EFFICIENT MINIMUM CYCLE MEAN ALGORITHMS AND THEIR APPLICATIONS

A Dissertation
Submitted to the Faculty
of
Purdue University
by
Supriyo Maji

In Partial Fulfillment of the
Requirements for the Degree
of
Doctor of Philosophy

August 2020
Purdue University
West Lafayette, Indiana

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF DISSERTATION APPROVAL

Prof. Cheng-Kok Koh, Chair
Department of Electrical and Computer Engineering
Prof. Anand Raghunathan
Department of Electrical and Computer Engineering
Prof. Byunghoo Jung
Department of Electrical and Computer Engineering
Prof. Dan Jiao
Department of Electrical and Computer Engineering
Prof. Tim Rogers
Department of Electrical and Computer Engineering

Approved by:
Prof. Dimitri Peroulis
Head of the School Graduate Program

ACKNOWLEDGMENTS

I would like to express my deepest gratitude to my advisor Prof. Cheng-Kok Koh for all the encouragement and help that he has provided during the course of my PhD study. I have been fortunate to have an advisor who gave me the opportunity to work on the project sensing my willingness to learn and think new. His patience in building up a complete work and commitment to pursuing rigorous analysis have helped develop in me a sense of responsibility to carry out quality and original research. Under him, I never lost motivation to stay committed to PhD study all these years.

I am also thankful to him for making me aware of the resources to improve the writing skill. That not only helped me express what I wanted to, clearly and elaborately, in this dissertation but also shortened the time that I would have spent otherwise on writing. My appreciation also goes to other members of my doctoral dissertation committee, Prof. Anand Raghunathan, Prof. Byunghoo Jung, Prof. Dan Jiao, and Prof. Tim Rogers for their feedback on research and presentation of the work, and also for help to organize the examinations smoothly. My past research experience at IIT Kharagpur, Synopsys (via Magma) and Qualcomm and the interactions that I had with the co-workers there have also been instrumental in shaping my thinking and approach to new problems.

This journey would never have been possible without the support and encouragement of my mother and brother who have stayed with me through many ups and downs of life. Above all, it is my mother, without whose perseverance and dedication to my well being, I wouldn't be here today at Purdue. It is because of her I have remained steadfast in my goal to attain the highest education.

I also thank anonymous reviewers, Dr. A. Dasdan, and Dr. C. Albretch for providing key insights to improve quality of the research presented here. Finally, I

would like to acknowledge NSF (award CCF-1527562) that has funded in part the research in this dissertation.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Maji, Supriyo PhD, Purdue University, August 2020. Efficient Minimum Cycle Mean Algorithms And Their Applications. Major Professor: Cheng-Kok Koh.

Minimum cycle mean (MCM) is an important concept in directed graphs. From clock period optimization, timing analysis to layout optimization, minimum cycle mean algorithms have found widespread use in VLSI system design optimization. With transistor size scaling to 10nm and below, complexities and size of the systems have grown rapidly over the last decade. Scalability of the algorithms both in terms of their runtime and memory usage is therefore important.

Among the few classical MCM algorithms, the algorithm by Young, Tarjan, and Orlin (YTO), has been particularly popular. When implemented with a binary heap, the YTO algorithm has the best runtime performance although 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. An early termination technique from Hartmann and Orlin (HO) can be directly applied to Karp's algorithm to improve its runtime performance and memory usage. Although not as efficient as YTO in runtime, HO algorithm has much less memory usage than YTO. We propose several improvements to HO algorithm. The proposed algorithm has comparable runtime performance to YTO for circuit graphs and dense random graphs while being better than HO algorithm in memory usage.

Minimum balancing of a directed graph is an application of the minimum cycle mean algorithm. Minimum balance algorithms have been used to optimally distribute slack for mitigating process variation induced timing violation issues in clock network.

In a conventional minimum balance algorithm, the principal subroutine is that of finding MCM in a graph. In particular, the minimum balance algorithm iteratively finds the minimum cycle mean and the corresponding minimum-mean cycle, and uses the mean and cycle to update the graph by changing edge weights and reducing the graph size. The iterations terminate when the updated graph is a single node. Studies have shown that the bottleneck of the iterative process is the graph update operation as previous approaches involved updating the entire graph. We propose an improvement to the minimum balance algorithm by performing fewer changes to the edge weights in each iteration, resulting in better efficiency.

We also apply the minimum cycle mean algorithm in latency insensitive system design. Timing violations can occur in high performance communication links in system-on-chips (SoCs) in the late stages of the physical design process. To address the issues, latency insensitive systems (LISs) employ pipelining in the communication channels through insertion of the relay stations. Although the functionality of a LIS is robust with respect to the communication latencies, such insertion can degrade system throughput performance. Earlier studies have shown that the proper sizing of buffer queues after relay station insertion could eliminate such performance loss. However, solving the problem of maximum performance buffer queue sizing requires use of mixed integer linear programming (MILP) of which runtime is not scalable. We formulate the problem as a parameterized graph optimization problem where for every communication channel there is a parameterized edge with buffer counts as the edge weight. We then use minimum cycle mean algorithm to determine from which edges buffers can be removed safely without creating negative cycles. This is done iteratively in the similar style as the minimum balance algorithm. Experimental results suggest that the proposed approach is scalable. Moreover, quality of the solution is observed to be as good as that of the MILP based approach.

1. INTRODUCTION

Minimum cycle mean (MCM) is an important concept in directed graphs [1] [2]. Minimum cycle mean algorithms are used as subroutines in various graph algorithms [3] [4]. The concepts of minimum cycle mean and the cycle that has the minimum mean (called minimum-mean cycle) have parallels to important parameters in VLSI systems. From clock period optimization, timing analysis to layout optimization, minimum cycle mean algorithms have found widespread use in sequential VLSI system design optimization [5] [6] [7] [8] [9] [10].

For sequential circuits to operate without error, data must be processed in a synchronized manner. In such circuits, the synchronization of data is governed by a clock signal. Proper assignment of clock arrival times to each sequential element in a sequential circuit can help run the circuit at a higher clock frequency with optimum power budget. Deep sub-micron technology nodes already exhibit significant wire delay variation, temperature fluctuation, crosstalk and jitter on signal and clock paths, all having large impact on how much effort is needed to get the product out in time to market. Since timing violations are the main cause of yield loss in modern VLSI systems, a proper clock scheduling or assignment is important.

Minimum balancing of a directed graph is an application of the minimum cycle mean algorithm. Minimum balance algorithms have been used to optimally distribute slack for mitigating the process variation issues in sequential circuits [11] [12] [5] [13]. In conventional minimum balance algorithms [13] [14] [3], the principal subroutine is that of finding minimum cycle mean (MCM) in a graph. In particular, the minimum balance algorithm iteratively finds the minimum cycle mean and the corresponding minimum-mean cycle, and uses the mean and cycle to update the graph by changing edge weights and reducing the graph size. The iterations terminate when the updated graph is a single node. As the transistor size scales down to 10nm or below and the need for integrating more functionalities grows, there could be 100 billion transistors in system-on-chip in near future. The graph structures that represent such systems could also be enormous. Such growing complexities of the system representation pose substantial challenges in achieving scalability of the design automation tools. The efficiency of the graph algorithms both in terms of its memory usage and runtime performance is therefore important.

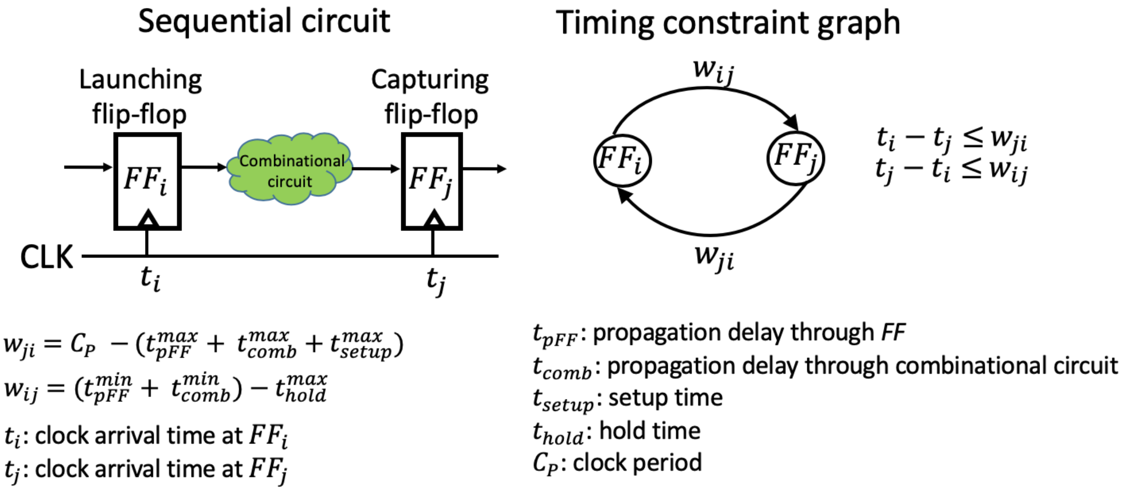

Sequential circuit
Timing constraint graph
Launching flip-flop
Capturing flip-flop
FF_i
Combinational circuit
FF_j
CLK
t_i
t_j
w_{ij}
w_{ji}
$t_i - t_j \leq w_{ji}$
$t_j - t_i \leq w_{ij}$
$w_{ji} = C_P - (t_{pFF}^{max} + t_{comb}^{max} + t_{setup}^{max})$
$w_{ij} = (t_{pFF}^{min} + t_{comb}^{min}) - t_{hold}^{max}$
t_i: clock arrival time at FF_i
t_j: clock arrival time at FF_j
t_{pFF}: propagation delay through FF
t_{comb}: propagation delay through combinational circuit
t_{setup}: setup time
t_{hold}: hold time
C_P: clock period

Fig. 1.1.: An example of a sequential circuit. In sequential circuit, flip-flops or latches are the sequential elements. Two sequential elements can be separated by combinational logic gates which have delays. For the slack distribution problem, setup and hold time constraints can be rewritten respectively as $t_i - t_j \leq w_{ji} - \lambda_{ji}$ and $t_j - t_i \leq w_{ij} - \lambda_{ij}$ to include the slack parameter λ to be maximized without violating any timing constraint. In a timing constraint graph, sequential cells can be represented by nodes and the setup- and hold-time constraints between them by edges.

