A Scalable, Memory-Efficient Algorithm for Minimum Cycle Mean Calculation in Directed Graphs

Supriyo Maji, *Member, IEEE*, and Cheng-Kok Koh, *Senior Member, IEEE*

***Abstract*—The concept of minimum cycle mean (MCM) in a directed graph has many applications in the design of circuits and systems. The algorithm by Young, Tarjan, and Orlin (YTO), when implemented with a binary heap, has been reported to be the fastest MCM algorithm in practice even when it has higher asymptotic time complexity than Karp's algorithm. However, as an efficient implementation of YTO relies on data redundancy, its memory usage is higher and could be a prohibitive factor in large size problems. On the other hand, a typical implementation of Karp's algorithm can also be memory hungry, thereby limiting its application to only small size problems. An early termination technique from Hartmann and Orlin (HO) can be directly applied to Karp's algorithm to improve its runtime performance. The early termination also allows memory to be allocated on an on-demand basis, which can reduce the memory requirement of Karp's algorithm. In our evaluation based on graphs constructed from IWLS 2005 benchmark circuits and randomly generated graphs, we empirically observe that the HO algorithm (or Karp's algorithm with early termination technique from the HO algorithm) has much less memory usage than YTO, but it lags behind YTO in runtime performance. We propose several improvements to the early termination technique of the HO algorithm. While further improving its memory advantage over YTO, we significantly improve the runtime performance of the HO algorithm to the extent that the proposed algorithm has runtime performance that is comparable to YTO for circuit-based graphs and for dense randomly generated graphs.**

***Index Terms*—Clock network optimization, memory usage, minimum cycle mean, parametric shortest path algorithm, runtime performance, VLSI timing optimization.**

I. Introduction

VARIOUS applications in the design of circuits and systems require computation of minimum cycle mean (MCM) in a directed graph [4], [7], [8], [15]–[17], [20]. Some important applications are in the areas of clock network optimization, including clock period minimization, slack optimization, and timing analysis [3], [4], [12], [22]. MCM algorithms are also used in other graph algorithms and applications [18], [19].

Optimizing clock network is an important goal in sequential circuit design [3]. In sequential circuit, flip-flops or latches are the sequential elements. Two sequential elements can be separated by combinational logic gates which have delays. In a timing constraint graph, sequential elements can be represented by nodes and skew constraints (i.e., hold and setup time constraints) between them by directed edges (see Fig. 1). With t_i and t_j being the arrival times of the clock signal to flip-flops FF_i and FF_j, respectively, the setup and hold time constraints (see Fig. 1) can be written, respectively, as $t_i - t_j \leq w_{ji} = C_P - (t_{pFF}^{\max} + t_{\text{comb}}^{\max} + t_{\text{setup}}^{\max})$ and $t_j - t_i \leq w_{ij} = (t_{pFF}^{\min} + t_{\text{comb}}^{\min}) - t_{\text{hold}}^{\max}$, where C_p is the clock period, $t_{pFF}^{\max}$ ($t_{pFF}^{\min}$) and $t_{\text{comb}}^{\max}$ ($t_{pFF}^{\min}$) are, respectively, the maximum (minimum) propagation delays through flip-flops and combinational circuits, and $t_{\text{setup}}^{\max}$ and $t_{\text{hold}}^{\max}$ are, respectively, the maximum setup and hold times of a flip-flop. In the graph representation, w_{ji} and w_{ij} are the weights of the directed edges.

Process variations in a sequential circuit can cause setup and hold time violations [4]. A timing constraint violation is equivalent to having a negative cycle in the timing constraint graph. To make a circuit robust, sufficient tolerance (or slack) must be added to the timing paths. For example, we can include a slack or tolerance parameter, denoted as λ in a setup- and hold-time constraint as follows: $t_i - t_j \leq w_{ji} - \lambda$ and $t_j - t_i \leq w_{ij} - \lambda$. One form of the slack optimization problem is that of determining the largest λ that could be assigned to all edges simultaneously without introducing a negative cycle in the graph.

One can calculate the mean of a cycle in a timing constraint graph by dividing the sum of edge weights by the number of edges in the cycle. When an edge participates in a cycle, the (uniform) slack that all edges in that cycle could be assigned is the cycle mean. The largest slack that an edge can be assigned in the smallest among the means of all cycles in which it participates. Therefore, the largest (uniform) slack, denoted as λ^*, that could be assigned to all edges in a timing constraint graph is the minimum among the means of all cycles in a graph, or the MCM of a graph [13].

There is a more general concept of a parameterized graph [14] where only some edges are associated with the parameter λ. If edge e is not parameterized, its weight is $w(e)$; otherwise, the weight is $w(e) - \lambda$. In such a formulation, the

Manuscript received March 9, 2021; revised May 31, 2021; accepted July 11, 2021. Date of publication July 15, 2021; date of current version May 20, 2022. This work was supported in part by NSF under Grant CCF-1527562. This article was recommended by Associate Editor C.-K. Cheng. *(Corresponding author: Supriyo Maji.)*

Supriyo Maji was with the Department of Electrical and Computer Engineering, Purdue University, West Lafayette, IN 47907 USA. He is now with Cadence Design Systems, San Jose, CA 95134 USA (e-mail: supriyomj@gmail.com).

Cheng-Kok Koh is with the Department of Electrical and Computer Engineering, Purdue University, West Lafayette, IN 47907 USA (e-mail: chengkok@purdue.edu).

Digital Object Identifier 10.1109/TCAD.2021.3097300

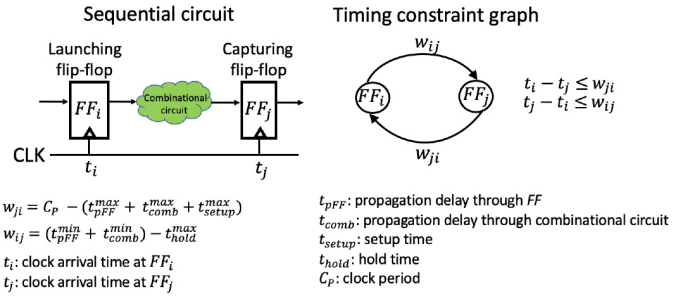

Fig. 1. Example of a sequential circuit. In a sequential circuit, flip-flops or latches are the sequential elements. Two sequential elements can be separated by combinational logic gates which have delays. In a timing constraint graph, sequential cells can be represented by nodes and skew constraints (i.e., hold and setup time constraints) between them by edges.

mean of a cycle is obtained by dividing the cycle weight by the number of parameterized edges in the cycle. For this work, we assume that all edges in a graph are parameterized, i.e., every edge is associated with the parameter λ.

Consider a weighted, directed, and strongly connected graph $G(V, E, w)$, where V is the node set, E is the edge set, and each edge e has an associated weight $w(e)$. Karp used dynamic programming to calculate MCM exactly in $\Theta(|V||E|)$ time [13] for a graph where all edges are parameterized. Karp's algorithm maintains a table of distances. In the table, row k records the shortest paths to all nodes (from an arbitrary source node) with exactly k edges, where k can range from 0 to $|V|$. The technique of relaxation is used to obtain a k-edge shortest path from a $(k-1)$-edge shortest path, which is similar to an iteration of the Bellman–Ford algorithm [5]. A typical implementation of Karp's algorithm requires a table of size $\Theta(|V|^2)$ to store distances information for the computation of MCM. Other memory overhead is also necessary when there is a need to also determine a minimum-mean cycle (a cycle whose mean is the minimum). As graph size becomes large, the quadratic memory requirement can be prohibitive.

Karp and Orlin (KO) [14] solved a series of parametric shortest-path (tree) problem for MCM calculation in $O(|V||E| \log |V|)$ time. We refer to this algorithm as KO. In this series of shortest-path trees, two consecutive trees differ by only two edges. KO uses a binary heap to keep track of the edges that could be moved into the next iteration of the shortest-path tree. Instead of keeping track of the edges, Young, Tarjan, and Orlin's algorithm [21] keeps track of the nodes for which its incoming edge could change in the next iteration. We refer to this algorithm as Young, Tarjan, and Orlin (YTO). With a Fibonacci heap, YTO improves the amortized time complexity to $O(|V||E| + |V|^2 \log |V|)$. However, YTO implemented with a binary heap has been reported to be the fastest MCM algorithm in practice [3], [4], [7], [10], [12] even when it has higher asymptotic time complexity (i.e., $O(|V||E| \log |V|)$) than Karp's algorithm. Such efficiency in runtime however comes at the expense of data redundancy. A state-of-the-art implementation of YTO uses both forward (outgoing) and reverse (incoming) graphs to store the input graph information [10]. Although the two structures contain the same information, they facilitate the faster update of the binary heap. As two graph structures are maintained, the memory overhead of YTO is $O(|E|)$.

The $O(|E|)$ overhead of YTO appears leaner than the $\Theta(|V|^2)$ overhead of Karp's algorithm. However, the overhead of Karp's algorithm could be reduced by the early termination technique in the Hartmann and Orlin's (HO) algorithm [11], which we refer to as HO. While HO can also handle graphs where some edges are not parameterized, we focus on its early termination technique that can improve the runtime performance and reduce the memory overhead of Karp's algorithm. Instead of always filling in all $|V|+1$ rows of the table of distances as in Karp's algorithm, HO may terminate earlier by checking the feasibility of constraints in a dual formulation of the MCM problem. In addition to improving the runtime performance, early termination also allows the program to allocate memory on-demand, allocating new rows in the distance table only when it could not terminate earlier. The memory usage of Karp's algorithm can be reduced from $\Theta(|V|^2)$ to $O(K|V|)$, where K is the total number of rows explored.

In our evaluation based on graphs constructed from IWLS 2005 benchmark circuits [2] and randomly generated graphs, we observe that HO (or Karp's algorithm with early termination technique from the HO algorithm) has much less memory usage than YTO, but lags behind YTO in runtime performance. While retaining the memory advantage of HO, we propose improvements to the early termination technique to boost the overall runtime performance of HO.

There are three steps in the early termination technique of HO: 1) detection of cycles; 2) calculation of dual vector π; and 3) checking of feasibility condition. At row k, $0 \leq k \leq |V|$, the detection of cycles requires traversal of up to $|V|$ k-edge shortest paths with a time complexity of $O(k|V|)$. Among all detected cycles, the cycle with the minimum mean is used to calculate the dual vector π in $O(k|V|)$ time complexity (see Section V for details). The checking of feasibility condition is equivalent to one iteration of the Bellman–Ford algorithm, which requires $O(|E|)$ time complexity. We propose filtering techniques that reduce the number of k-edge shortest paths to be traversed for cycle detection. This allows a best-case $O(|V|)$ time complexity of cycle detection when none of the k-edge shortest paths have to be considered while still maintaining the same worst-case time complexity of $O(k|V|)$. Moreover, we propose a filtering technique to improve the runtime efficiency of calculating the dual vector π, achieving again $O(|V|)$ time complexity at best while maintaining the same worst-case time complexity of $O(k|V|)$. Furthermore, the proposed algorithm can carry out the checking of feasibility condition in $O(|V|)$ time complexity.

