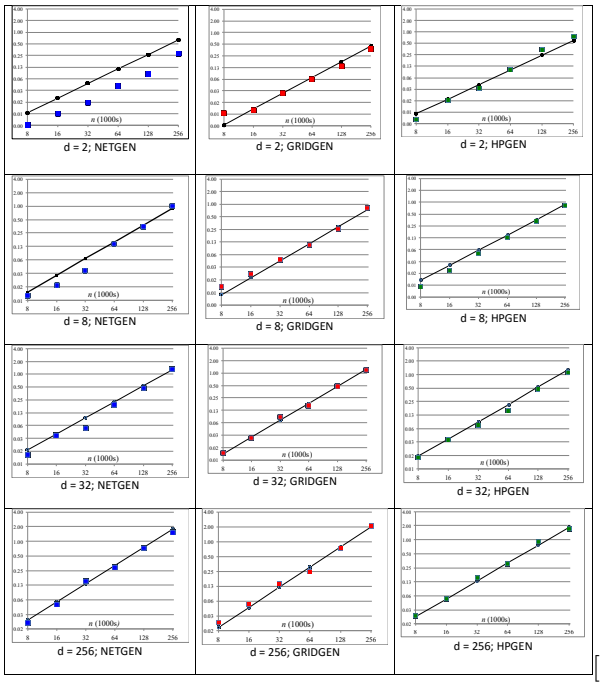

Figure 11. Regression estimates of the CPU time for BOSNB minus the time for Howard's Algorithm. The x-axis is the number of nodes in 1000s. The estimated curve is the black line.

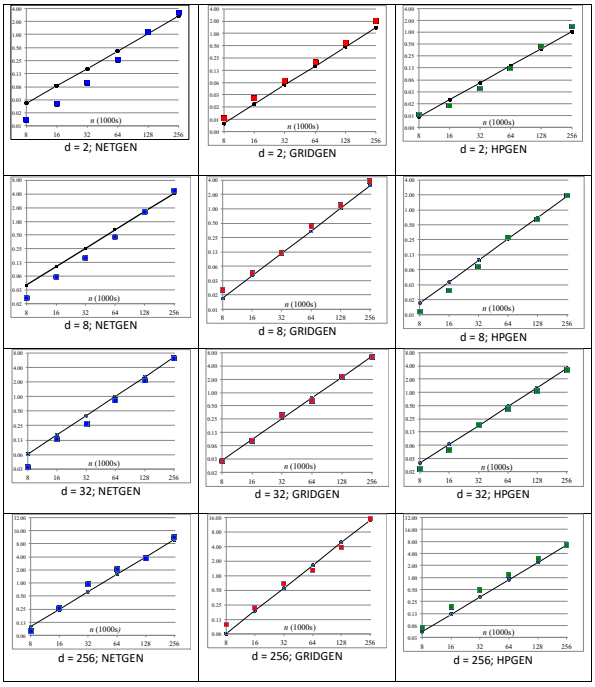

Figure 12. Regression estimates of the CPU time for Howard's Algorithm. The x-axis is the number of nodes in 1000s.