The problem of optimal distribution of slack in a clock network in sequential circuits can be formulated as a minimum balance problem in directed graphs [11] [12] [5] [13] [15] [16]. In a sequential circuit as shown in Fig. 1.1, we can write the setup and hold time constraints between two sequentially adjacent flip-flops FF_i and FF_j as follows. Let t_i and t_j be respectively the arrival times of the clock signal to FF_i and FF_j; C_p the clock period; $t_{pFF}^{\max}$ and $t_{comb}^{\max}$ ($t_{pFF}^{\min}$ and $t_{comb}^{\min}$) respectively the longest (shortest) propagation delays through a flip-flop and the combinational circuit between FF_i and FF_j; and $t_{setup}^{\max}$ and $t_{hold}^{\max}$ respectively the maximum setup and hold times of a flip-flop. We define $w_{ji} = C_P - (t_{pFF}^{\max} + t_{comb}^{\max} + t_{setup}^{\max})$ and $w_{ij} = (t_{pFF}^{\min} + t_{comb}^{\min}) - t_{hold}^{\max}$, Then, $t_i - t_j \leq w_{ji}$ and $t_j - t_i \leq w_{ij}$ are respectively the setup- and hold-time constraints [17]. The timing constraints of a sequential circuit can be represented with a directed graph, known as a timing constraint graph (see Fig. 1.1), where the nodes are the sequential elements and a timing constraint between a pair of sequential elements is captured as a weighted directed edge between them. For a circuit to be timing-correct, the equivalent timing constraint graph must not have negative cycles.

Process variations in sequential circuits may change any of the propagation delays, setup time, and hold time, and introduce a negative cycle in the timing constraint graph of an otherwise timing-correct circuit. To make the circuit robust, sufficient tolerance (or slack) may be added to the signal paths and clock skews. For the slack distribution problem, we rewrite the setup and hold time constraints respectively as $t_i - t_j \leq w_{ji} - \lambda_{ji}$ and $t_j - t_i \leq w_{ij} - \lambda_{ij}$ to include the slack parameter λ to be maximized without violating any timing constraint. In other words, the slack λ_{uv} in an edge (u, v) in the timing constraint graph allows the circuit to tolerate variations as high as λ_{uv} in w_{uv} without causing a timing violation.

The problem of determining the maximum timing slack λ_{uv} in each edge (u, v) in the timing constraint graph is related to the problem of finding the minimum balance of a directed graph, which is defined below. Consider a weighted, directed, and strongly connected graph $G(V, E, w)$ with node set V and edge set E. Each edge e has an associated weight $w(e)$, given by a weight function $w : E \rightarrow \mathbf{R}$ and a parameter $\lambda(e)$ such that the parameterized weight of that edge is $w(e) - \lambda(e)$. The minimum balance problem is that of finding largest $\lambda(e)$, $e \in E$, by simultaneously maximizing $\lambda(e)$ for all edges without creating any negative cycle [14] [11] [5].

In Fig. 4.8(a), for example, we simultaneously increase $\lambda(e)$ (from $-\infty$) for all edges by the same amount until we get a zero weight cycle in C_1. We cannot increase $\lambda(e)$ further for the edges in C_1 since C_1 would turn negative otherwise. Therefore, $\lambda(e) = 2$ for $e \in E(C_1)$. Now, all edges have their weights reduced by 2. Moreover, all

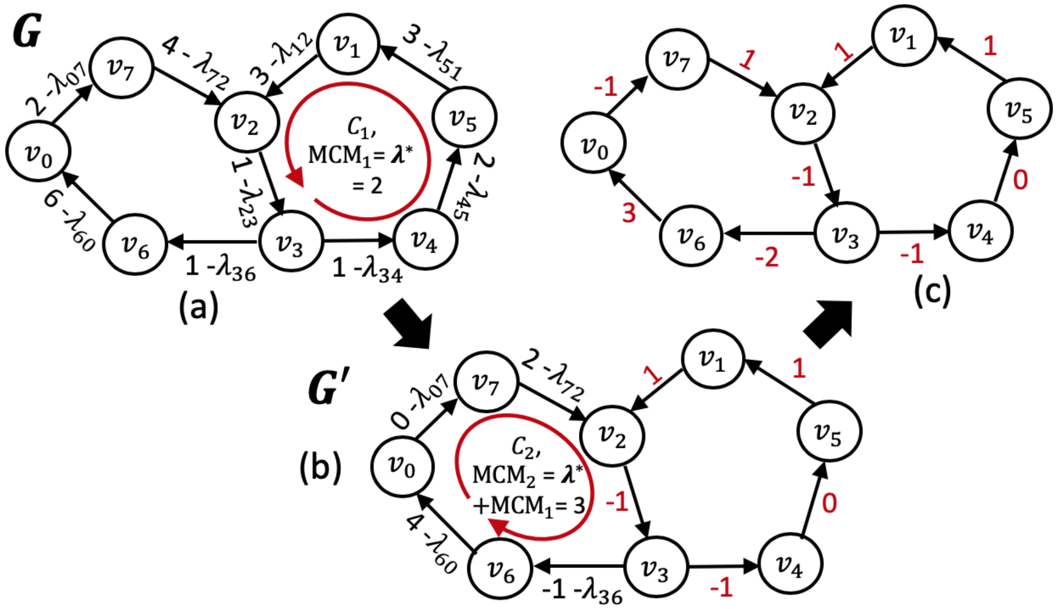


Fig. 1.2.: (a) Every edge in the graph has a parameter $\lambda(e)$. Largest value of $\lambda(e)$ for $e \in E(C_1)$ is 2, since beyond that value cycle C_1 is negative. (b) Given that $\lambda(e)$, $e \in E(C_1)$ is 2, largest value of $\lambda(e)$ for $e \in E(C_2)$ and $\notin E(C_1)$ is 3, since beyond that value cycle C_2 is negative. (c) Minimum-balanced graph.

edges in C_1 now lose their parameters $\lambda(e)$ (see Fig. 4.8(b)). We repeat the process of increasing $\lambda(e)$ (from $-\infty$). This time, we stop when all $\lambda(e)$ is 1 and the cycle C_2 becomes a zero weight cycle. Since all the edge weights have been reduced by 2 earlier because of C_1, $\lambda(e) = 3$ for $e \in E(C_2)$ and $\notin E(C_1)$. Fig. 4.8(c) shows the minimum-balanced graph where every edge (e) has got its weight downshifted by $\lambda(e)$ (from its original weight).

This example also demonstrates two main subroutines used in a conventional minimum balance algorithm [14] [11]. The first computes the minimum cycle mean (MCM) and finds the corresponding minimum-mean cycle. The mean of a cycle is obtained by dividing the sum of edge weights $(w(e))$ by the number of edges with parameters $\lambda(e)$ in the cycle. The MCM is the minimum of all cycle means in the graph [1] [6] and a cycle whose mean is minimum is a minimum-mean cycle. Finding the minimum cycle mean of the graph is equivalent to finding the largest $\lambda(e)$ for which there are no negative cycles [2]. For example, the value 2 that we found for

$\lambda(e)$ for edges $e \in E(C_1)$ is the MCM of G (denoted as MCM_1 in Fig. 4.8(a)) and the cycle C_1 is the minimum-mean cycle.

In second chapter of the dissertation, we study few classical graph algorithms to solve MCM problem that can also be used in a conventional minimum balance algorithm [2] [14] [18] [1] [19]. The algorithm by Young, Tarjan, and Orlin (YTO) [14], when implemented with a binary heap, has been reported to be the fastest MCM algorithm in practice [7] even when it has higher asymptotic time complexity than Karp's algorithm [1] . 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) [18] 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. While studying these algorithms, one particularly interesting characteristic of the sequential circuits came to our observation, that is that the cycles are usually small, made of few edges. For this reason we infer that even though the problem in hand is quite enormous it exhibits a well-defined structure, something to leverage on. Most importantly, this particular characteristic can make early termination of Karp's algorithm even more useful. Motivated by this we went on to investigate the Hartman and Orlin's algorithm [18] in-depth. 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.

The second main subroutine in the conventional minimum balance algorithm updates the graph to facilitate the determination of the remaining $\lambda(e)$. For example, it re-weights every edge in G by downshifting its weight by 2 and removing parameters $\lambda(e)$ from the edges in C_1. The MCM (denoted as λ^* in Fig. 4.8(b)) of the updated graph is 1, with C_2 being the minimum-mean cycle. To account for the downshifting by 2, $\lambda(e) = \lambda^*+$ MCM$_1 = 3$ (denoted as MCM$_2$ in Fig. 4.8(b)) for $e \in E(C_2)$ and $e \notin E(C_1)$. It is therefore apparent how minimum balance problem can be solved by solving the MCM problem on a series of updated graphs. One can note that the graph update operation involves the entire graph. Thus, the graph update operation, which has a time-complexity of $\Theta(V+E)$ [11], has been considered to be a bottleneck in the conventional minimum balance algorithm [14]. Schneider and Schneider presented in an earlier work [3] a different approach for updating the graph; however, the time complexity of the graph update operation in [3] is the same as that from [14] [11].

In third chapter of the dissertation we propose an improvement to the conventional minimum balance algorithms [14] [11] [3] where the graph update operation involves only edges adjacent to the minimum-mean cycle; it involves the entire graph only in the worst case. This improves the efficiency of the minimum balance algorithm. We evaluate the conventional algorithms and the proposed algorithm using graphs obtained from IWLS 2005 benchmark circuits [20] and graphs that are randomly generated. We observe that our algorithm has better runtime performance compared to the conventional minimum balance algorithms, averaging 41.20% and 31.43% runtime reductions for circuit graphs and randomly generated graphs, respectively.

We also apply the minimum cycle mean algorithm in latency insensitive system design. Timing violations in high performance communication links may occur in the late stages of the physical design process of system-on-chips (SoCs). To address that, latency insensitive systems (LISs) employ pipelining in the communication channels through the insertion of relay stations. Although the functionality of an LIS is robust with respect to the communication latencies, imbalances in relay station insertion may degrade the throughput performance of the system [21] [22] [23]. While having a large number of buffer queues could eliminate such performance loss, the system may not have adequate area to accommodate these buffers. The problem of buffer queue sizing for maximizing performance while meeting buffer area constraints has been solved using a mixed integer linear program (MILP) formulation; however, such an approach is not scalable [24] [25] [22]. In fourth chapter of the dissertation, we formulate the buffer queue sizing problem as a parameterized graph optimization problem where for every communication channel there is a parameterized edge with buffer counts as the edge weight. We then use a minimum cycle mean algorithm to determine from which edges buffers can be removed safely. Experimental results on large LISs suggest that the proposed approach is scalable. Moreover, the quality of the solutions, in terms of the throughput and the size of buffer queues, is observed to be as good as that of the MILP based approach.