We observe that for circuit-based graphs, the proposed algorithm has better runtime performance than HO and produces comparable results to YTO. For random graphs, the proposed algorithm has better runtime performance than HO and is comparable to YTO as the graph becomes denser. The proposed algorithm has better memory efficiency compared to both HO and YTO algorithms.

II. Minimum Cycle Mean

Consider a weighted, directed graph $G(V, E, w)$ that is strongly connected. Let $w(C) = \sum_{e \in C} w(e)$ and $\tau(C)$ denote, respectively, total edge weight and total number of edges of a cycle C. The cycle mean of C, denoted as $\lambda(C)$, is defined as

follows:

$$\lambda(C) = \frac{w(C)}{\tau(C)} \tag{1}$$

which is the total edge weight of the cycle divided by the number of edges. The MCM λ^* of G is defined as $\lambda^* = \min_{C \in \mathcal{C}}(\lambda(C))$ where $\mathcal{C}$ is the set of all cycles in G. The problem of finding λ^* is called the MCM problem [13], [14].

If the graph G is not strongly connected, the MCM can be obtained by computing the smallest among the MCMs of all strongly connected components (SCCs) of the graph. In our presentation of algorithms in the following sections, we assume that the input graph to an algorithm is strongly connected.

In the following, we first review the KO and YTO algorithms, as they represent the fastest MCM algorithms. We then review Karp's algorithm and the HO algorithm, as they are similar and are the algorithms we seek to improve in this work.

III. Parametric Shortest-Path Algorithms

KO [14] proposed a parametric shortest-path algorithm for the MCM problem. Based on the observation that λ^* is the largest (edge parameter) λ for which G has no negative cycles, the algorithm starts with $\lambda = -\infty$ and computes a shortest-path tree (to all other nodes) from an arbitrary source node. In each iteration, the algorithm increments λ such that the shortest-path tree changes by only one edge. In Fig. 2, for example, for a smaller λ, node v has a shortest path through node z for the top shortest-path tree whose source node is s. As λ increases, node v acquires a new shortest path through node u, as shown in the bottom shortest-path tree. The top and bottom shortest-path trees differ by the two edges (z, v) and (u, v). The algorithm terminates when a cycle of zero weight is detected. This cycle is the minimum-mean cycle, or the cycle with the minimum mean.

Using a binary heap to keep track of the next edge to be exchanged with an existing edge in the shortest-path tree, the algorithm runs in $O(|V||E| \log |V|)$ time. YTO's algorithm [21] improves the runtime complexity to $O(|V||E| + |V|^2 \log |V|)$ (amortized) using two techniques. First, a Fibonacci heap implementation is used to improve the amortized time complexity of heap operations.

Second, instead of keeping track of the edges to be moved into the next iteration of the shortest-path tree, the heap keeps track of the nodes for which its incoming edge could change in the next iteration. In Fig. 2, for example, as the shortest path to node v has changed, all nodes in the (shortest path) subtree of v would be updated with the new shortest-path information.

Because of the changes in the shortest paths to these nodes, the algorithm has to update information for all nodes in the heap that are adjacent to the subtree. For a node on the subtree, this requires accessing all incoming edges to the subtree and for a node adjacent to the subtree, this requires accessing all outgoing edges from the subtree [6], [7], [10], [14]. In Fig. 2, edges that are adjacent to the shortest-path subtree of v are shown as red dashed arrows.

To access the edges that are adjacent to the nodes in the subtree of v efficiently, state-of-the-art implementations of YTO use both forward (outgoing) and reverse (incoming) graphs to store the input graph information [7], [10]. As a consequence, the memory overhead of the YTO algorithm is $O(|E|)$. The reader may refer [7], [10], [21] for more details of the YTO algorithm and its implementation. YTO using binary heap has been reported to be the fastest MCM algorithm [3], [4], [7], [10], [12]. Our implementation of YTO in this work follows the pseudocode presented in [7].

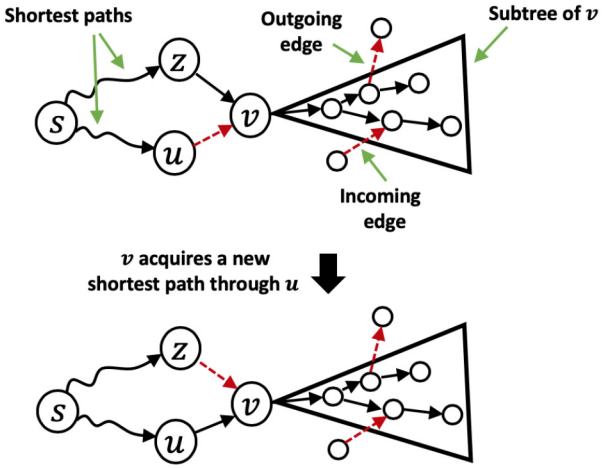

Fig. 2. As λ increases, the shortest path from source node s to node v changes the parent node of v from z to u. The shortest paths to all nodes in the subtree of v are therefore updated and the algorithm has to update information for all nodes in the heap that are adjacent to the subtree. For a node on the subtree, this requires accessing all incoming edges to the subtree and for a node adjacent to the subtree, this requires accessing all outgoing edges from the subtree [6], [7], [10], [14]. All adjacent edges to the subtree are shown as red, dashed arrows.

IV. Karp's Algorithm

Karp's dynamic programming algorithm [13] is an exact algorithm with exact complexity bound to solve the MCM problem in a directed graph. Given a graph $G(V, E, w)$, any $|V|$-edge shortest path, i.e., a path containing exactly $|V|$ edges, must contain a cycle in it. Karp proved that a minimum-mean cycle (C) must be present in one of the $|V|$-edge shortest paths from an arbitrary source. Note that C may not be a simple cycle. In fact, if C is of length $|V| - k$, with $0 \le k \le |V|-1$, there must be a node, say v on C, whose shortest path from a source node (s) has exactly k edges, and the $|V|$-edge shortest path would include C (starting and ending with v), as illustrated in Fig. 3. With this, Karp characterized MCM (λ^*) as

$$\lambda^* = \min_{v \in V} \max_{0 \le k \le |V|-1} \left[\frac{D_{|V|}(v) - D_k(v)}{|V| - k} \right] \tag{2}$$

where $D_k(v)$ is the weight of k-edge shortest path from the source node s (arbitrarily chosen) to v. If no such path exists, $D_k(v) = \infty$.

Karp's algorithm uses the following recurrence to compute the entries in a table of distances (D-table):

$$D_k(v) = \min_{(u,v) \in E} [D_{k-1}(u) + w(u, v)] \tag{3}$$

for $k = 1, 2, \ldots, |V|$, with the initial conditions that $D_0(s) = 0$ and $D_0(v) = \infty$ for $v \in V - \{s\}$. We refer to this as a

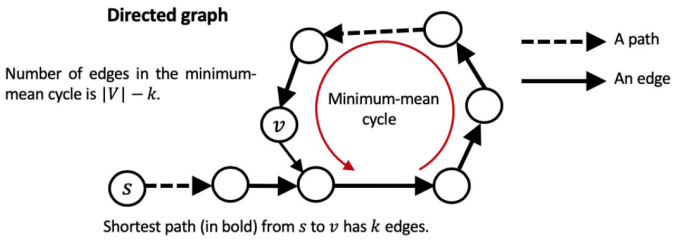

Fig. 3. Minimum-mean cycle in a graph has $|V| - k$ edges. Node v on the cycle has a shortest path (in bold) which contains k edges. The $|V|$-edge shortest path includes the cycle.

Algorithm 1: Vertical Relaxation

```
1  vertical_relaxation(G, D, P, k)
2    for each edge (u, v) ∈ E do
3      if D_k[v] > D_{k−1}[u] + w(u, v) then
4        D_k[v] = D_{k−1}[u] + w(u, v)
5        P_k[v] = u
6      end if
7    end for
8  end vertical_relaxation
```

Algorithm 2: Karp's Algorithm

```
   Input  : Directed graph G(V, E, w), a strongly connected component
            (SCC)
   Output: Minimum cycle mean λ*
1    /* Initialization */
2    λ* = ∞ /* global variable to store minimum cycle mean */
3    for k = 0 to |V| do
4      for each node v ∈ V do
5        D_k[v] = ∞
6        P_k[v] = −1
7      end for
8    end for
9    D_0[s] = 0
10   /* D- and P-tables computation*/
11   for k = 1 to |V| do
12     vertical_relaxation(G, D, P, k)
13   end for
14   /* MCM computation */
15   for each node v ∈ V do
16     if D_{|V|}[v] ≠ ∞ then
17       λ_v = −∞
18       for k = 0 to |V| − 1 do
19         λ_v = max(λ_v, (D_{|V|}[v] − D_k[v])/(|V| − k))
20       end for
21       λ* = min(λ*, λ_v)
22     end if
23   end for
24   return λ*
```

vertical relaxation (see the pseudocode below) because it uses the $(k-1)$th row to compute the kth row of the D-table. As we are typically also interested in retrieving the nodes in a cycle, a P-table is also used to keep track of the parent of a node in all shortest paths.

Karp's algorithm is presented in the pseudocode below. We first initialize D- and P-tables. To compute entries in the tables, we perform $|V|$ iterations of vertical relaxation. After that, we compute the MCM based on (2).

An iteration of vertical relaxation takes $\Theta(|E|)$ time. The total time complexity is therefore $\Theta(|V||E|)$. Notwithstanding the fact that the algorithm uses only one graph structure (forward or reverse) compared to two structures (forward and reverse) in YTO, $\Theta(|V|^2)$ memory usage in Karp's algorithm for storing the D- and P-tables can be prohibitive for large graph applications. We shall now present the early termination technique from HO in the next section.

V. Early Termination of Karp's Algorithm (HO)

As we compute rows of the D-table, many cycles may appear before k reaches $|V|$ and one of them could be the minimum-mean cycle. A technique for early termination of Karp's algorithm was proposed by HO in [11] based on the following.