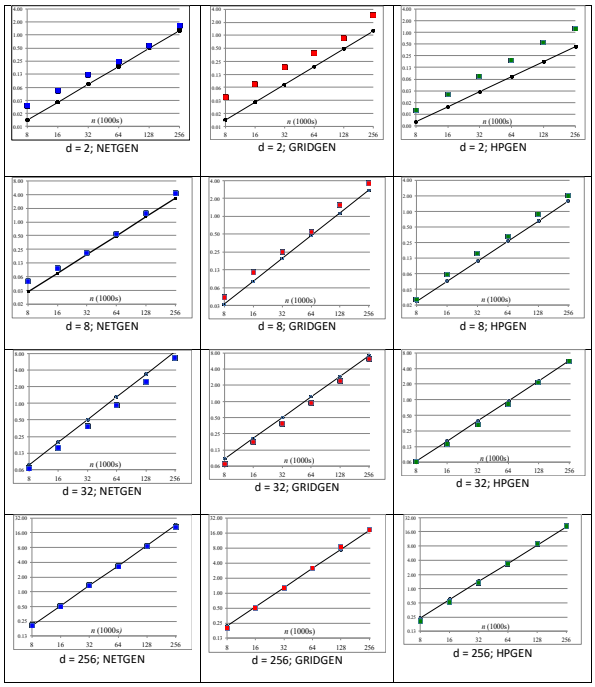

Figure 13. Regression estimates of the CPU time for BHEAP. The x-axis is the number of nodes in 1000s.

32 or 128, and we fix the network generator. For any given value of n, the CPU time given is the average over the 10 instances with the given parameter settings. The graphs reveal that the regression lines are an excellent fit with the possible exception of some of the estimations when $d = 2$. Recall that we are averaging CPU times over 10 instances, which leads to a better fit.

Conclusion 7. BOSBN and BHEAP solve the MLDC much faster than the worst case analysis would suggest. Recall that the best worst case analysis for MLDC is $O(nm)$, and the running time of BHEAP is
$O(nm \log n)$. If the BHEAP regression analysis were accurate for large values of n and m, and substituting m/n for d, then the running time of BHEAP in practice will be as follows: $O(n^{.63}m^{.73})$ for NETGEN instances, $O(n^{.5}m^{.76})$ for GRIDGEN instances, and $O(n^{.46}m^{.87})$. In each case, the estimated run time is far better than the worst case run time.

5 Summary.

Our primary theoretical result is a faster algorithm for finding a minimum length directed cycle in directed graphs. We solved the problem in $O(nm)$ time improving upon the previous best bound of $O(nm + n^2 \log \log n)$.

In our computational experiments, we relied on using Howard's Algorithm to solve the MMC. In gen-

eral, we observed that solving the MMC before the MLDC does lead to a speed up in the running time for the instances we generated from NETGEN, GRIDGEN, and HPGEN. The speed-up is due to a reduction in the number of nodes that need to be scanned for each shortest path computation. However, if the results of our experiments could be extrapolated to larger values of n, it seems likely that skipping the solution of the MMC is preferable for small values of $d = m/n$ and for $n > 1,000,000$.

We also observed that the bidirectional version of Dijkstra's Algorithm used in our implementations to solve the MLDC led to modest improvements in the run time. A surprising (and significant) observation is that the Howard's Algorithm is typically the bottleneck operation in solving the MLDC despite the fact that Howard's Algorithm runs much faster than $O(nm)$ time in practice. Further improvements in the run time for the MMC will lead to corresponding improvements in the run time for solving the MLDC.

References

[1] Ahuja, R., Magnanti, T., & Orlin, J.B., *Network Flows*, Prentice-Hall, Inc., 1993

[2] Ahuja, R., & Orlin, J.B., *The scaling network simplex algorithm*, Operations Research 40 (1992), S5- S13.

[3] Balinski, M.L., *An algorithm for finding all vertices of convex polyhedral sets*, Journal of the Society for Industrial and Applied Mathematics 9 (1961), 72-88.

[4] Bellman, R., *On a route problem*, Quart. Appl. Math. 16 (1958), 87-90.

[5] Chegireddy, C.R., & Hamacher, H.W., *Algorithms for finding K best matchings in an undirected graph*, Annals of Discrete Mathematics 18 (1987), 155-165.

[6] Coppersmith, D and Winograd, S. *Matrix multiplication via arithmetic progressions*, J. Symbolic Computation 9 (1990), 251 280.

[7] Dasdan, A., *Experimental analysis of the fastest optimum cycle ratio and mean algorithms*, ACM Transactions on Design Automation of Electronic Systems 9 (2004), 385-418.

[8] Dasdan, A., Irani, S., & Gupta, R.K., *Efficient algorithms for optimum cycle mean and optimum cost to time ratio problem*, In Proceedings of the 36th Design Automation Conference (1999), 37-42.