2. MINIMUM CYCLE MEAN (MCM) ALGORITHMS

Various applications in the design of circuits and systems require computation of minimum cycle mean (MCM) in a directed graph [5] [6] [7] [8] [9] [10]. Some important applications are in the areas of clock network optimization, including clock period minimization, slack optimization, and timing analysis [12] [5] [13] [35]. MCM algorithms are also used in other graph algorithms and applications [3] [4].

Following the discussion in Chapter 1, one can realize that the calculation of the mean of a cycle in a timing constraint graph can be done 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 minimum cycle mean (MCM) of a graph [1].

There is a more general concept of a parameterized graph [2] 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 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 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 [1] 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 [36]. 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 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 solved a series of parametric shortest path (tree) problem for MCM calculation in $O(|V||E|\log|V|)$ time [2]. 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 shortest-path tree. Instead of keeping track of the edges, Young, Tarjan and Orlin's algorithm [14] keeps track of the nodes for which its incoming edge could change in the next iteration. We refer to this algorithm as 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 [37] [12] [7] [5] [13] 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 incoming and outgoing adjacency lists to store the graph information [37]. Although the two lists contain the same information, they facilitate faster update of the binary heap. As two adjacency lists are maintained, the memory overhead of YTO is $O(|E|)$.

The $O(|E|)$ overhead of YTO appears leaner than the $O(|V|^2)$ 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 algorithm [18], 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 [20] 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: detection of cycles, calculation of dual vector π, and 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 2.4 for details). The checking of feasibility condition is equivalent to one iteration of Bellman-Ford algorithm, which requires $O(|E|)$ time complexity.

The first improvement is a filtering technique that reduces the number of k-edge shortest paths to be traversed to detect cycles. 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 $O(k|V|)$. Second, 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 $O(k|V|)$. Third, the checking of feasibility condition can now be carried out with $O(|V|)$ 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.

2.1 Minimum cycle mean (MCM)

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:

$$\lambda(C) = \frac{w(C)}{\tau(C)}, \tag{2.1}$$

which is the total edge weight of the cycle divided by the number of edges. The minimum cycle mean 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 minimum cycle mean (MCM) problem [1] [2].

If the graph G is not strongly connected, the minimum cycle mean can be obtained by computing the smallest among the minimum cycle means of all strongly connected components 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.

2.2 Parametric shortest path algorithms

Karp and Orlin (KO) [2] proposed a parametric shortest path algorithm for the MCM problem. Based on the observation that λ^* is the largest (edge parameter) λ

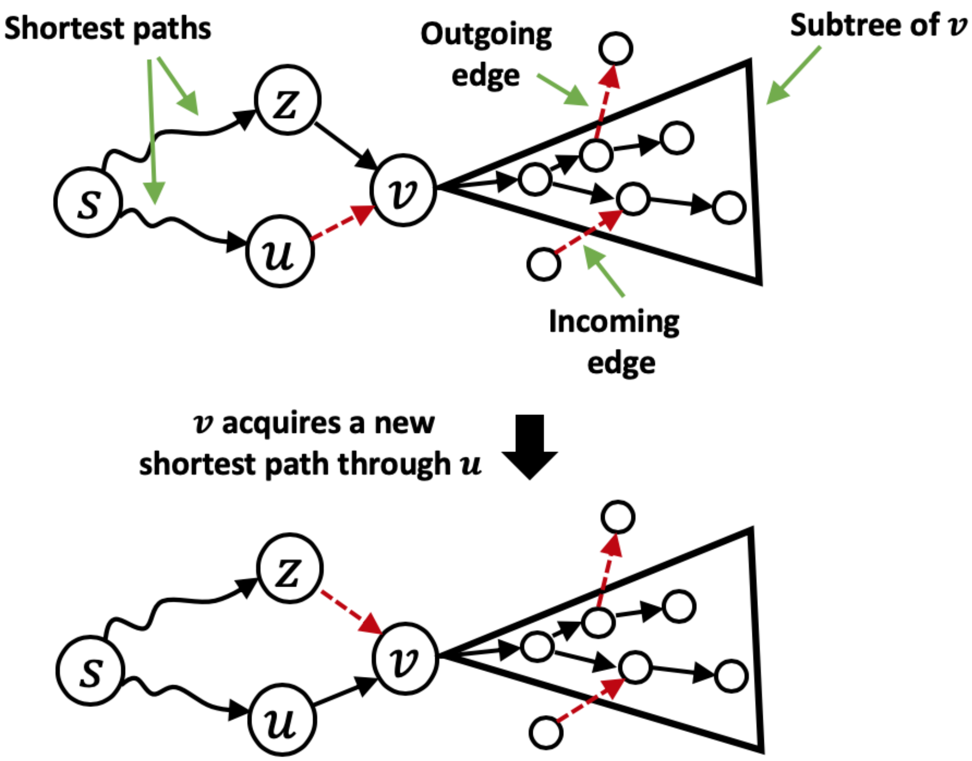


Fig. 2.1.: As λ increases, the shortest path from source node s to node v changes the parent node of v from z to u. The shortest paths to all nodes in the subtree of v are therefore updated and the the algorithm has to update information for all nodes in the heap that are adjacent to the subtree. For 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 [2] [19] [7] [37]. All adjacent edges to the subtree are shown as red, dashed arrows.

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. 2.1, 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. Young, Tarjan and Orlin's algorithm (YTO) [14] 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 shortest-path tree, the heap keeps track of the nodes for which its incoming edge could change in the next iteration. In Fig. 2.1, 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 [2] [19] [7] [37]. In Fig. 2.1, 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 [37] [7]. As a consequence, the memory overhead of the YTO algorithm is $O(|E|)$. The reader may refer [14] [37] [7] for more details of the YTO algorithm and its implementation. YTO using binary heap has been reported to be the fastest MCM algorithm [37] [12] [7] [5] [13]. Our implementation of YTO in this work follows the pseudocode presented in [7].

2.3 Karp's algorithm

Karp's dynamic programming algorithm [1] 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 $0 \leq k \leq |V| - 1$, 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. 2.2. With this, Karp characterized MCM (λ^*) as:

$$\lambda^* = \min_{v \in V} \max_{0 \leq k \leq |V|-1} \left[\frac{D_{|V|}(v) - D_k(v)}{|V| - k} \right], \tag{2.2}$$

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, $D_k(v) = \infty$.

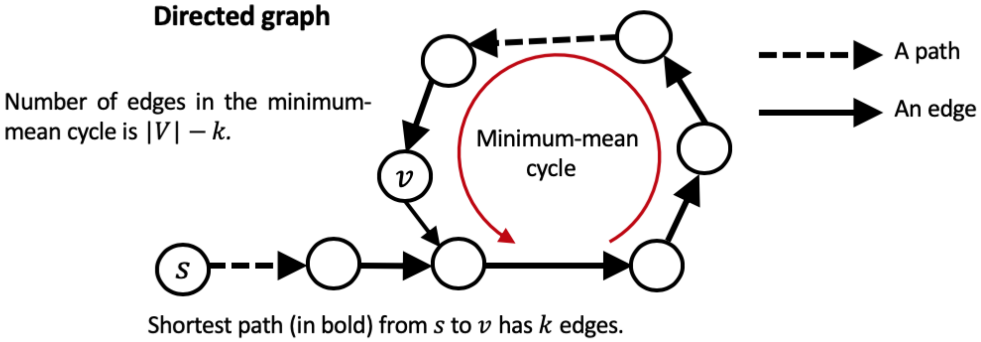

Directed graph
Number of edges in the minimum-mean cycle is $|V| - k$.
Minimum-mean cycle
v
s
A path
An edge
Shortest path (in bold) from s to v has k edges.

Fig. 2.2.: A 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.

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

$$D_k(v) = \min_{(u,v) \in E} [D_{k-1}(u) + w(u,v)], \tag{2.3}$$

for $k = 1, 2, ..., |V|$, 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 *vertical relaxation* (see the pseudocode below) because it uses the $(k-1)$-st row to compute the k-th 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.

Vertical Relaxation

```
vertical_relaxation($G$, $D$, $P$, $k$)
  for each edge $(u, v) \in E$ do
    if $D_k[v] > D_{k-1}[u] + w(u, v)$ then
      $D_k[v] = D_{k-1}[u] + w(u, v)$
      $P_k[v] = u$
    end if
  end for
end vertical_relaxation
```

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 Eq. (2.2).

Karp's Algorithm

```
Input  : Directed graph $G(V, E, w)$, a strongly connected component (SCC)
Output: Minimum cycle mean $\lambda^*$
/* Initialization */
$\lambda_{global} = \infty$ /* global variable to store minimum cycle mean */
for $k = 0$ to $|V|$ do
  for each node $v \in V$ do
    $D_k[v] = \infty$
    $P_k[v] = -1$
  end for
end for
$D_0[s] = 0$
/* $D$- and $P$-tables computation*/
for $k = 1$ to $|V|$ do
  vertical_relaxation($G$, $D$, $P$, $k$)
end for
/* MCM computation */
for each node $v \in V$ do
  if $D_{|V|}[v] \neq \infty$ then
    $\lambda_v = -\infty$
    for $k = 0$ to $|V| - 1$ do
      $\lambda_v = \max(\lambda_v, (D_{|V|}[v] - D_k[v])/(|V| - k))$
    end for
    $\lambda^* = \min(\lambda^*, \lambda_v)$
  end if
end for
return $\lambda_{global}$
```

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 adjacency list compared to two adjacency lists 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.

2.4 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 the Karp's algorithm was proposed by Hartmann and Orlin (HO) in [18] 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 non-minimum-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 $\pi[v]$ denote the shortest distance from the source node to v in this modified graph. Then, the following constraints

$$\pi[v] \leq \pi[u] + w(u, v) - \lambda_{\min},\ \forall e(u, v) \in E \tag{2.4}$$

must be satisfied. This is the dual formulation referred to in [18], and we will refer to π as the dual vector and Eq. (2.4) 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, $0 \leq j \leq k$. Therefore, $\pi[\cdot]$ can be computed using the expression

$$\pi[v] = \min_{0 \leq j \leq k} [D_j[v] - j\lambda_{\min}],\ \forall v \in V, \tag{2.5}$$

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 (Eq. (2.5)) and the feasibility check (Eq. (2.4)) are straightforward and their pseudocodes are provided below.