Let $\lambda_{\min}$ denote the smallest cycle mean among the cycles in all k-edge shortest paths. Consider a modified graph $G(V, E, w - \lambda_{\min})$, where each edge has its weight reduced by mean $\lambda_{\min}$. If $\lambda_{\min} = \lambda^*$, a minimum-mean cycle in $G(V, E, w-\lambda_{\min})$ has a weight of 0 and a nonminimum-mean cycle has a positive weight. In other words, the shortest-path distances from an arbitrary source to all nodes in the modified graph are well defined. Let $\pi[v]$.dist denote the shortest distance from the source node to v in this modified graph. Additionally, we have $\pi[v]$.parent and $\pi[v]$.length to denote, respectively, the parent node of the shortest distance and the length (i.e., the number of edges) of the shortest path from the source node to v in the same graph. We will use them when we discuss the proposed algorithm. The following constraints therefore hold:

$$\pi[v].dist \leq \pi[u].dist + w(u, v) - \lambda_{\min} \ \forall (u, v) \in E \qquad (4)$$

must be satisfied. This is the dual formulation referred to in [11], and we will refer to π as the dual vector and (4) as the dual constraints.

If $\lambda_{\min}$ is λ^* and the shortest paths to all nodes are of length no greater than k, the shortest path to any node v must be one of the j-edge shortest paths, $1 \leq j \leq k$. Therefore, $\pi[\,\cdot\,]$ can be computed using the expression

$$\pi[v].dist = \min_{1 \leq j \leq k} [D_j[v] - j\lambda_{\min}] \ \forall v \in V \qquad (5)$$

and they must satisfy the dual constraints.

On the other hand, if $\lambda_{\min} > \lambda^*$, the modified graph $G(V, E, w-\lambda_k^{\min})$ has negative cycles that would violate some dual constraints. If some shortest paths in the modified graph have path lengths greater than k, some dual constraints will also be violated. Therefore, when all dual constraints are satisfied, all shortest paths have been found, and as the shortest distances are well defined, $\lambda_{\min} = \lambda^*$. Consequently, we can terminate the iterative process of vertical relaxation in Karp's algorithm.

It should now be clear that to decide whether Karp's algorithm can terminate at row k, the HO algorithm has to calculate $\lambda_{\min}$, the smallest cycle mean among the cycles in all k-edge shortest paths, calculate π based on $\lambda_{\min}$, and check that all dual constraints are feasible. The steps for the calculation of π (5) and the feasibility check (4) are straightforward and their pseudocodes are provided below.

The calculation of π has $O(k|V|)$ time complexity. The feasibility check has $O(|E|)$ time complexity. In fact, the feasibility check of the dual constraints is equivalent to the check for negative cycles in the Bellman–Ford algorithm.

We shall now focus on the calculation of $\lambda_{\min}$. This calculation requires the detection of cycles in the k-edge shortest

Algorithm 3: π Calculation

```
1  π_calculation(D, k, λ_min)
2    for each node v∈V do
3      π[v].dist = ∞
4      for j = 0 to k do
5        if π[v].dist > D_j[v] − j ∗ λ_min then
6          π[v].dist = D_j[v] − j ∗ λ_min
7          π[v].parent = P_j[v]
8          π[v].length = j
9        end if
10     end for
11   end for
12   return π
13 end π_calculation
```

Algorithm 4: Feasibility Check

```
1  feasibility_check(π, λ_min)
2    for each edge e = (u, v) ∈ E do
3      if π[v].dist > π[u].dist + w(u, v) − λ_min then
4        return false
5      end if
6    end for
7    return true
8  end feasibility_check
```

paths. Given a k-edge shortest path, we can detect the cycles in the path using a forward traversal (from the source node) or backward (from a destination node) [11]. HO suggested several techniques for making forward traversal efficient. It was argued that a forward traversal for a path can be truncated as soon as a cycle is encountered in that path. Furthermore, to disallow repeated visits of a shortest path that does not extend to a shortest path of longer path length, they pruned the path from future exploration after checking for early termination. We refer to the version of HO that uses forward traversal for cycle detection (and the calculation of $\lambda_{\min}$) as HO/f.

In this work, we adapt the backward traversal method in [6] and [9] to find a cycle along a path. When we discuss the proposed method we will explain how our cycle finding strategy also uses some concepts of forward traversal presented in [11] to eliminate the need for path traversal for cycle detection. The pseudocode is provided below. Here, the variable *level_array* stores the cycle information during traversal. This array is initialized at the beginning of the main MCM algorithm (see pseudocode *HO Algorithm* at the end of the section). An additional array *level_stack* makes the reinitialization of *level_array* efficient.

The pseudocode **$\lambda_{\min}$_***calculation_in_a_path*** is called by the following pseudocode to compute the $\lambda_{\min}$ at row k.

We are now ready to present the pseudocode for early termination. To decide whether Karp's algorithm can terminate at row k, we have to perform $\lambda_{\min}$ calculation (for all k-edge shortest paths), π calculation, and feasibility check.

Both $\lambda_{\min}$ calculation and π calculation have $O(k|V|)$ time complexity and the feasibility check has $O(|E|)$ time complexity. If we perform these three steps at every row, the worst-case complexity of Karp's algorithm with early termination is $O(|V|^3 + |V||E|)$. However, with early termination performed at every power of two, as shown in the pseudocode, the time complexity is $O(|V||E|)$ for both HO/f and HO/b [11].

Algorithm 5: $\lambda_{\min}$ Calculation in a Path

```
1  λ_min_calculation_in_a_path(D, P, k, v_start, λ_min)
2    length = 0
3    v = v_start
4    j = k
5    while j ≥ 0 do
6      if level_array[v] > −1 then
7        λ = (D_level_array[v][v] − D_j[v])/(level_array[v] − j)
8        λ_min = min(λ_min, λ)
9      end if
10     level_array[v] = j
11     level_stack[length] = v
12     + + length
13     v = P_j[v]
14     − − j
15   end while
16   for j = length − 1 to 0 do
17     level_array[level_stack[j]] = −1
18   end for
19   return λ_min
20 end λ_min_calculation_in_a_path
```

Algorithm 6: $\lambda_{\min}$ Calculation

```
1  λ_min_calculation(D, P, k)
2    λ_min = ∞
3    for each node v∈V do
4      if D_k[v] ≠ ∞ then
5        λ_min = λ_min_calculation_in_a_path(D, P, k, v, λ_min)
6      end if
7    end for
8    return λ_min
9  end λ_min_calculation
```

Algorithm 7: Early Termination

```
1  early_termination(D, P, k, |V|)
2    if k is a power of 2 or k == |V| then
3      /* λ_min calculation */
4      λ_min = λ_min_calculation(D, P, k)
5      λ_global = λ_min /* update λ_global */
6      /* Dual vector π calculation */
7      π = π_calculation(D, k, λ_global)
8      /* Feasibility condition check */
9      return feasibility_check(π, λ_global)
10   end if
11   return false
12 end early_termination
```

Algorithm 8: Allocate and Initialize D_k and P_k

```
1  allocate_and_initialize_D_k_and_P_k(k)
2    allocate memory for D_k and P_k
3    for each node v ∈ V do
4      D_k[v] = ∞
5      P_k[v] = −1
6    end for
7  end allocate_and_initialize_D_k_and_P_k
```

As HO/b is more compatible with Karp's algorithm (because of the presence of the P-table), our focus in this work is to improve the HO/b algorithm, or rather the implementation of the HO/b algorithm by Dasdan *et al.* [1], [6], [9] (referred to as HO/b-D here), note that HO did not provide any pseudocode. The pseudocode of the HO/b-D algorithm is shown below.

Here, early termination allows memory to be allocated on-demand instead of a one-time allocation of $\Theta(|V|^2)$ memory at

Algorithm 9: HO/b-D Algorithm (Karp's Algorithm With Early Termination)

Input : Directed graph $G(V, E, w)$, a strongly connected component (SCC)
Output: Minimum cycle mean λ^*

```
/* Initialization */
λ_global = ∞ /* global variable to store minimum cycle mean */
for each node v ∈ V do
  level_array[v] = −1 /* global array for λ_min calculation */
end for
allocate_and_initialize_D_k_and_P_k(0)
D_0[s] = 0
/* D- and P-tables computation*/
for k = 1 to |V| do
  allocate_and_initialize_D_k_and_P_k(k)
  vertical_relaxation(G, D, P, k)
  if early_termination(D, P, k, |V|) then /* λ_global updated */
    break
  end if
end for
return λ_global
```

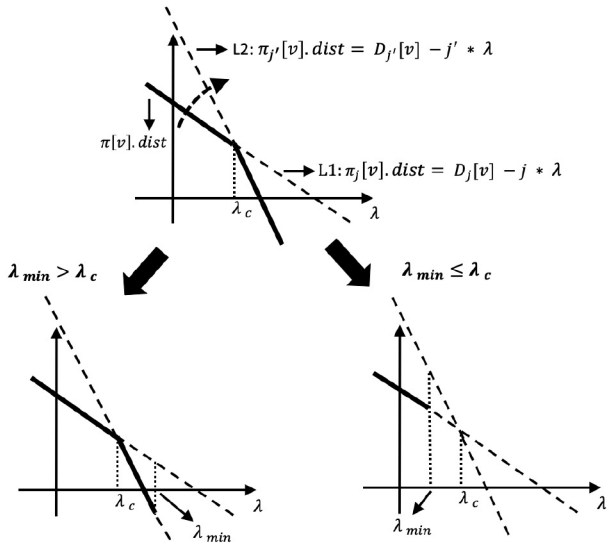

Fig. 4. Filtering of redundant k-edge shortest paths for efficient π calculation. $L1$ and $L2$ represent the j- and j'-edge shortest paths, respectively, with $j < j'$. If $\lambda_{\min} \leq \lambda_c$, where λ_c is the intersection of $L1$ and $L2$, the j'-edge path is redundant and can be pruned.

the beginning, as is the case with Karp's algorithm. With early termination, memory allocation for additional rows of D- and P-table is required only when additional vertical relaxations have to take place. Such on-demand allocation strategy reduces the memory usage of Karp's algorithm for D- and P-tables from $\Theta(|V|^2)$ to $O(K|V|)$, where K is the total number of rows explored. The HO/b-D algorithm and Karp's algorithm have the same memory usage for storing the graph using a single graph structure (forward or reverse).