[9] Denardo, E.V. and Fox, B.L. *Shortest-route methods 1: Reaching, pruning, and buckets*, Operations Research 27 (1979), 161-186.

[10] Dial, R.B., *Algorithm 360: Shortest path forest with topological ordering*, Communications of the ACM 12 (1969), 632-633.

[11] Dijkstra., E.W., *A note on two problems in connection with graphs*, Numer. Math. 1 (1959), 269-271.

[12] Floyd, R.W., *Algorithm 97: shortest path*, Comm. ACM 5 (1962), 345.

[13] Ford, L.R., (1956). *Network flow theory*, The Rand Corporation Report P-923, Santa Monica, Calif.

[14] Fredman, M.L. & Tarjan, R.E., *Fibonacci heaps and their uses in improved network optimization algorithms*, Journal of the ACM 34 (1987), 596-615.

[15] Georgiadis, L., Goldberg, A.V., Tarjan, R.E., & Werneck, R. F., *An experimental study of minimum mean cycle algorithms*, in Proceedings of the 11th Workshop on Algorithm Engineering and Experiments (ALENEX). SIAM, Philadelphia, (2009) 1-13.

[16] Hagerup, T., *Improved shortest paths on the word RAM*, International Colloquium on Automata, Languages, and Programming, (2000), 61-72.

[17] Howard, R.A. *Dynamic Programming and Markov Processes*, The M.I.T. Press, Cambridge, Mass., 1960.

[18] Itai, A., & Rodeh, M., *Finding a minimum circuit in a graph*, SIAM J. Computing 7 (1978), 413 - 423.

[19] Karp, R.M., *A characterization of the minimum cycle mean in a digraph*, Discrete Mathematics 23 (1978), 309-11.

[20] Klingman, D., Napier, A., & Stutz, J., *NETGEN: A program for generating large scale assignment, transportation, and minimum cost flow problems*, Management Science 20 (1974), 814-821.

[21] Lee, Y., & Orlin, J. B., GRIDGEN, ftp://dimacs.rutgers.edu/pub/netflow/generators/network/gridgen/, 1991.

[22] Le Gall, F., *Powers of tensors and fast matrix multiplication*, in Proceedings of the 39th international symposium on symbolic and algebraic computation, (2014), 296-303.

[23] Moore, E.F., *The shortest path through a maze*, in Proceedings of an International Symposium on the Theory of Switching, 2-5 April 1957, Part II, The Annals of the Computation Laboratory of Harvard University 30, Harvard University Press, 1957, 285-292.

[24] Orlin, J.B., & Ahuja, R.K., *New scaling algorithms for the assignment and minimum mean cycle problems*, Mathematical Programming 54 (1992), 41-56.

[25] Pallottino, S., *Shortest-path methods: Complexity, interrelations and new propositions*, Networks 14 (1984), 257-267.

[26] Pettie, S., *A new approach to all-pairs shortest paths on real-weighted graphs*, Theoretical Computer Science 312 (2004), 47-74.

[27] Provan J,S., *Efficient enumeration of the vertices of polyhedra associated with network LP's*, Mathematical Programming 63 (1994), 47-64.

[28] Roditty, L. & Williams, V.V. *Minimum weight cycles and triangles: equivalences and algorithms*, arXiv:1104.2882v1 14 Apr 2011.

[29] Sedeño-Noda, A., & Espino-Martin, J.J., *On the K best integer network flows*, Computers and Operational Research 40 (2013), 616-626.

[30] Tarjan, R.E., *Depth-first search and linear graph algorithms*, SIAM Journal on Computing 1 (1972), 146-160.

[31] Hagerup, T., *Improved shortest paths on the word RAM*, International Colloquium on Automata, Lan-

guages, and Programming, (2000), 61-72.
[32] Warshall, S., *A theorem on Boolean matrices*, Journal of the ACM 9 (1962), 11-12.