π Calculation

```
π_calculation($D$, $k$, $\lambda_{\min}$)
  for each node $v \in V$ do
    $\pi[v] = \infty$
    for $j = 0$ to $k$ do
      $\pi[v] = \min(\pi[v], D_j[v] - j * \lambda_{\min})$
    end for
  end for
  return $\pi$
end π_calculation
```

Feasibility Check

```
feasibility_check($\pi$, $\lambda_{\min}$)
  for each edge $e = (u, v) \in E$ do
    if $\pi[v] > \pi[u] + w(u, v) - \lambda_{\min}$ then
      return $false$
    end if
  end for
  return $true$
end feasibility_check
```

The calculation of π have $O(k|V|)$ 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 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) [18]. Hartmann and Orlin suggested several techniques for making forward traversal efficient. It was argued that any cycle in the path of minimum-mean cycle is a minimum-mean cycle. Therefore 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 [19] [38] to find a cycle along a path. 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 re-initialization of *level_array* efficient.

$\lambda_{\min}$ Calculation in a Path

```
$\lambda_{\min}$_calculation_in_a_path($D$, $P$, $k$, $v_{start}$, $\lambda_{\min}$)
  $length = 0$
  $v = v_{start}$
  $j = k$
  while $j \geq 0$ do
    if $level\_array[v] > -1$ then
      $\lambda = (D_{level\_array[v]}[v] - D_j[v])/(level\_array[v] - j)$
      $\lambda_{\min} = \min(\lambda_{\min}, \lambda)$
      break
    end if
    $level\_array[v] = j$
    $level\_stack[length] = v$
    $++length$
    $v = P_j[v]$
    $--j$
  end while
  for $j = length - 1$ to 0 do
    $level\_array[level\_stack[j]] = -1$
  end for
  return $\lambda_{min}$
end $\lambda_{\min}$_calculation_in_a_path
```

As mentioned earlier, all cycles in a k-edge shortest path that contains a minimum-mean cycle are minimum-mean cycles [18]. Therefore, it is only necessary to detect the first simple cycle in either forward traversal or backward traversal of the path. Therefore, the pseudocode performs an incomplete backward traversal that terminates when the first cycle is detected, with a **break** statement in line 8. We refer to this version as HO/b. The pseudocode used here is almost identical to that from [19] except that the original version does not have a **break** statement in line 8. In other words, the version from [19] detects all the simple cycles along a k-edge shortest path.

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

$\lambda_{\min}$ Calculation

```
λmin_calculation($D$, $P$, $k$)
  $\lambda_{\min} = \infty$
  for each node $v \in V$ do
    if $D_k[v] \neq \infty$ then
      $\lambda_{\min}$ = λmin_calculation_in_a_path($D$, $P$, $k$, $v$, $\lambda_{\min}$)
    end if
  end for
  return $\lambda_{\min}$
end λmin_calculation
```

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.

Early Termination

```
early_termination($D$, $P$, $k$, $|V|$)
  if $k$ is a power of 2 or $k == |V|$ then
    /* $\lambda_{\min}$ calculation */
    $\lambda_{\min}$ = λmin_calculation($D$, $P$, $k$)
    $\lambda_{global} = \lambda_{\min}$ /* update $\lambda_{global}$ */
    /* Dual vector $\pi$ calculation */
    $\pi$ = π_calculation($D$, $k$, $\lambda_{global}$)
    /* Feasibility condition check */
    return feasibility_check($\pi$, $\lambda_{global}$)
  end if
  return $false$
end early_termination
```

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 the 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 [18]. As HO/b is more compatible with the Karp's algorithm (because of the presence of the P-table), our focus in this work is to improve the HO/b algorithm. The pseudocode for the HO/b algorithm, or Karp's algorithm with early termination from HO, is shown below.

Allocate and Initialize D_k and P_k

```
allocate_and_initialize_$D_k$_and_$P_k$($k$)
  allocate memory for $D_k$ and $P_k$
  for each node $v \in V$ do
    $D_k[v] = \infty$
    $P_k[v] = -1$
  end for
end allocate_and_initialize_$D_k$_and_$P_k$
```

HO Algorithm (Karp's Algorithm with Early Termination)

```
Input  : Directed graph $G(V, E, w)$, a strongly connected component (SCC)
Output: Minimum cycle mean $\lambda^*$
/* Initialization */
$\lambda_{global} = \infty$ /* global variable to store minimum cycle mean */
for each node $v \in V$ do
  $level\_array[v] = -1$ /* global array for $\lambda_{\min}$ 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 /* $\lambda_{global}$ updated */
    break
  end if
end for
return $\lambda_{global}$
```

Here, early termination allows memory to be allocated on-demand instead of a one-time allocation of $\Theta(|V|^2)$ memory at 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 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 algorithm and Karp's algorithm have the same memory usage for storing the graph using a single adjacency list.

2.5 The proposed MCM algorithm

In order to keep the same theoretical time complexity as Karp's algorithm in the worst case, Hartmann-Orlin 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) [18]. 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 being 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 sufficient number of rows have been computed. This is a serious bottleneck of Hartmann-Orlin's algorithm for large graphs since the cost of vertical relaxation is $O(|E|)$ for every row. However, we could not perform early termination check at every row as the cost of 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 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 the section.

2.5.1 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

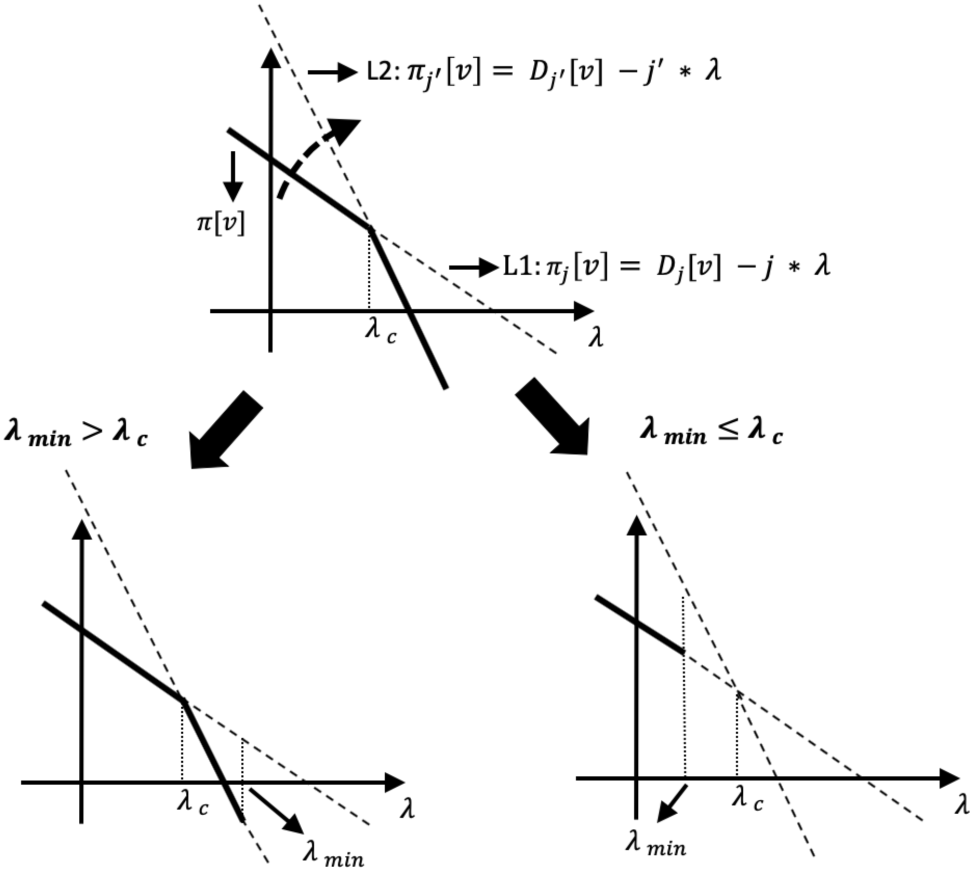

L2: $\pi_{j'}[v] = D_{j'}[v] - j' * \lambda$
$\pi[v]$
L1: $\pi_j[v] = D_j[v] - j * \lambda$
λ_c
λ
$\lambda_{min} > \lambda_c$
$\lambda_{min} \leq \lambda_c$
λ_c
λ
λ_{min}
λ_c
λ
λ_{min}

Fig. 2.3.: Filtering of redundant k-edge shortest paths for efficient π calculation. $L1$ and $L2$ represents the j- and j'-edge shortest paths, respectively, with $j < j'$. If λ_{min} $\leq \lambda_c$, where λ_c is the intersection of $L1$ and $L2$, the j'-edge path is redundant and can be pruned.

than λ_{global} after vertical relaxation at row k, it is necessary to re-calculate the dual vector π in the modified graph $G(V, E, w - \lambda_{\text{min}})$. At row k, for a particular node v, we calculate for each j-edge shortest path, $0 \leq j \leq k$, $\pi_j[v] = D_j[v] - j\lambda_{\text{min}}$, and pick the minimum among them to be $\pi[v]$ (see Eq. (2.5)).

However, not all j-edge shortest paths have the potential to become the real shortest path in the modified graph. In particular, as λ_{min} decreases towards λ^*, many of these paths can never be the shortest path for lower λ_{min}. We illustrate that in Fig. 2.3 where line $L1$ corresponds to $\pi_j[v]$ of the j-edge shortest path to v and line $L2$ corresponds to the $\pi_{j'}[v]$ of the j'-edge shortest path to v, with $j' > j$. The bold contour shows the dual vector $\pi[v] = \min(\pi_j[v], \pi_{j'}[v])$ 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 $\pi_j[v] \leq \pi_{j'}[v]$, 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.