VI. Proposed MCM Algorithm

In order to keep the same theoretical time complexity as Karp's algorithm in the worst case, HO [11] suggested that early termination must be checked when k (i.e., number of rows computed) is a power of two (say 2^n with n being integer). But such an implementation would suffer if the actual number of rows, sufficient to calculate MCM value falls close to but larger than a power of two (2^{n-1}). The reason is that the algorithm would continue to perform vertical relaxation row after row until it reaches the next power of two (2^n) even when the minimum-mean cycle has already appeared. The cost of such relaxation could go up significantly, or for that matter saving on runtime as well as memory usage could be large if the algorithm is terminated as soon as a sufficient number of rows have been computed. This is a bottleneck of HO's algorithm for large graphs since the cost of vertical relaxation is $O(|E|)$ for every row. However, we could not perform an early termination check at every row as the cost of an early termination check at row k is $O(k|V| + |E|)$. Although there would be saving on vertical relaxation, the cost on early termination would increase, negating any benefits of performing fewer vertical relaxations. We address the issue by proposing several improvements to early termination such that the early termination check can be performed at best in $O(|V|)$ time. In the worst case, the cost of an early termination check at row k is $O(k|V|)$. However, experimental results show that it is rare that the proposed early termination check incurs $O(k|V|)$ worst-case time complexity.

The efficiency of the proposed early termination technique stems from improvements to π calculation, feasibility check, and $\lambda_{\min}$ calculation. We shall elaborate on these improvements in the remainder of this section.

A. Efficient π Calculation

Recall that $\lambda_{\min}$ is the smallest cycle mean at row k and λ_{global} is the global variable that stores the incumbent smallest cycle mean. When we obtain a $\lambda_{\min}$ that is less than λ_{global} after vertical relaxation at row k, it is necessary to recalculate the dual vector π in the modified graph $G(V, E, w - \lambda_{\min})$. At row k, for a particular node v, we calculate for each j-edge shortest path, $0 \leq j \leq k$, $\pi_j[v].dist = D_j[v].dist - j\lambda_{\min}$, and pick the minimum among them to be $\pi[v].dist$ [see (5)].

However, not all j-edge shortest paths have the potential to become the real shortest path in the modified graph. In particular, as $\lambda_{\min}$ decreases toward λ^*, many of these paths can never be the shortest path for lower $\lambda_{\min}$. We illustrate that in Fig. 4 where line $L1$ corresponds to $\pi_j[v].dist$ of the j-edge shortest path to v and line $L2$ corresponds to the $\pi_{j'}[v].dist$ of the j'-edge shortest path to v, with $j' > j$. The bold contour shows the dual vector $\pi[v].dist = \min(\pi_j[v].dist, \pi_{j'}[v].dist)$ as λ varies and λ_c is the intersection of $L1$ and $L2$. If $\lambda_{\min} \leq \lambda_c$, the j'-edge shortest path will never be the shortest path in the modified graph, therefore, we do not have to keep it. However, it is necessary to keep both j- and j'-edge shortest paths if $\lambda_{\min} > \lambda_c$.

Of course, there is no need to compute the intersection point λ_c. If $\pi_j[v].dist \leq \pi_{j'}[v].dist$, the j'-edge shortest path will not be the shortest path in any of the future modified graphs. We call it redundant and discard it. This suggests an approach of scanning the j-edge shortest path in increasing order of j to filter out paths that will never be the shortest path in any of the future modified graphs, while keeping track of the index of the incumbent shortest path, as shown in the pseudocode *efficient_π_calculation*. Note that the index of a j-edge shortest path is j, the path length.

In the pseudocode, $\pi\_edge[\,\cdot\,][v]$ stores the indices of the irredundant paths and $\pi\_stack[v]$ stores the highest index of

Algorithm 10: Efficient π Calculation

```
1  efficient_π_calculation(D, k, λmin)
2    for each node v∈V do
3      ++π_stack[v]
4      π_edge[π_stack[v]][v] = k
5      π[v].dist = ∞
6      index_min = −1
7      for i = 0 to π_stack[v] do
8        j = π_edge[i][v]
9        if π[v].dist > Dj[v] − j ∗ λmin then
10         π[v].dist = Dj[v] − j ∗ λmin
11         π[v].parent = Pj[v]
12         π[v].length = j
13         index_min++
14         π_edge[index_min][v] = j
15       end if
16     end for
17     π_stack[v] = index_min
18   end for
19   return π
20 end efficient_π_calculation
```

Algorithm 11: Efficient Feasibility Check

```
1  efficient_feasibility_check(π, D, k, λmin)
2    early_termination_flag = true
3    for each node v ∈ V do
4      if π[v].dist > Dk[v] − k ∗ λmin then
5        π[v].dist = Dk[v] − k ∗ λmin
6        π[v].parent = Pk[v]
7        π[v].length = k
8        ++π_stack[v]
9        π_edge[π_stack[v]][v] = k
10       early_termination_flag = false
11     end if
12   end for
13   return early_termination_flag
14 end efficient_feasibility_check
```

valid $\pi\_edge[\,\cdot\,][v]$, with the assumption that $\pi\_stack[v]$, $v \in V$, has been assigned to -1 at the beginning of the MCM algorithm to indicate there are no valid paths initially. In lines 3–4, to account for the new k-edge shortest path for node v, $\pi\_stack[v]$ must be incremented, and $\pi\_edge[\pi\_stack[v]][v]$ must store the index of the new path, i.e., k.

The variable *index_*min indirectly stores the index of the incumbent shortest path in that $\pi\_edge[index\_\min][v]$ stores the actual index. When we find a smaller $\pi[v].dist$, we record the index of the shortest path that accounts for that (lines 9–15). In other words, as we scan the current irredundant paths in the order of increasing path length, any paths that result in a smaller $\pi[v].dist$ are kept as irredundant paths for the calculation of π in the future.

If the filtering is effective, each node in V has only a small number of irredundant paths essential for π calculation, and at best the complexity is $O(|V|)$. In the worst case, the filtering is ineffective and at row k, each node in V has $O(k)$ irredundant paths. Therefore, the worst-case time complexity is still $O(k|V|)$ at row k.

B. Efficient Feasibility Check With Integrated Update of π

Let us re-examine the dual constraints. Equation (4) effectively "asks" whether node v can be reached from u with a shorter distance in the modified graph $G(V, E, w - \lambda_{\min})$, where $\lambda_{\min}$ is the smallest cycle mean at row k. Since $\pi[v]$ and $\pi[u]$ are calculated using j-edge shortest path, $0 \le j \le k$, the question becomes that of asking whether we can get a smaller $\pi[v].dist$ from a $(k+1)$-edge shortest path.

We can obtain a $(k+1)$-edge shortest path by performing a vertical relaxation to fill in $D_{k+1}[\,\cdot\,]$. Assuming that $\lambda_{\min}$ is still the smallest cycle mean at row $k+1$, and (4) is equivalent to

$$\pi[v].dist \le D_{k+1}[v] - (k+1)\lambda_{\min} \ \forall v \in V. \tag{6}$$

It should be obvious that the equivalent dual constraints take only $O(|V|)$ to evaluate instead of $O(|E|)$.

One may argue that the vertical relaxation still takes $O(|E|)$ and therefore there is no saving. However, if an early termination check fails, we would have to perform vertical relaxation for the next row in any case. Therefore, the only wasteful $O(|E|)$ efforts are in the last early termination check that succeeds. All earlier $O(|E|)$ efforts per row account for every necessary vertical relaxation.

We shall now present the pseudocode for efficient feasibility check. Assume that we have just completed the vertical relaxation at row k and calculated the corresponding $\lambda_{\min}$. There are two possible scenarios: $\lambda_{\min} = \lambda_{\text{global}}$ and $\lambda_{\min} < \lambda_{\text{global}}$, where λ_{global} is a global variable to store the incumbent smallest cycle mean. The *efficient_feasibility_check* pseudocode is called for the first scenario, assuming that $\lambda_{\min} \neq \infty$.

At this point, the dual vector π stores the shortest distances to all nodes with path length $< k$. Since $\lambda_{\min}$ at row $k-1$ and row k are the same, we can apply the equivalent dual constraints in lines 4–9. Note that the right-hand side of (6) is similar to the term that appears on the right-hand side of (5). We can therefore use the right-hand side of (6) to update the dual vector π (line 5) when there is a violation. Moreover, as the k-edge shortest path has become the shortest path in the modified graph $G(V, E, w - \lambda_{\min})$, we have to update $\pi\_stack[v]$ and $\pi\_edge[\pi\_stack[v]][v]$ accordingly in lines 6–7. As we have to maintain the correctness of π for the next feasibility check at row $k+1$, we have to iterate through all nodes in V and cannot return *false* immediately when there is a violation in the equivalent dual constraints, unlike the pseudocode *feasibility_check* where we immediately return *false* when there is a violation of dual constraints.

In the second scenario, the dual vector π has to be calculated with the new $\lambda_{\min}$; this is covered in the preceding section. Recall that in the HO algorithm, a feasibility check is performed after the calculation of π. In a sense, the calculation of π is based on rows 0 through k, and the feasibility check is based on the existence of shortest paths of length $k+1$. Therefore, in the proposed MCM algorithm, we do not perform a feasibility check after the calculation of π immediately. Rather, we will do that after the next vertical relaxation.

Even though we have not presented improvements to efficiently calculate $\lambda_{\min}$, we have the necessary details to present the pseudocode for efficient early termination, which is provided below.

This is called after a vertical relaxation to fill in row k of D- and P-tables. The smallest cycle mean for all cycles on k-edge shortest paths are computed using the pseudocode

Algorithm 12: Efficient Early Termination

```
1  efficient_early_termination(D, P, k, λ_global)
2    /* λ_min calculation */
3    λ_min = efficient_λ_min_calculation(D, P, k, λ_global)
4    if λ_global > λ_min then
5      λ_global = λ_min /* update λ_global */
6      /* Dual vector π calculation */
7      π = efficient_π_calculation(D, k, λ_global)
8    else if λ_min ≠ ∞ then
9      /* Feasibility condition check */
10     return efficient_feasibility_check(π, D, k, λ_global)
11   end if
12   return false
13 end efficient_early_termination
```

*efficient_**λ**$_{\mathbf{min}}$_calculation*, the details of which will be provided in the next section. As mentioned earlier, there are two cases to be considered: $\lambda_{\min} = \lambda_{\text{global}}$ and $\lambda_{\min} < \lambda_{\text{global}}$. In the former, we perform an efficient feasibility check, and update π if necessary. In none of the entries of π is updated, we can terminate the MCM algorithm; the algorithm continues otherwise. In the latter, the algorithm cannot terminate. It has to calculate π based on the smaller $\lambda_{\min}$ and filter out redundant shortest paths.

C. Efficient λ_{min} Calculation

HO [11] argued that a minimum-mean cycle can be detected if we follow the predecessor chain back from a k-edge shortest path for every node $v \in V$ until the path contains a cycle. They also argued that there exists a predecessor chain where every cycle contained in that chain is a minimum mean cycle. We make a simple argument that there exists at least one path that traverses only the cycles (not necessarily identical) with a minimum mean.