Efficient π Calculation

```
1  efficient_π_calculation($D$, $k$, $\lambda_{\min}$)
2    for each node $v \in V$ do
3      if $D_k[v] \neq \infty$ then
4        $++\pi\_stack[v]$
5        $\pi\_edge[\pi\_stack[v]][v] = k$
6      end if
7      $\pi[v] = \infty$
8      $index\_min = -1$
9      for $i = 0$ to $\pi\_stack[v]$ do
10       $j = \pi\_edge[i][v]$
11       if $\pi[v] > D_j[v] - j * \lambda_{\min}$ then
12         $\pi[v] = D_j[v] - j * \lambda_{\min}$
13         $index\_min++$
14         $\pi\_edge[index\_min][v] = j$
15       end if
16     end for
17     $\pi\_stack[v] = index\_min$
18   end for
19   return $\pi$
20 end efficient_π_calculation
```

In the pseudocode, $\pi\_edge[\cdot][v]$ stores the indices of the irredundant paths and $\pi\_stack[v]$ stores the highest index of 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 3–6, if there is a k-edge shortest path for node v, $\pi\_stack[v]$ must be incremented to account for the new path, 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 $\pi[v]$, we record the index of the shortest path that accounts for that (lines 11–15). In other words, as we scan the current irredundant paths in the order of increasing path length, any paths that result in a smaller $\pi[v]$ 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.

2.5.2 Efficient feasibility check

Let us re-examine the dual constraints. Eq. (2.4) 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, $0 \leq j \leq k$, the question becomes that of asking whether we can get a smaller $\pi[v]$ 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 Eq. (2.4) is equivalent to

$$\pi[v] \leq D_{k+1}[v] - (k+1)\lambda_{\min},\ \forall v \in V. \tag{2.6}$$

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 is in the last early termination check that succeeds. All earlier $O(|E|)$ efforts account for the necessary vertical relaxations.

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: $\lambda_{\min} = \lambda_{global}$ and $\lambda_{\min} < \lambda_{global}$, 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 $\lambda_{\min} \neq \infty$.

Efficient Feasibility Check

```
1  efficient_feasibility_check($\pi$, $D$, $k$, $\lambda_{\min}$)
2    $early\_termination\_flag = true$
3    for each node $v \in V$ do
4      if $\pi[v] > D_k[v] - k * \lambda_{\min}$ then
5        $\pi[v] = D_k[v] - k * \lambda_{\min}$
6        $++\pi\_stack[v]$
7        $\pi\_edge[\pi\_stack[v]][v] = k$
8        $early\_termination\_flag = false$
9      end if
10   end for
11   return $early\_termination\_flag$
12 end efficient_feasibility_check
```

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 3–8. length k found in the modified graph $G(V, E, w - \lambda_{\min})$. Note that the right hand side of Eq. (2.6) is similar to the term that appears on the right hand side of Eq. (2.5). We can therefore use the right hand side of Eq. (2.6) 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 subsection. 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:

Efficient Early Termination

```
efficient_early_termination($D$, $P$, $k$, $\lambda_{global}$)
  /* $\lambda_{\min}$ calculation */
  $\lambda_{min}$ = efficient_$\lambda_{\min}$_calculation($D$, $P$, $k$, $\lambda_{global}$)
  if $\lambda_{global} > \lambda_{min}$ then
    $\lambda_{global} = \lambda_{min}$ /* update $\lambda_{global}$ */
    /* Dual vector $\pi$ calculation */
    $\pi$ = efficient_$\pi$_calculation($D$, $k$, $\lambda_{global}$)
  else if $\lambda_{\min} \neq \infty$ then
    /* Feasibility condition check */
    return efficient_feasibility_check($\pi$, $D$, $k$, $\lambda_{global}$)
  end if
  return $false$
end efficient_early_termination
```

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 **efficient_$\boldsymbol{\lambda}_{\mathbf{min}}$_calculation**, the details of which will be provided in the next subsection. As mentioned earlier, there are two cases to be considered: $\lambda_{\min} = \lambda_{global}$ and $\lambda_{\min} < \lambda_{global}$. In the former, we perform efficient feasibility check, and update π if necessary. In none of 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.

2.5.3 Efficient $\lambda_{\min}$ calculation

For $\lambda_{\min}$ calculation at row k, the bottleneck is in the detection of cycles in all k-edge shortest paths. We propose three improvements to reduce the number of k-edge shortest paths that we have to consider for cycle detection.

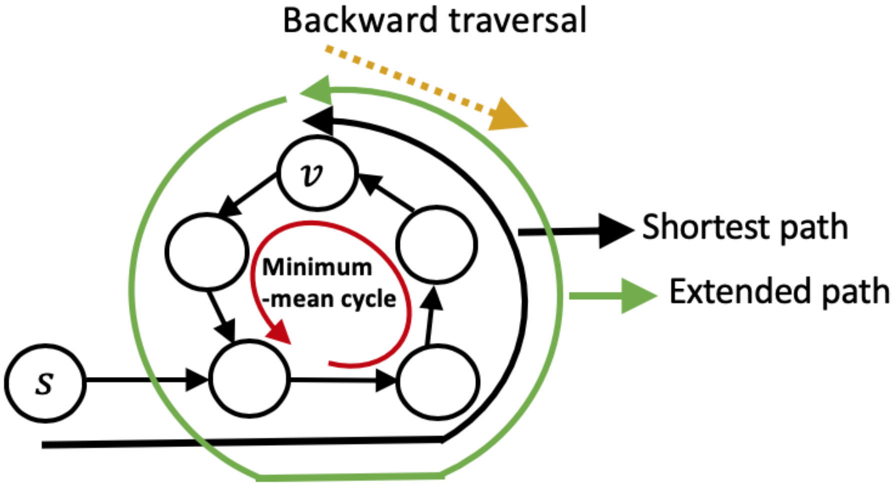


Fig. 2.4.: The minimum-mean cycle containing node v occurs in the extended path beyond the shortest path to v.

2.5.3.1 Filtering based on Karp's characterization of MCM

The first improvement is based on the characterization of MCM in [1]. Consider a k-edge shortest path that ends with node v. If v is on the minimum-mean cycle, the k-edge shortest path that contains that cycle is actually an extension of the shortest path to v from s. This is illustrated in Fig. 2.4. 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. 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. However, such information is not explicitly calculated in Karp's algorithm or in HO. While the proof of characterization of MCM in [1] relies on the shortest path, the algorithm itself does not explicitly compute this information anywhere.

To overcome that, we can make use of the concept of a pseudo-source node. 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 [36], and the initialization of the shortest path tree when $\lambda = -\infty$ in YTO [14]. 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. If the edge weights in the input graph are all positive, the edges from the pseudo-source node to all other nodes in the graph are the shortest paths. If some edge weights in the input graph are negative, we simply subtract the most negative edge weight from all edges in the input graph to turn them non-negative. Of course, in that case, we will have to add this offset to the computed MCM value to obtain the true MCM.

If the source node s in Fig. 2.4 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 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 0-th 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.

Given a k-edge shortest path that ends with node v, we are of course not interested to check whether $P_k[v] = P_0[v]$ because the pseudo-source node is not in the original graph and there are no incoming edges to it. Rather, we will perform vertical relaxation to obtain the first row of D- and P-table. For a k-edge shortest path that ends with node v, we will check for the condition to decide whether we should perform a backward traversal for cycle detection.

$$P_k[v] = P_1[v]. \tag{2.7}$$

2.5.3.2 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 a minimum cycle mean. Based on the available shortest path information in $sp[v]$, the mean of this possible cycle C is $w(C)/\tau(C) = (D_k[v] - D_1[v])/(k-1)$. If we have the smallest cycle mean computed so far (up to row $k-1$) 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}:

$$(D_k[v] - D_1[v])/(k-1) < \lambda_{global} \tag{2.8}$$

2.5.3.3 Filtering based on Hartmann-Orlin's characterization of minimum-mean cycle

Hartmann-Orlin argued in [18] that all cycles in a k-edge shortest path that contains a minimum-mean cycle are minimum-mean cycles. Therefore, it is only necessary in the backward traversal of a path to detect the first simple cycle, which is being done in the pseudocode $\boldsymbol{\lambda_{\min}}$**_calculation_in_a_path**.

Some of the paths where cycles have been found 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 are also minimum-mean cycles. They will result in the same $\lambda_{\min} = \lambda^*$ $(= \lambda_{global})$, 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

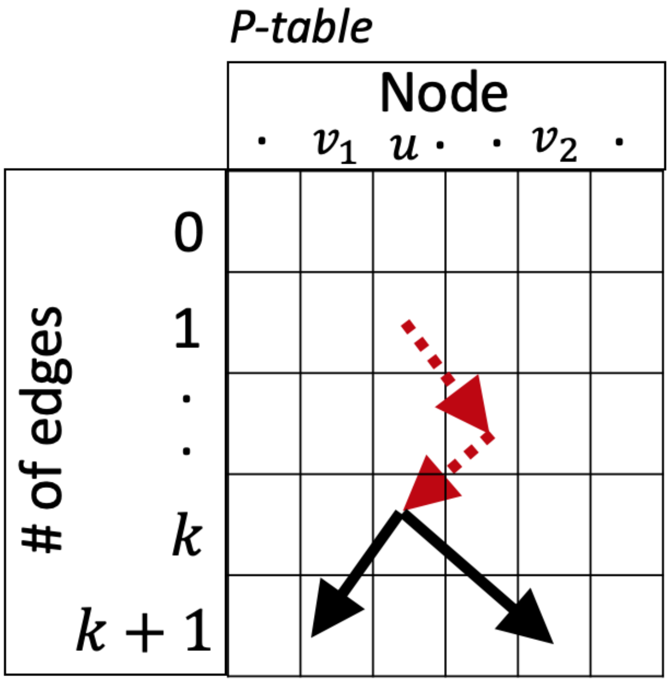


Fig. 2.5.: Filtering in cycle detection using a concept presented in [18]: Any cycle in the path of minimum-mean cycle is a minimum-mean cycle. Therefore, there is no need to detect cycles in an extension of a path that has a cycle detected earlier. At row k, we detect a cycle (red dotted line) by traversing for node u. We do not have to consider cycle detection and λ_{min} calculation for the extended paths at ends at nodes v_1 and v_2.

cycle, there is no reason to consider traversing the path since no cycle with minimum-mean value can occur in that path, as argued by Hartmann-Orlin.

We can conclude that there is no reason 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 λ_{min} calculation.

The example in Fig. 2.5 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 $detected_k[v]$, $v \in V$, 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

$$detected_k[v] = false. \tag{2.9}$$

The filtering of k-edge shortest paths based on Eq. (2.7), Eq. (2.8), and Eq. (2.9) is performed in line 6 of the pseudocode **efficient_$\boldsymbol{\lambda}_{\mathbf{min}}$_calculation**.

Efficient $\lambda_{\min}$ Calculation

```
1  efficient_λmin_calculation($D$, $P$, $k$, $\lambda_{global}$)
2    $\lambda_{min} = \lambda_{global}$
3    for each node $v \in V$ do
4      if $D_k[v] < \infty$ then
5        $detected_k[v] = detected_{k-1}[P_k[v]]$
6        if $P_k[v] == P_1[v]$ and $(D_k[v] - D_1[v])/(k-1) < \lambda_{min}$ and $detected_k[v] == false$ then
7          $\lambda_{min}$ = λmin_calculation_in_a_path($D$, $P$, $k$, $v$, $\lambda_{min}$)
8        end if
9      end if
10   end for
11   swap $detected_{k-1}$ and $detected_k$
12   return $\lambda_{\min}$
13 end efficient_λmin_calculation
```

Note that $detected_k[v]$, $v \in V$, takes on the flag of its parent node $P_k[v]$ in line 5 of the pseudocode. In the pseudocode **$\boldsymbol{\lambda}_{\mathbf{min}}$_calculation_in_a_path**, it may be necessary to update $detected_k[v]$ when a cycle is detected. Assuming that the pseudocode **$\boldsymbol{\lambda}_{\mathbf{min}}$_calculation_in_a_path** has access to $detected_k[v]$, it has to set $detected_k[v]$ to $true$ when a cycle is detected. The following statement has to be inserted between line 6 and line 7 of the pseudocode **$\boldsymbol{\lambda}_{\mathbf{min}}$_calculation_in_a_path**.

```
$detected_k[v] = true$
```

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

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.

2.5.4 Pseudocode of the proposed MCM algorithm

The pseudocode of the proposed MCM algorithm is presented below.

The Proposed MCM Algorithm

```
   Input  : Directed graph $G(V, E, w)$, a strongly connected components (SCC)
   Output: Minimum cycle mean $\lambda^*$
1  /* Initialization */
2  $\lambda_{global} = \infty$ /* global variable to store minimum cycle mean */
3  allocate_and_initialize_$D_k$_and_$P_k$(0)
4  for each node $v \in V$ do
5    $level\_array[v] = -1$ /* global array for $\lambda_{\min}$ calculation */
6    $detected_{k-1}[v] = false$ /* global array for $\lambda_{\min}$ calculation */
7    $\pi\_stack[v] = -1$ /* global array for $\pi$ calculation */
8    $D_k[v] = 0$ /* pseudo-source initialization */
9  end for
10 /* $D$- and $P$-tables computation */
11 for $k = 1$ to $|V|$ do
12   allocate_and_initialize_$D_k$_and_$P_k$($k$)
13   vertical_relaxation($G$, $D$, $P$, $k$)
14   if efficient_early_termination($D$, $P$, $k$, $\lambda_{global}$) then /* $\lambda_{global}$ updated */
15     break
16   end if
17 end for
18 return $\lambda_{global}$
```

Here, we assume that all edge weights are non-negative so that we can directly apply the concept of 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.

Similar to HO/b, 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. The iterations in lines 11–17 are almost identical to lines 9–15 in the HO algorithm except for one major difference. We call **efficient_early_termination** here. It is important to note that the proposed algorithm performs early termination check at every row.

2.6 Experimental results

We have implemented three algorithms: YTO, HO/b, and the proposed MCM algorithm. All algorithms are implemented in C++ and compiled with compiler level optimization enabled. Our implementation of YTO is similar to the pseudocode reported in [7] that uses binary heap. These implementations are evaluated on a standalone server, which is a Linux machine with Intel(R) Xeon(R) CPU E5-2660 (2.60GHz) and 66GB of RAM.

We evaluate the performance using IWLS 2005 benchmark circuits [20] and randomly generated strongly connected directed graphs created with a technique prescribed in [7] [8]. We will discuss the circuit graph creation in Section 2.6.3. To generate a random graph using the technique in [7] [8], 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 pre-defined range are generated and assigned. Unless mentioned otherwise, all reported runtimes include everything from initializing data structures, reading graphs, to getting the final MCM output.

2.6.1 Benchmarking

The YTO algorithm has been implemented by various groups in the past, targeting different applications [12] [5] [13] [35] [6] [7] [8]. Dasdan et. al. [19] [7] [6] 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. [37]. These studies looked at the sparse graph examples.

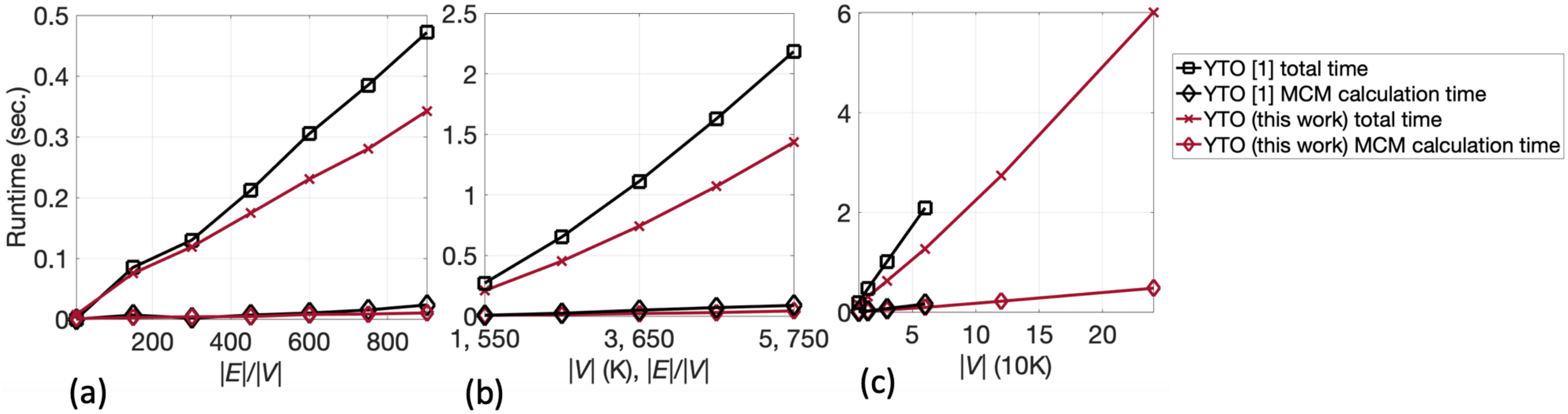


Fig. 2.6.: Benchmarking our implementation of YTO with the implementation from [7] [39]. The results for dense graphs are shown in (a) and (b). In (a), the graph density $|E|/|V|$ is varied from 5 to 900 for $|V| = 1\text{K}$, and in (b) the graph size is varied by changing $(|V|, |E|/|V|)$ from (1K, 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 [7] [39] cannot handle large graphs as shown by the missing data in the plot in (c).

Dasdan [39] 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 [37], there may be some loss in accuracy. We have also observed that the implementation from [39] 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 [39]. Fig. 2.6 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. 2.6(c) shows that our implementation can handle larger graphs than the implementation from [39]. Therefore, for the remainder of this section, all results reported under YTO are obtained using the floating point version of our implementation.

2.6.2 Memory usage

We first evaluate memory usage of the three algorithms: YTO, HO/b, and the proposed algorithm. While Karp's algorithm uses $\Theta(|V|^2)$ memory for storing information in the D- and P-tables [1], HO/b 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 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 and the proposed algorithm store only one graph structure (forward or reverse). Therefore, HO/b and

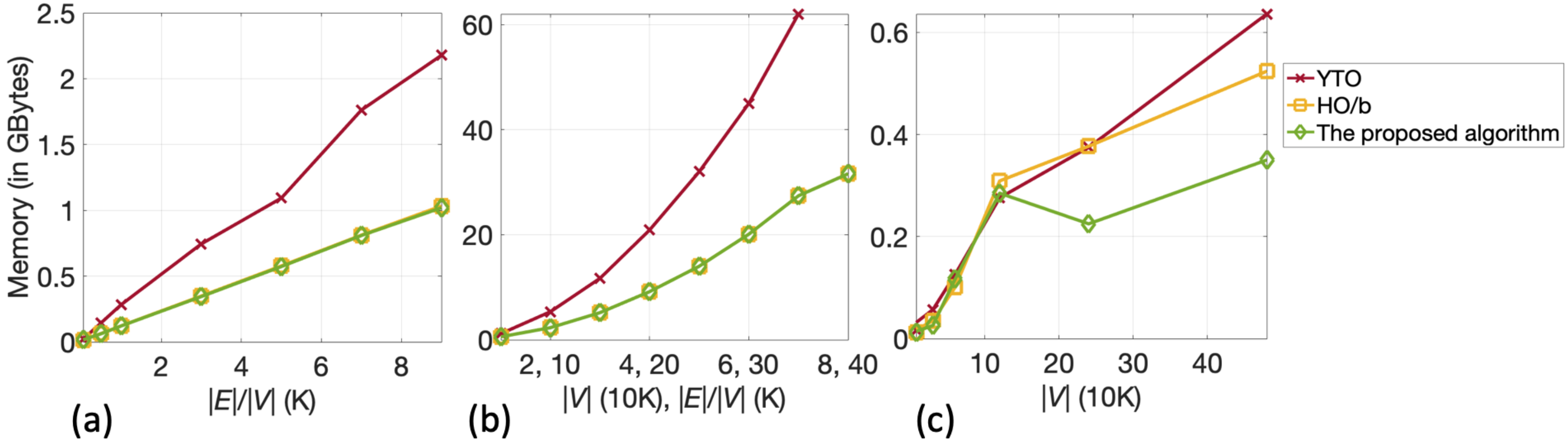


Fig. 2.7.: Comparison of memory usage of YTO, HO-B, and the proposed MCM algorithm. The results for dense graphs are shown in (a) and (b). In (a), the graph density $|E|/|V|$ is increased from 50 to 9000 for $|V| = 10\text{K}$, and in (b), the graph size is varied by changing ($|V|$, $|E|/|V|$) from (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.

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. 2.7. One can observe that the proposed algorithm and HO/b 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. 2.7, 2.8 and 2.9. We also observe (not shown here) that the implementation of YTO from [39] uses more memory than our implementation of YTO.

We must point out that as the proposed algorithm performs early termination check at every row, it has lower memory usage than HO/b. While its advantage over HO/b in Fig. 2.7(a) and (b) is relatively small, the difference is evident in Fig. 2.7(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.