Suppose there is a node on the minimum-mean cycle with the shortest path to that node defined in the modified graph [i.e., $G(V, E, w-\lambda^*)$]. Since in the modified graph minimum-mean cycle is a zero-weight cycle, any number of repetitions of that cycle by extending the shortest path does not add additional weight. Therefore, this extended path is also a shortest path of some edges, say k. For example, in Fig. 5, the extended path (green edges) of the shortest path to node v is also a shortest path of some edges to v. This path also remains a k-edge shortest path in the original graph since adding λ^* to every edge increases the total distance of all k-edge paths by the same amount (i.e., $k * \lambda^*$). Therefore, there must be a k-edge shortest path that only has minimum-mean cycles in it.

This essentially means that the backward traversal for a node can be terminated as soon as a cycle is detected. Therefore, it is only necessary in the backward traversal of a path to detect the first simple cycle. We call this incomplete backward traversal. This helps to cut down on the traversal length. We can perform an incomplete backward traversal that terminates when the first cycle is detected, with a *break* statement between lines 8 and 9 in the pseudocode *$\boldsymbol{\lambda}_{\mathbf{min}}$_calculation_in_a_path*. The implementation in [1], [6], and [9] of HO's algorithm does not have a break statement. In other words, the version from [1], [6], and [9] detects all the simple cycles along a k-edge shortest path.

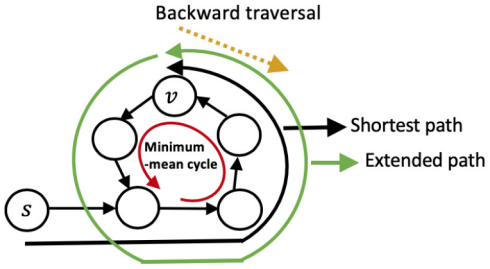

Fig. 5. Minimum-mean cycle containing node v occurs in the extended path beyond the shortest path to v in the modified graph.

Although incomplete traversal is useful, for $\lambda_{\min}$ calculation at row k, the bottleneck could still be in the detection of cycles in all k-edge shortest paths. We show that traversals for some of those k-edge shortest paths are actually redundant. Therefore, there is no need to consider them.

1) Filtering Based on Karp's Characterization of MCM: The first improvement is based on the characterization of MCM in [13]. Recall that the HO algorithm starts with a λ that is very large and progressively moves downward to converge to λ^*. The algorithm goes through many cycles whose means are larger than λ^*. Therefore, an actual minimum-mean cycle in $G(V, E, w)$ is a negative cycle in the modified graph $G(V, E, w-\lambda_{\text{global}})$ for $\lambda_{\text{global}} > \lambda^*$. It becomes a zero weight cycle when λ_{global} reaches λ^*. Before that happens, for any node (say, v) on the minimum-mean cycle in the original graph, we can reduce the shortest-path weight to the node by going through the corresponding negative cycle in the modified graph. Therefore, the node should have a parent node also residing on the cycle when the length of the path is long enough, i.e., when the k-edge shortest path has a large enough k to contain the minimum-mean cycle. This is illustrated in Fig. 5. Therefore, the parent node of v at row k in the P-table must be the same as the parent node of v in the shortest path from s to v in $G(V, E, w - \lambda_{\text{global}})$. Any k-edge shortest path that does not satisfy this condition can be ignored.

To perform this filtering, we must know the shortest path from s to every other node in $G(V, E, w-\lambda_{\text{global}})$. Remember this information is already available in $\pi[v]$ (in *dist*, parent, and length as we defined them earlier). Therefore, we will check for the condition to decide whether we should perform a backward traversal for cycle detection

$$P_k[v] = \pi[v].\text{parent}. \tag{7}$$

2) Filtering Based on λ_{global}: Another criterion for considering a k-edge shortest path for traversal is whether the path can potentially yield a cycle mean that is an MCM. Based on the available shortest-path information in $\pi[v]$, the mean of this possible cycle C is $w(C)/\tau(C) = (D_k[v] - D_{\pi[v].\text{length}}[v])/(k-\pi[v].\text{length})$. If we have the smallest cycle mean computed so far (up to row k) stored in the variable λ_{global}, we would traverse this k-edge shortest path only if the mean of the potential cycle is less than λ_{global}

$$(D_k[v] - D_{\pi[v].\text{length}}[v])/(k - \pi[v].\text{length}) < \lambda_{\text{global}}. \tag{8}$$

3) Filtering Using Efficient Forward Traversal Strategy From HO: Some of the paths where cycles have been found

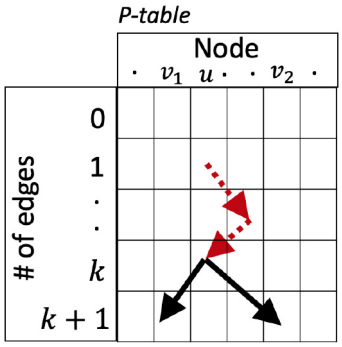

Fig. 6. There is no need to detect cycles in an extension of a path that has a cycle detected earlier. At row k, we detect a cycle (red dotted line) by traversing for node u. We do not have to consider cycle detection and $\lambda_{\min}$ calculation at nodes v_1 and v_2.

earlier may get extended. Therefore, the same cycles may be detected over and over again. First, it is important to not detect the same cycles as such detections do not help to terminate the MCM algorithm early. Second, the extensions may result in new cycles along the path. However, these new cycles also do not help to terminate the MCM algorithm early as follows.

Suppose the cycle detected earlier is a minimum-mean cycle, the new cycles that we would detect could also be minimum-mean cycles. They will result in the same $\lambda_{\min} = \lambda^*$ $(= \lambda_{\text{global}})$, where λ_{global} is the global variable to store the incumbent smallest cycle mean. The only reason that we have not terminated the MCM is that we have not discovered all shortest paths. If the cycle detected earlier is not a minimum-mean cycle, there is no reason to consider traversing the path since no cycle with minimum-mean value can occur in at least one such path. This is based on HO's argument to make forward traversal more efficient that there exists a predecessor chain where all cycles are minimum-mean cycles.

We can conclude that there is no need to detect cycles along paths that are extension of paths that contain cycles detected in the earlier backward traversals of paths. However, if the traversal of a path has yet to detect a cycle, we must consider its extended path for cycle detection and $\lambda_{\min}$ calculation.

The example in Fig. 6 illustrates this filtering technique. At row k, we detect a cycle (red dotted line) by traversing the k-edge shortest path that ends at node u. That k-edge shortest path to u is then extended to $(k+1)$-edge shortest paths, arriving at nodes v_1 and v_2. We do not have to detect cycles for these two $(k+1)$-edge shortest paths when we are at row $k+1$. In fact, all future paths $(> k+1)$ that are extended from the path to u do not have to be considered for cycle detection and $\lambda_{\min}$ calculation.

To implement the concept, we introduce flags $\text{detected}_k[v]$, $v \in V$, where the variables indicate whether cycles have been detected at a path that ends at v at row k or earlier. If *true* is stored, a cycle has been detected earlier; *false* otherwise. Therefore, we perform a backward traversal to detect cycles at row k for node v only if

$$\text{detected}_k[v] = \text{false}. \tag{9}$$

The filtering of k-edge shortest paths based on (7)–(9) is performed in line 5 of the pseudocode *efficient_*$\boldsymbol{\lambda}_{\min}$*_calculation*.

Algorithm 13: Efficient $\lambda_{\min}$ Calculation

```
1  efficient_λmin_calculation(D, P, k, λglobal)
2    λmin = λglobal
3    for each node v∈V do
4      detected_k[v] = detected_{k−1}[P_k[v]]
5      if P_k[v] == π[v].parent and (D_k[v] − D_{π[v].length}[v])/(k −
       π[v].length) < λmin and detected_k[v] == false then
6        λmin = λmin_calculation_in_a_path(D, P, k, v, λmin)
7      end if
8    end for
9    swap detected_{k−1} and detected_k
10   return λmin
11 end efficient_λmin_calculation
```

Note that $\text{detected}_k[v]$, $v \in V$, takes on the flag of its parent node $P_k[v]$ in line 4 of the pseudocode. In the pseudocode $\boldsymbol{\lambda}_{\min}$*_calculation_in_a_path*, it may be necessary to update $\text{detected}_k[v]$ when a cycle is detected. Assuming that the pseudocode $\boldsymbol{\lambda}_{\min}$*_calculation_in_a_path* has access to $\text{detected}_k[v]$, it has to set $\text{detected}_k[v]$ to *true* when a cycle is detected. The following statement has to be inserted between lines 6 and 7 of the pseudocode $\boldsymbol{\lambda}_{\min}$*_calculation_in_a_path*.

$$\text{detected}_k[v] = \text{true}$$

While it is possible to create a table of detected flags, we use only two arrays $\text{detected}_{k-1}[\,\cdot\,]$ and $\text{detected}_k[\,\cdot\,]$, and exchange the two arrays at the end of the pseudocode *efficient_*$\boldsymbol{\lambda}_{\min}$*_calculation*. In other words, detected_k becomes detected_{k-1} and detected_{k-1} serves as detected_k in the next iteration.

To summarize, we have presented a backward traversal approach for efficient $\lambda_{\min}$ calculation which unifies some of the concepts of truncation presented by HO in two separate approaches of backward and forward traversals for cycle detection. At best, the time complexity of efficient $\lambda_{\min}$ calculation is $O(|V|)$ because the filtering may allow us to not traverse any k-edge shortest paths to detect cycles. Of course, in the worst case, it will take $O(k|V|)$ to calculate $\lambda_{\min}$ at row k.

D. Pseudo-Source

The concept of a pseudo-source node has been used quite extensively in the past. Examples include the solution of a system of difference constraints using the Bellman–Ford algorithm [5], and the initialization of the shortest-path tree when $\lambda = -\infty$ in YTO [21]. We first "add" a pseudo-source node to the input graph and "create" directed edges of weight 0 from this pseudo-source node to all other nodes in the graph. Here, we assume that all edge weights are non-negative so that we can directly apply the concept of the pseudo-source node; otherwise, we have to transform the edge weights to non-negative values by subtracting the most negative edge weight from all edges and at the end, add the offset to the computed MCM value.

Note that the concept of pseudo-source can also be used in Karp's algorithm. But the HO/b or even the original Karp's algorithms do not make use of that. The benefit of having a pseudo-source in HO/b can be understood from the fact that as we search for the minimum-mean cycle, the HO algorithm must compute row after row to reach a node on the cycle before it can start going along the cycle. The actual number of rows to be computed depends on how far the cycle is located from

Algorithm 14: Proposed MCM Algorithm

Input : Directed graph $G(V, E, w)$, a strongly connected components (SCC)
Output: Minimum cycle mean λ^*

```
1  /* Initialization */
2  λ_global = ∞ /* global variable to store minimum cycle mean */
3  allocate_and_initialize_D_k_and_P_k(k = 0)
4  for each node v ∈ V do
5    level_array[v] = −1 /* global array for λ_min calculation */
6    detected_{k−1}[v] = false /* global array for λ_min calculation */
7    π_stack[v] = −1 /* global array for π calculation */
8    D_0[v] = 0 /* pseudo-source initialization */
9  end for
10 /* D- and P-tables computation */
11 allocate_and_initialize_D_k_and_P_k(k = 1)
12 vertical_relaxation(G, D, P, k=1)
13 for each node v ϵ V do
14   π[v] · parent = P_{k=1}[v]
15   + + π_stack[v]
16   π_edge[π_stack[v]][v]=1
17 end for
18 for k = 2 to |V| do
19   allocate_and_initialize_D_k_and_P_k(k)
20   vertical_relaxation(G, D, P, k)
21   if efficient_early_termination(D, P, k, λ_global) then /* λ_global
     updated */
22     break
23   else if k == |V| /* early termination not successful */
24     λ_global = λ_min_calculation(D, P, k)
25   end if
26 end for
27 return λ_global
```

the source node. Pseudo-source however rids the algorithm of the computation of rows to reach a node on the cycle.

If the source node s in Fig. 5 is the "added" pseudo-source node, there will also be "directed edges" from s to other nodes in the original graph, and all these edges are the shortest paths to the respective nodes. We do not have to actually create the pseudo-source node and its edges. Instead, we simply have to initialize the 0th rows of the D- and P-tables as $D_0[v] = 0$ and $P_0[v] = -1$ for all $v \in V$, assuming that -1 is the label of the pseudo-source node.

E. Pseudocode of the Proposed MCM Algorithm

The pseudocode of the proposed MCM algorithm is presented below. Similar to HO/b-D, we first initialize λ_{global} and *level_array*. Moreover, we initialize detected$_{k-1}$ and π_stack. D_0 and P_0 are allocated and initialized, assuming the presence of a pseudo-source node. Lines 11–26 here are almost identical to lines 9–15 in the HO/b-D algorithm except three major differences. Before the first call to **efficientλ_{min}_calculation** is made in the pseudocode **efficient_early_termination**, $\pi[v].parent$ must be initialized. Therefore, vertical relaxation for $k = 1$ must be first carried out to get $P_1[v]$ before we iterate over other values of k in the loop. Second, we call **efficient_early_termination** here. Third, when early termination is not successful (algorithm has reached $k = |V|$) **λ_{min}_calculation** (not the efficient version) is called directly. It is also important to note in the pseudocode of the proposed algorithm early termination call is made at every row.

VII. Experimental Results

We have implemented three algorithms: 1) YTO; 2) HO/b-D; and 3) the proposed MCM algorithm. All algorithms are implemented in C++ and compiled with compiler level optimization (-O3) enabled. Our implementation of YTO is similar to the pseudocode reported in [7] that uses binary heap. HO did not provide any pseudocode, but there is a pseudocode of the HO/b algorithm provided in [1], [6], and [9]. By HO/b-D, we refer to our implementation of that pseudocode. These implementations are evaluated on a standalone server, which is a Linux machine with Intel Xeon CPU E5-2660 (2.60 GHz) and 66 GB of RAM.

We evaluate the performance using IWLS 2005 benchmark circuits [2] and randomly generated strongly connected directed graphs created with a technique prescribed in [7] and [20]. We will discuss the circuit graph creation in Section VII-C. To generate a random graph using the technique in [7] and [20], we first connect all nodes in a circular manner to make the graph strongly connected. Next, two nodes are randomly selected and connected by a directed edge. The process continues until the desired number of edges connections are made. For the edge weight, random values within a predefined range are generated and assigned. Unless mentioned otherwise, all reported runtimes include everything from initializing data structures, reading graphs, to get the final MCM output.

A. Benchmarking

The YTO algorithm has been implemented by various groups in the past, targeting different applications [3], [4], [7], [8], [12], [20], [22]. Dasdan *et al.* [6]–[8] did a comparative study of various MCM algorithms and reported YTO to be the fastest MCM algorithm in practice [7]. The conclusion drawn in [7] was also corroborated later by Georgiadis *et al.* [10]. These studies looked at the sparse graph examples.

Dasdan [1] provided us with a working version of the YTO algorithm. This implementation handles only graphs with integer edge weights. Even though the integer version can still handle floating-point edge weights by scaling [10], there may be some loss in accuracy. We have also observed that the implementation from [1] cannot handle very large graphs.

As we are interested in more general applications, we prefer a version that can handle floating-point edge weights. Consequently, we implemented a version of the YTO algorithm that can handle floating-point edge weights, and then customized it such that it can handle integer edge weights. We compare the customized version of our implementation against the implementation from [1]. Fig. 7 shows that our version of the implementation has better runtime performance, in terms of the total runtime and the runtime for only MCM calculation, for dense graphs in (a) and (b) and for sparse graphs in (c). The runtime for only MCM calculation does not include the time for reading in the graph and/or other initializations. Fig. 7(c) shows that our implementation can handle larger graphs than the implementation from [1]. Therefore, for the remainder of this section, all results reported under YTO are obtained using the floating-point version of our implementation.

B. Memory Usage

We first evaluate memory usage of the three algorithms: 1) YTO; 2) HO/b-D; and 3) the proposed algorithm. While

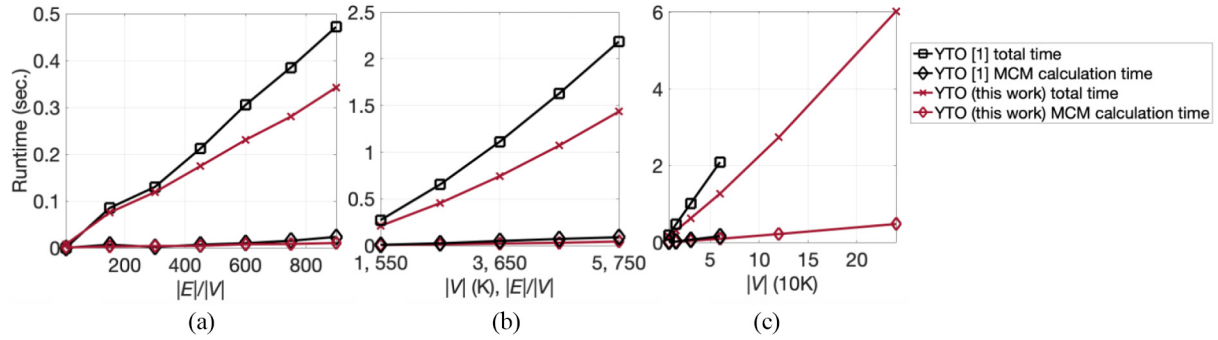

Fig. 7. Benchmarking our implementation of YTO with the implementation from [1] and [7]. The results for dense graphs are shown in (a) and (b). In (a), the graph density $|E|/|V|$ is varied from 5 to 900 for $|V|$ = 1K, and in (b), the graph size is varied by changing ($|V|$, $|E|/|V|$) from (1K, 550) to (5K, 750). The results for sparse graphs are shown in (c). In (c), $|V|$ is varied from 7.5K to 240K while $|E|/|V|$ is fixed at 50. The implementation from [1] and [7] cannot handle large graphs as shown by the missing data in the plot in (c).

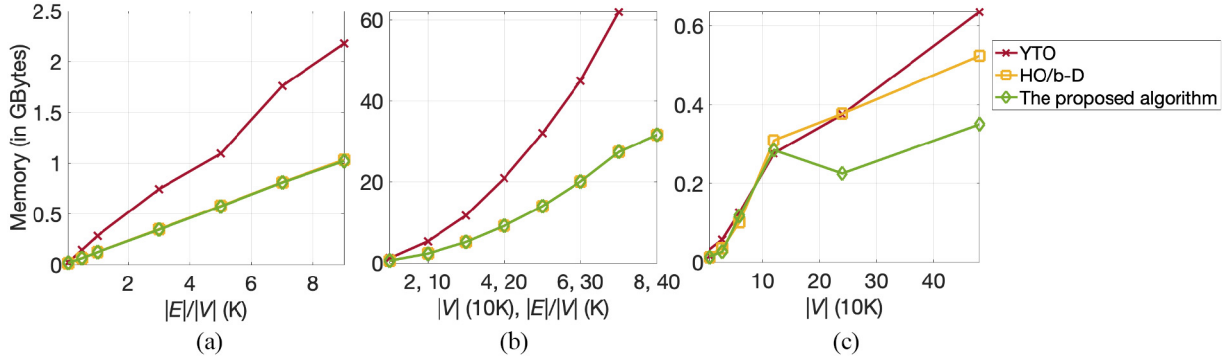

Fig. 8. Comparison of memory usage of YTO, HO/b-D, and the proposed MCM algorithm. The results for dense graphs are shown in (a) and (b). In (a), the graph density $|E|/|V|$ is increased from 50 to 9000 for $|V|$ = 10K, and in (b), the graph size is varied by changing ($|V|$, $|E|/|V|$) from (10K, 5K) to (80K, 40K). In (b), HO and the proposed algorithm have much less memory usage compared to YTO. No memory usage is reported for YTO at (80K, 40K) as the algorithm fails to run to completion. The results for sparse graphs are in (c). $|V|$ is varied from 7.5K to 480K while keeping $|E|/|V|$ fixed at 50.

Karp's algorithm uses $\Theta(|V|^2)$ memory for storing information in the D- and P-tables [13], HO/b-D and the proposed algorithm can be memory efficient if memory allocation for a row in a table is done only if the algorithm has to continue to look for more cycles. Since K, the number of rows computed in HO/b-D or the proposed algorithm, is much smaller than $|V|$ in practice, the memory usage is reduced from $\Theta(|V|^2)$ to $O(K|V|)$. Moreover, both HO/b-D and the proposed algorithm store only one graph structure (forward or reverse). Therefore, HO/b-D and the proposed algorithm can be more memory efficient than YTO, which stores both forward (outgoing) and reverse (incoming) graph structures.

A plot of the typical memory usage of the three algorithms is shown in Fig. 8. One can observe that the proposed algorithm and HO/b-D both consume much less memory compared to YTO. In fact, for very large graphs, YTO fails to compute as it runs out of memory. For this reason, there are missing data points for YTO in Figs. 8–9. For an efficient implementation, YTO requires additional memory space (beyond two graph structures) [7], as one can see in Fig. 8. We further observe (not shown here) that the implementation of YTO from [1] uses more memory than our implementation of YTO.