2.6.3 Runtime performance

To evaluate the performance using practical circuits, timing constraint graphs are constructed after synthesis of ISCAS89 and OpenCores designs from IWLS 2005 benchmark [20]. These designs are specified in Verilog using Synopsys tool chain and a $32nm$ technology library. We decompose the graph into strongly connected components, and apply the algorithms on these individual components. Table 2.1 provides the circuit graph information. We present runtime results for the largest strongly connected components in Table 2.2. Our algorithm performs better than HO/b and is comparable to YTO.

We now present results on random graphs. In Fig. 2.8(a), the density of the graph, $|E|/|V|$, is varied while in Fig. 2.8(b), the graph size is varied by changing ($|V|$, $|E|/|V|$) from (10K, 5K) to (80K, 40K). Graphs in Figs. 2.8(a) and (b) are

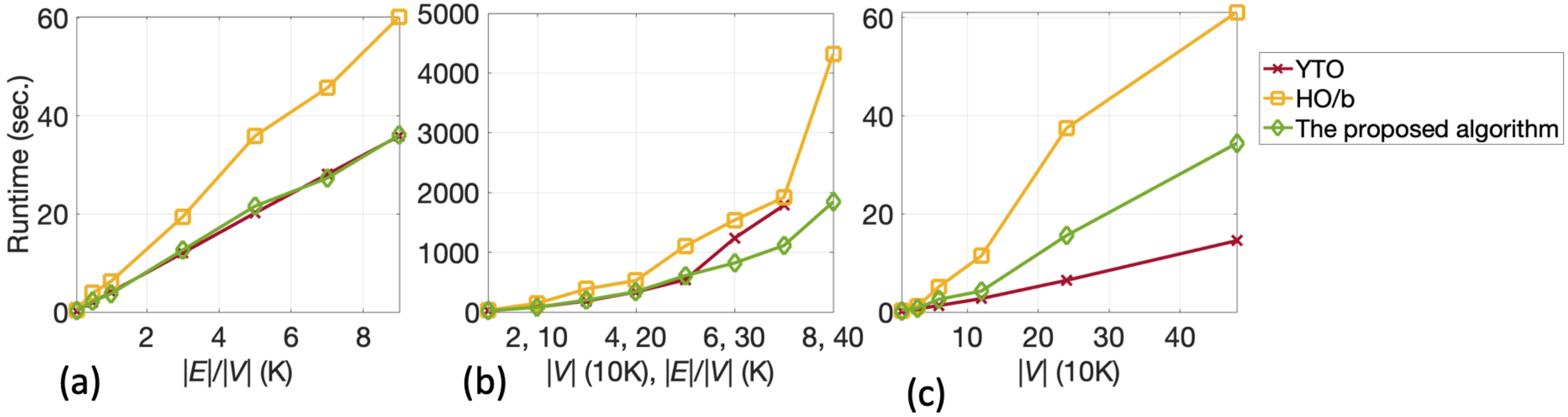


Fig. 2.8.: Comparison of runtime performance of YTO, HO/b and the proposed algorithm on random graphs. The graphs in (a) and (b) are dense. In (a), the graph density $|E|/|V|$ is varied from 50 to 9000 for $|V| = 10\text{K}$. The proposed algorithm performs better than HO/b and is comparable to YTO as 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 2.1.: Information of graphs derived from IWLS 2005 benchmark circuits [20].

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

considered dense graphs. The proposed algorithm performs better than HO/b and is comparable to YTO. The absence of data for YTO indicates that YTO has failed to execute because of excessive memory consumption. Fig. 2.8(c) compares runtimes of YTO, HO/b 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.

2.6.4 Effectiveness of efficient early termination techniques

In Table 2.3, we show the improvements resulted from the filtering techniques in efficient π calculation and $\lambda_{\min}$ calculation. The improvements are based on the comparisons of two variants of the proposed MCM algorithm against HO/b. The first variant considers only the efficient π calculation and the second variant considers only the efficient $\lambda_{\min}$ calculation. All two variants perform early termination check when

Table 2.2.: Runtimes of YTO, HO/b and the proposed algorithm on graphs derived from IWLS 2005 benchmark circuits [20].

Circuit name	Runtime (ms)		
	YTO	HO/b	The proposed algorithm
$s15850$	0.914	1.121	1.189
$s5378$	2.014	2.153	2.010
$s13207$	1.826	1.878	1.859
$s38584$	9.491	18.947	11.069
$s35932$	8.520	15.854	5.912
$s38417$	23.297	32.049	23.816
$des3$	0.568	0.710	0.885
fht	0.678	0.717	0.722
vga_enh	3.938	4.304	3.951
aes_cipher	13.230	39.394	11.055
fpu	19.383	88.454	18.173
$usbf$	16.583	18.126	17.176
$ac97\_ctrl$	18.112	226.274	13.832
$pci\_bridge32$	26.339	28.158	26.205
dma	52.762	58.518	50.664
eth	74.668	73.604	70.638

Table 2.3.: Improvements (%) in efficient π calculation and efficient $\lambda_{\min}$ calculation for dense random graphs.

$\|V\|$	$\|E\|$	% reduction in #of paths in π calculation	% reduction in traversal length in $\lambda_{\min}$ calculation
10000	50000000	82.064	49.637
20000	200000000	72.759	40.897
30000	450000000	87.561	58.872
40000	800000000	79.426	55.444
50000	1250000000	64.392	49.588
60000	1800000000	83.633	50.559
70000	2450000000	47.072	86.256
80000	3200000000	46.218	61.169

k is a power of 2. On average over six set of random graphs, the filtering technique reduces total number of paths considered for computation of the dual vector π by

70.39%. Efficient $\lambda_{\min}$ calculation reduces total traversal length in cycle detection by 56.55%.

In reference to the discussion in Section 2.5, we now present the observed time complexities of efficient $\lambda_{\min}$ calculation and efficient π calculation. For the early termination technique in HO/b, 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 Table 2.4 and Table 2.5 are results from the first variant when applied to graphs in Figs. 2.7(a) and (b), respectively. Three columns of results are included in a block: the number of paths considered for π calculation, the traversal length for cycle detection in $\lambda_{\min}$ calculation, and the number of rows computed. The middle blocks of results in Table 2.4 and Table 2.5 are results from the second variant. The right most blocks of results in Table 2.4 and Table 2.5 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 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 Table 2.4 and Table 2.5, K'' is less than K' by 26.93% and 32.38%, respectively.

The numbers of paths considered for π calculation and the traversal length when the early termination check is performed when the row number is a power of two are about $2K'|V|$, which matches the expectation. They are about $K''^2|V|/2$ for the early termination check from HO/b, performed at every row, which again matches the expectation. However, when we perform efficient early termination check of the proposed MCM algorithm at every row, the number of paths considered for π calculation can be as high as $5.0K''|V|$ in Table 2.4 and $4.7K''|V|$ in Table 2.5. The average number of paths considered is about $2.4K''|V|$. In other words, the average time complexity of efficient π calculation is indeed $O(|V|)$.

Similarly, the traversal length for cycle detection in efficient $\lambda_{\min}$ calculation can be as high as $12.7K''|V|$ in Table 2.4 and $27.1K''|V|$ in Table 2.5. The average traversal length is about $8.2K''|V|$. Again, the average time complexity of efficient $\lambda_{\min}$ calculation is indeed $O(|V|)$.

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.

2.7 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 tree in the graph when λ is $-\infty$ and then maintaining it as λ increases towards 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, 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. 2.4). The value of N in the proposed algorithm is smaller than in HO algorithm because of the use of pseudo-source (see Section 2.5.3) as that removes the term L_{sp} effectively from N. We define K'' to be the number of rows to be

Table 2.4.: Comparing three versions of early termination on the graphs from Fig. 2.7(a) for the number of paths considered for π calculation (#paths), the traversal length for cycle detection in $\lambda_{\min}$ calculation (length), and the number of rows computed (#rows).

Graph		Early termination		Efficient early termination
		power of two	every row	every row w/ filtering
$\lvert V\rvert(K)$	$\lvert E\rvert(K)$	#paths, length, #rows	#paths, length, #rows	#paths, length, #rows
10	500	2.580000e+06, 2.525380e+06, 129	3.234000e+07, 3.157544e+07, 79	9.501040e+05, 4.658284e+06, 79
10	5000	1.290000e+06, 1.260000e+06, 65	6.240000e+06, 5.940505e+06, 34	5.291290e+05, 2.799510e+05, 34
10	10000	2.580000e+06, 2.540000e+06, 129	5.880000e+07, 5.777105e+07, 107	5.389126e+06, 1.355420e+07, 107
10	30000	1.240000e+06, 1.260000e+06, 65	1.700000e+07, 1.652289e+07, 57	2.051447e+06, 2.266274e+06, 57
10	50000	1.290000e+06, 1.260000e+06, 65	1.530000e+07, 1.484503e+07, 54	1.027588e+06, 2.189516e+06, 54
10	70000	1.290000e+06, 1.260000e+06, 65	1.316000e+07, 1.274704e+07, 50	1.123183e+06, 2.468851e+06, 50
10	90000	1.290000e+06, 1.260000e+06, 65	1.075000e+07, 1.034901e+07, 45	1.940204e+06, 1.628681e+06, 45

Table 2.5.: Comparing three versions of early termination on the graphs from Fig. 2.7(b) for the number of paths considered for π calculation (#paths), the traversal length for cycle detection in $\lambda_{\min}$ calculation (length), and the number of rows computed (#rows).

Graph		Early termination		Efficient early termination
		power of two	every row	every row w/ filtering
$\lvert V\rvert(K)$	$\lvert E\rvert(K)$	#paths, length, #rows	#paths, length, #rows	#paths, length, #rows
10	50000	6.400000e+05, 6.200000e+05, 33	5.220000e+06, 4.955019e+06, 31	1.584540e+05, 2.084400e+05, 31
20	200000	2.580000e+06, 2.520000e+06, 65	2.842000e+07, 2.754990e+07, 52	1.071292e+06, 7.670400e+05, 52
30	450000	1.545000e+07, 1.530000e+07, 257	4.776300e+08, 4.725751e+08, 177	1.236244e+07, 7.421342e+07, 177
40	800000	2.060000e+07, 2.040000e+07, 257	5.409600e+08, 5.346201e+08, 163	4.229690e+06, 1.766382e+08, 163
50	1250000	1.290000e+07, 1.270000e+07, 129	2.724000e+08, 2.677750e+08, 103	2.442102e+07, 6.135449e+07, 103
60	1800000	1.548000e+07, 1.524000e+07, 129	1.892400e+08, 1.848303e+08, 78	4.613551e+06, 4.624026e+07, 78
70	2450000	9.030000e+06, 8.820000e+06, 65	1.035300e+08, 1.001350e+08, 53	7.694204e+06, 9.618850e+06, 53
80	3200000	4.120000e+07, 4.080000e+07, 257	9.052000e+08, 8.939600e+08, 149	4.587966e+07, 2.246421e+08, 149

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.