We must point out that as the proposed algorithm performs an early termination check at every row, it has lower memory usage than HO/b-D. While its advantage over HO/b-D in Fig. 8(a) and (b) is relatively small, the difference is evident in Fig. 8(c).

We have shown that the proposed algorithm is more memory efficient than YTO. We shall show that it does so without losing performance in runtime. In particular, we observe that the proposed algorithm has comparable runtime to YTO for graphs built from circuit examples as well as for dense random graphs.

C. Runtime Performance

To evaluate the performance using practical circuits, timing constraint graphs are constructed after synthesis of ISCAS89 and OpenCores designs from the IWLS 2005 benchmark [2]. These designs are specified in Verilog using Synopsys tool chain and a 32-nm technology library. We decompose the graph into SCCs, and apply the algorithms on these individual components. Table I provides the circuit graph information. We present runtime results for the largest SCCs in Table II. Our algorithm performs better than HO/b-D and is comparable to YTO.

We now present results on random graphs. In Fig. 9(a), the density of the graph, $|E|/|V|$, is varied while in Fig. 9(b), the graph size is varied by changing ($|V|$, $|E|/|V|$) from (10K, 5K) to (80K, 40K). Graphs in Fig. 9(a) and (b) are considered dense graphs. The proposed algorithm performs better

TABLE I
INFORMATION OF GRAPHS DERIVED FROM IWLS 2005 BENCHMARK CIRCUITS [2]

Benchmark	Circuit name	Largest SCC	
		$\|V\|$	$\|E\|$
ISCAS89	$s15850$	104	560
	$s5378$	136	1536
	$s13207$	231	1194
	$s38584$	1166	11010
	$s35932$	1728	7480
	$s38417$	1498	37478
OpenCores	$des3$	51	100
	fht	34	264
	vga_enh	206	3758
	aes_cipher	525	12526
	fpu	662	25796
	$usbf$	1431	24522
	$ac97\_ctrl$	1895	18320
	$pci\_bridge32$	1920	46484
	dma	2087	102018
	eth	8381	151484

TABLE II
RUNTIMES OF YTO, HO/B-D, AND THE PROPOSED ALGORITHM ON GRAPHS DERIVED FROM IWLS 2005 BENCHMARK CIRCUITS [2]

Circuit name	Runtime (ms)		
	YTO	HO/b-D	The proposed algorithm
$s15850$	0.958	1.174	1.209
$s5378$	2.079	2.155	2.001
$s13207$	1.837	1.879	1.985
$s38584$	9.172	17.434	11.801
$s35932$	8.039	15.675	7.249
$s38417$	24.196	32.417	23.382
$des3$	0.585	0.725	0.884
fht	0.655	0.730	0.723
vga_enh	3.593	4.317	3.974
aes_cipher	13.414	46.982	11.708
fpu	19.025	97.094	18.947
$usbf$	17.485	18.402	17.907
$ac97\_ctrl$	15.646	381.704	12.917
$pci\_bridge32$	23.378	24.051	22.247
dma	43.700	50.719	45.376
eth	61.210	59.399	58.947

than HO/b-D and is comparable to YTO. The absence of data for YTO indicates that YTO has failed to execute because of excessive memory consumption. Fig. 9(c) compares runtimes of YTO, HO/b-D, and the proposed algorithm on sparse random graphs where $|V|$ is varied from 7.5K to 480K while keeping $|E|/|V|$ fixed at 50. For sparse graphs, YTO has the best runtime performance followed by the proposed algorithm.

D. Effectiveness of Efficient Early Termination Techniques

In Table III, we show the improvements resulted from the filtering techniques in efficient $\lambda_{\min}$ calculation and π calculation. The improvements are based on the comparisons of two variants of the proposed MCM algorithm against HO/b-D. The first variant considers only the efficient $\lambda_{\min}$ calculation and the second variant considers only the efficient π calculation. All two variants perform an early termination check when k is a power of 2. On average over six sets of random graphs, efficient $\lambda_{\min}$ calculation reduces total traversal length in cycle detection by 23.49%. The total number of paths considered for computation of the dual vector π reduces by 48.80%.

TABLE III
IMPROVEMENTS (%) IN EFFICIENT $\lambda_{\min}$ CALCULATION AND IN EFFICIENT π CALCULATION FOR DENSE RANDOM GRAPHS

$\|V\|$	$\|E\|$	% reduction in traversal length in $\lambda_{\min}$ calculation	% reduction in #of paths in π calculation
10K	50M	30.533	68.401
20K	200M	20.072	48.523
30K	450M	24.960	44.931
40K	800M	19.827	42.374
50K	1250M	32.666	45.758
60K	1800M	20.275	54.172
70K	2450M	19.764	50.933
80K	3200M	19.812	35.280

In reference to the discussion in Section VI, we now present the observed time complexities of efficient $\lambda_{\min}$ calculation and efficient π calculation. For the early termination technique in HO/b-D, we consider two variants: one is to apply the early termination check when the row number is a power of two and one is to apply the early termination check at every row. The left most blocks of results in Tables IV and V are results from the first variant when applied to graphs in Fig. 8(a) and (b), respectively. Three columns of results are included in a block: 1) the number of paths considered for π calculation; 2) the traversal length for cycle detection in $\lambda_{\min}$ calculation; and 3) the number of rows computed. The middle blocks of results in Tables IV and V are results from the second variant. The right most blocks of results in Tables IV and V are results from the efficient early termination techniques from the proposed MCM algorithm.

We shall now use K' to denote the number of rows explored when we perform an early termination check when the row number is a power of two and K'' when we do that at every row. Clearly, the middle blocks and the rightmost blocks of results have the same K'' for every row. In Tables IV and V, K'' is less than K' by 32.27% and 20.93%, respectively.

For the number of paths and traversal length (after normalizing by the number of rows), we observe that in Table IV, they remain mostly constant as the number of nodes in the graph remains constant, while in Table V, the variation is comparable to that of the node size in the graph. Therefore, we can conclude that the observed complexity of efficient π calculation or traversal length for cycle detection in efficient $\lambda_{\min}$ calculation is close to $O(|V|)$.

Therefore, while both efficient $\lambda_{\min}$ calculation and efficient π calculation have worst-case time complexity $O(k|V|)$ at row k, the best-case time complexity and the observed average time complexity are $O(|V|)$. Recall that we have already established that the time complexity of efficient feasibility check is $O(|V|)$ for each row. In other words, the efficient early termination techniques proposed in this work has the appropriate time complexity for early termination check to be performed at every row to reduce the number of vertical relaxations ($O(|E|)$ for each row) and therefore reduce the overall runtime.

VIII. ANALYSIS

Note that the search for λ^* happens in opposite direction in YTO and Karp's algorithm (or HO and the proposed algorithm). YTO works by first calculating a shortest-path

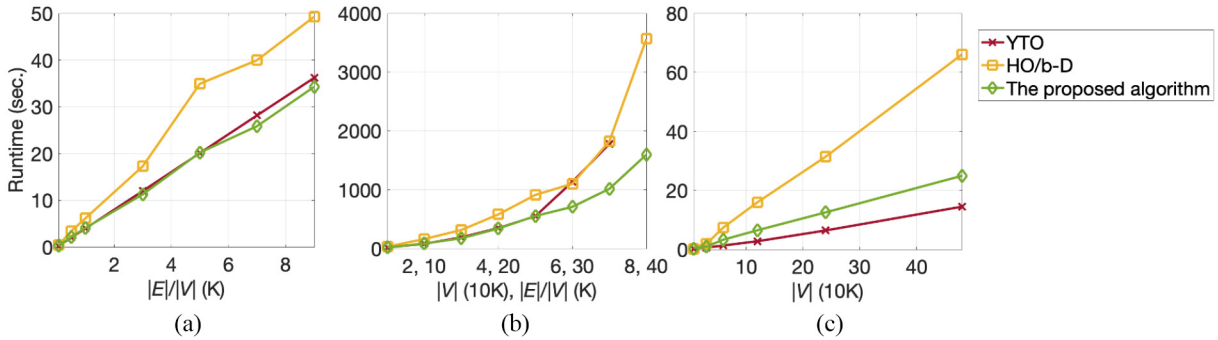

Fig. 9. Comparison of runtime performance of YTO, HO/b-D, and the proposed algorithm on random graphs. The graphs in (a) and (b) are dense. In (a), the graph density $|E|/|V|$ is varied from 50 to 9000 for $|V|$ = 10K. The proposed algorithm performs better than HO/b-D and is comparable to YTO as the graph becomes denser. In (b), the graph size is varied by changing ($|V|$, $|E|/|V|$) from (10K, 5K) to (80K, 40K). YTO fails for the graph at (80K, 40K) because of its excessive memory consumption. For sparse graphs in (c), $|V|$ is varied from 7.5K to 480K while $|E|/|V|$ is fixed at 50.

TABLE IV
COMPARING THREE VERSIONS OF EARLY TERMINATION ON THE GRAPHS FROM FIG. 8(A) FOR THE NUMBER OF PATHS CONSIDERED FOR π CALCULATION (#PATHS), THE TRAVERSAL LENGTH FOR CYCLE DETECTION IN λ_{min} CALCULATION (LENGTH), AND THE NUMBER OF ROWS COMPUTED (#ROWS)

Graph		Early termination						Efficient early termination every row w/ filtering		
		power of two			every row					
$\|V\|$	$\|E\|$	#paths	length	#rows	#paths	length	#rows	#paths	length	#rows
10K	500K	1.503M	1.458M	75.667	17.98M	17.47M	53.333	1.912M	3.872M	53.333
10K	5000K	1.380M	1.366M	70.333	15.35M	14.92M	48.833	1.697M	3.092M	49.000
10K	10000K	1.826M	1.793M	91.667	24.76M	24.16M	64.667	2.108M	4.616M	64.667
10K	30000K	1.818M	1.793M	91.667	21.25M	20.70M	61.333	2.348M	4.251M	61.333
10K	50000K	1.406M	1.384M	71.600	18.92M	18.44M	57.200	1.846M	3.750M	57.200
10K	70000K	1.818M	1.793M	91.667	17.03M	16.56M	53.667	1.811M	2.909M	53.667
10K	90000K	1.930M	1.880M	98.000	11.44M	10.92M	61.000	1.081M	1.176M	61.000

TABLE V
COMPARING THREE VERSIONS OF EARLY TERMINATION ON THE GRAPHS FROM FIG. 8(B) FOR THE NUMBER OF PATHS CONSIDERED FOR π CALCULATION (#PATHS), THE TRAVERSAL LENGTH FOR CYCLE DETECTION IN λ_{min} CALCULATION (LENGTH), AND THE NUMBER OF ROWS COMPUTED (#ROWS)

Graph		Early termination						Efficient early termination every row w/ filtering		
		power of two			every row					
$\|V\|$	$\|E\|$	#paths	length	#rows	#paths	length	#rows	#paths	length	#rows
10K	50M	2.580M	2.540M	129.000	28.40M	27.74M	74.000	4.618M	5.886M	74.000
20K	200M	2.580M	2.520M	65.000	23.32M	22.54M	47.000	3.334M	1.922M	47.000
30K	450M	3.870M	3.780M	65.000	33.54M	32.41M	46.000	4.302M	3.128M	46.000
40K	800M	10.32M	10.16M	129.000	145.9M	142.7M	84.000	19.69M	26.12M	84.000
50K	1250M	12.65M	12.70M	129.000	208.7M	204.7M	90.000	10.96M	30.54M	90.000
60K	1800M	15.48M	15.24M	129.000	165.9M	162.0M	73.000	4.521M	15.59M	73.000
70K	2450M	18.06M	17.78M	129.000	193.8M	189.0M	73.000	7.509M	26.85M	73.000
80K	3200M	20.64M	20.32M	129.000	428.0M	420.2M	102.000	21.06M	91.68M	102.000

tree in the graph when λ is $-\infty$ and then maintaining it as λ increases toward λ^*. The number of shortest-path trees explored in the course of the algorithm can be as large as $|V|^2$. For a binary heap-based implementation, the complexity is $O(|V||E|\log|V| + |V|^2\log|V|)$. The second term is really $O(T\log|V|)$, where T is the number of trees explored.

On the contrary, the HO algorithm can be terminated as early as after $K' = max(M, N)$ number of rows have been computed. Here, M is the maximum length of the shortest path in the modified graph [i.e., $G(V, E, w-\lambda^*)$] and $N = L_{sp}+L_{ep}$, where L_{sp} is the length of the shortest path and L_{ep} is the length of the extended path that contains the cycle (see Fig. 5). The value of N in the proposed algorithm is smaller than in the HO algorithm because of the use of pseudo-source (see Section VI-D) as that effectively removes the term L_{sp} from N. We define K'' to be the number of rows to be computed in the proposed algorithm. Computation of a row requires running vertical relaxation. Therefore, the total time complexity for relaxation is $O(K''(|E|))$ in the proposed algorithm. While each vertical relaxation is computationally more expensive than updating a shortest-path tree in YTO, K'' or even K' are typically much smaller than T when a graph is dense for the following reason. When a graph is dense, the longest shortest-path length is likely to be small and there are likely to be many short cycles. Therefore, K' or K'' are likely to be small. On the other hand, there are many cycles that YTO has to explore before reaching the minimum. Consequently, there is a good chance for the proposed algorithm to be comparable to YTO when a graph is dense. Note that the circuit graphs are usually sparse. However, most cycles in them are short, as each is constructed of only two nodes (two sequential elements). The fact that most cycles in circuit graphs are short gives the

proposed algorithm a chance to compete with YTO, as we have seen in the experimental results.

We experimentally observe K'' to have comparable or better scaling profile than T in YTO algorithm when the graph density and size are changed. Scaling profile of K'' and K' are comparable but average value of K'' is observed to be much smaller than K'.

IX. Conclusion

Memory usage and runtime performance of Karp's MCM algorithm can be improved with the early termination technique from HO, which we have referred to as HO/b in this work. We have experimentally observed on IWLS 2005 benchmark circuits and randomly generated graphs that the HO/b-D (i.e., Dasdan's implementation of HO/b) algorithm consumes much less memory compared to YTO. But when it comes to runtime, YTO performs better than HO/b-D. We have proposed several techniques to improve the early termination check in the HO/b-D algorithm. These improvements allow the efficient early termination check to be performed with $O(|V|)$ time complexity in the best-case scenario. Such efficiency allows the early termination check to be performed at every row to reduce the cost of vertical relaxation, which has a best-case time complexity of $O(|E|)$. Consequently, the proposed algorithm has better runtime performance than HO/b-D and produces comparable results to YTO for circuit graphs. For random graphs, the proposed algorithm has better runtime performance than HO/b-D and is comparable to YTO as the graph becomes denser, all these while improving memory usage of the HO/b-D algorithm.

Acknowledgment

The authors thank Dr. A. Dasdan for sharing the YTO algorithm code and Dr. C. Albrecht for sharing information on the version of the YTO algorithm that is generally used for solving the optimal slack distribution problem in clock network design. They are also thankful to anonymous reviewers of an earlier submission of this manuscript for providing important insights.

References

[1] Accessed: Oct. 10, 2019. [Online]. Available: https://github.com/alidasdan

[2] C. Albretch, *IWLS 2005 Benchmarks*, 2005, [Online]. Available: http://iwls.org/iwls2005/benchmark_presentation.pdf

[3] C. Albretch, B. Korte, J. Schietke, and J. Vygen, "Cycle time and slack optimization for VLSI-chips," in *Proc. Int. Conf. Comput. Aided Design*, 1999, pp. 232–238.

[4] C. Albretch, B. Korte, J. Schietke, and J. Vygen, "Maximum mean weight cycle in a digraph and minimizing cycle time of a logic chip," *Discr. Appl. Math.*, vol. 123, pp. 103–127, Nov. 2002.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction To Algorithms*. Cambridge, MA, USA: MIT Press, 2009.

[6] A. Dasdan, "An experimental study of minimum mean cycle algorithms," Dept. Inf. Comput. Science, Univ. California, Irvine, CA, USA, Rep. # 98-32, 1998.

[7] A. Dasdan, "Experimental analysis of the fastest optimum cycle ratio and mean algorithms," *ACM Trans. Design Autom. Electron. Syst.*, vol. 9, no. 4, pp. 385–418, 2004.

[8] A. Dasdan and R. K. Gupta, "Faster maximum and minimum mean cycle algorithms for system-performance analysis," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 17, no. 10, pp. 889–899, Oct. 1998.

[9] A. Dasdan, S. S. Irani, and R. K. Gupta, "Efficient algorithms for optimum cycle mean and optimum cost to time ratio problems," in *Proc. Design Autom. Conf.*, 1999, pp. 37–42.

[10] L. Georgiadis, A. V. Goldberg, R. E. Tarjan, and R. F. Wemeck, "An experimental study of minimum mean cycle algorithms," in *Proc. 11th Workshop Algorithm Eng. Exp. (ALENEX)*, 2009, pp. 1–13.

[11] M. Hartmann and J. B. Orlin, "Finding minimum cost to time ratio cycles with small integral transit times," *Networks*, vol. 23, no. 6, pp. 567–574, 1993.

[12] S. Held, B. Korte, J. Massberg, M. Ringe, and J. Vygen, "Clock scheduling and clocktree construction for high performance ASICS," in *Proc. Int. Conf. Comput. Aided Design*, 2003, pp. 232–240.

[13] R. M. Karp, "A characterization of the minimum cycle mean in a diagraph," *Discr. Mathe.*, vol. 23, no. 3, pp. 309–311, 1978.

[14] R. M. Karp and J. B. Orlin, "Parametric shortest path algorithms with an application to cyclic staffing," *Discr. Appl. Math.*, vol. 3, no. 1, pp. 37–45, 1981.

[15] R. Lu and C.-K. Koh, "Performance optimization of latency insensitive systems through buffer queue sizing of communication channels," in *Proc. Int. Conf. Comput.-Aided Design*, 2003, pp. 227–231.

[16] R. Lu and C.-K. Koh, "Performance analysis of latency-insensitive systems," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 25, no. 3, pp. 469–483, Mar. 2006.

[17] S. Maji and C.-K. Koh, "A scalable buffer queue sizing algorithm for latency insensitive systems," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, early access, Nov. 24, 2020, doi: 10.1109/TCAD.2020.3040244.

[18] J. B. Orlin and A. Sedeno-Noda, "An O(nm) time algorithm for finding the min length directed cycle in a graph," in *Proc. ACM-SIAM Symp. Discr. Algorithms*, 2017, pp. 1–24.

[19] H. Schneider and M. H. Schneider, "Max-balancing weighted directed graphs and matrix scaling," *Math. Oper. Res.*, vol. 16, no. 1, pp. 208–222, 1991.

[20] G. Wu and C. Chu, "A fast incremental cycle ratio algorithm," in *Proc. Int. Symp. Phys. Design*, 2017, pp. 75–82.

[21] N. E. Young, R. E. Tarjan, and J. B. Orlin, "Faster parametric shortest path and minimum-balance algorithms," *Networks*, vol. 21, no. 2, pp. 205–221, 1991.

[22] L. Zhang, J. S. Tsai, Y. Hu, and C. C-P. Chen, "Convergence-provable statistical timing analysis with level-sensitive latches and feedback loops," in *Proc. Asia South Pac. Design Autom. Conf.*, 2006, pp. 941–946.

Supriyo Maji (Member, IEEE) received the bachelor's degree in electronics and telecommunication engineering from the Indian Institute of Engineering Science and Technology Shibpur, Howrah, India, in 2007, the master's degree in electronics and electrical communication engineering from the Indian Institute of Technology Kharagpur, Kharagpur, India, in 2011, and the Ph.D. degree in electrical and computer engineering from Purdue University, West Lafayette, IN, USA, in 2020.

He currently works with Cadence Design Systems, San Jose, CA, USA, as a Research and Development Engineer. He worked with Synopsys (via Magma), Bengaluru, India; Qualcomm, San Diego, CA, USA; Hindustan Aeronautics Ltd. (SLRDC), Hyderabad, India; and Mindtree Ltd., Bengaluru. His current research interests include design automation of electronic circuits and systems.

Cheng-Kok Koh (Senior Member, IEEE) received the B.S. degree (First Class Hons.) and the M.S. degree in computer science from the National University of Singapore, Singapore, in 1992 and 1996, respectively, and the Ph.D. degree in computer science from the University of California at Los Angeles, Los Angeles, CA, USA, in 1998.

He is currently a Professor of Electrical and Computer Engineering with Purdue University, West Lafayette, IN, USA. His current research interests include physical design of very large-scale integration circuits and modeling and analysis of large-scale systems.

Prof. Koh was a recipient of the ACM Special Interest Group on Design Automation Meritorious Service Award, the Distinguished Service Award, the National Science Foundation CAREER Award, and the Semiconductor Research Corporation Inventor Recognition Award.