Fig. 2.9 shows how T, K', and K'' scale when the graph density and graph size are changed. The result has been normalized against the respective average values. It is evident that the proposed algorithm has comparable or better scaling profile than YTO. While scaling profiles of HO and the proposed algorithms appear to be comparable, the average value of K'' is much smaller than K'.

2.8 Conclusions

Memory usage and runtime performance of Karp's MCM algorithm can be improved with early termination technique from Hartmann and Orlin, 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 HO/b algorithm consumes much less memory compared to YTO. But when it comes to runtime, YTO performs better than HO/b. We have proposed several techniques to improve the early termination check in the HO/b algorithm. These improvements allow the efficient early termination check to be performed in $O(|V|)$ time complexity on the average empir-

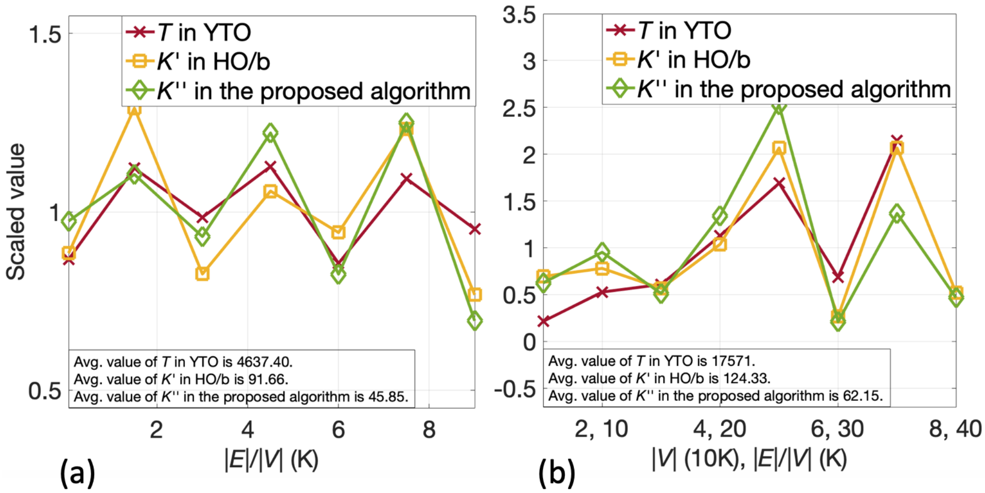


Fig. 2.9.: K'' (the proposed algorithm) has comparable or better scaling profile than T (YTO) for increasing graph density (a), and increasing size of dense graph (b). The plots are obtained by scaling the actual number for K', K'' and T with their respective average values. Average value of K'' is smaller than K'. No result is reported for YTO at (80K, 40K) since the algorithm fails to run to completion.

ically. Such efficiency allows the early termination check to be performed at every row to reduce the cost of vertical relaxation, which has a time complexity of $O(|E|)$. Consequently, the proposed algorithm has better runtime performance than HO/b and produces comparable results to YTO for circuit graphs. For random graphs, the proposed algorithm has better runtime performance than HO/b and is comparable to YTO as the graph becomes denser, all these while improving memory usage of the HO/b algorithm.

3. MINIMUM BALANCE ALGORITHMS

Minimum balancing of a directed graph is an application of the minimum cycle mean algorithm. Minimum balance algorithms have been used to optimally distribute slack for clock network design [11] [12] [5] [13]. In conventional minimum balance algorithms [13] [14] [3], there are two main subroutines. First subroutine runs the MCM algorithm to calculate the minimum cycle mean and also to find the corresponding minimum-mean cycle. The second subroutine updates the entire graph by changing the edge weights and reducing the graph size. These operations run in iterations. The algorithm terminates when the updated graph is a single node. The graph update operation, which has a time-complexity of $\Theta(V+E)$ [11], has been considered to be a bottleneck in the conventional minimum balance algorithms [14]. We first discuss these algorithms in detail.

3.1 Conventional minimum balancing

We show the flow of the conventional minimum balance algorithms in Fig. 3.1(a) [14] [11] [3]. Two main subroutines in the algorithm are the calculation of MCM and the graph update operation.

There are several MCM algorithms [1] [18] [2] [14] that can be used in a minimum balance algorithm. Schneider and Schneider [3] [40], for example, used a modified version of Karp's minimum cycle mean algorithm [1] to solve a closely related problem, that of maximum balancing of a graph. Both the Karp-Orlin [2] and Young-Tarjan-Orlin [14] algorithms can not only find the minimum cycle mean, but be extended to solve the minimum balance problem as well [14]. In particular, the Young-Tarjan-Orlin algorithm (referred to as YTO), has been used to solve the slack distribution problem [11] [12] [5] [13] [15] [16].

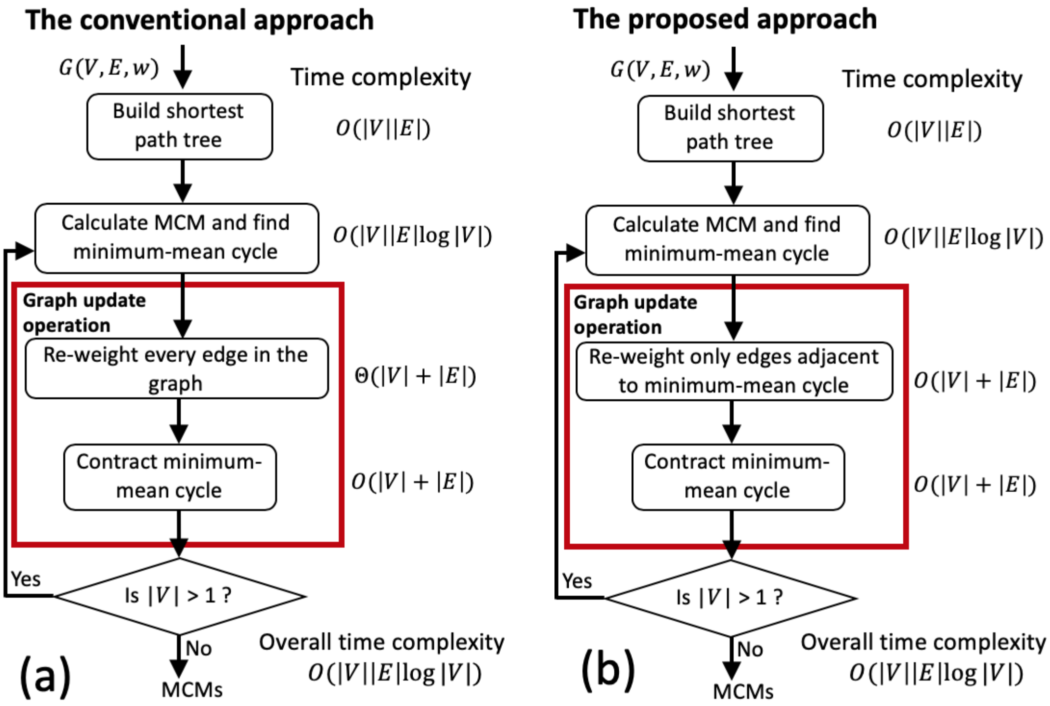


Fig. 3.1.: (a) The conventional and (b) the proposed minimum balance algorithms. The proposed algorithm does not require re-weighting of the entire graph as is the case with [14] [11] [3]. It re-weights only edges adjacent to the minimum-mean cycle, thereby reducing time complexity of the graph update from $\Theta(|V| + |E|)$ to $O(|V| + |E|)$.

After obtaining the MCM of a graph $G(V, E, w)$, the graph update operation involves two steps [14] [11]. Let λ^* be the MCM of the graph $G(V, E, w)$. In the first step, every edge in G is re-weighted by downshifting its weight by λ^* to obtain a new graph $G(V, E, w - \lambda^*)$. Further adjustments are made on the incoming and outgoing edges of the minimum-mean cycle using shortest path distances (a byproduct of the MCM subroutine). Schneider and Schneider presented in [3] a different first step for re-weighting *all* edges in the graph by using the shortest path distances. The first step of the graph update operation has a time complexity of time $\Theta(|V| + |E|)$ [11]; it has been considered a bottleneck in the conventional minimum balance algorithm [14].

The second step of the graph update operation contracts the minimum-mean cycle to reduce the graph size. Moreover, during cycle contraction, some redundant edges may appear. These redundant edges can be removed from the graph. Cycle contraction and redundant edge removal reduce graph size considerably. As a consequence, subsequent iterations of MCM calculation and graph update run progressively faster. As the cycle contraction and redundant edge removal involves only edges adjacent to the minimum-mean cycle, the time complexity is $O(|V| + |E|)$.

In this work, we present an improvement to the conventional minimum balance algorithms [14] [11] [3] where the re-weighting by downshifting as in [14] [11] or the re-weighting by using the shortest path distances as in [3] involving the entire graph is not required. Instead our algorithm updates only edges adjacent to the minimum-mean cycle, as shown in Fig. 3.1(b). The graph update operation involves the entire graph only in the worst case. Consequently, there is improvement in efficiency.

In the remainder of this section, we present the MCM calculation subroutine and the graph update operations in a conventional minimum balance algorithm. We shall present in Section 3.2 the proposed approach.

3.1.1 MCM calculation

For solving the MCM problem in a minimum balance algorithm, there are several options. Karp used dynamic programming to calculate MCM exactly in $\Theta(|V||E|)$ time [1]. With the early termination technique introduced by Hartmann and Orlin [18], Karp's algorithm can compute MCM in $O(|V||E|)$ time.

Young-Tarjan-Orlin [14] (YTO) algorithm solved a series of parametric shortest path problem for MCM calculation. Based on an observation that $\lambda^* = \text{MCM}$ is the largest (edge parameter) $\lambda(e)$ for which $G(V, E, w - \lambda^*)$ 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 $\lambda(e)$ such that the shortest-path tree changes by only one edge. Note that the shortest path tree is defined based on the graph $G(V, E, w - \lambda(e))$. The algorithm terminates when a cycle of zero weight is detected in the shortest path tree. A binary heap based implementation of YTO with a time complexity $O(|V||E|\log|V|)$ has been reported to be the fastest MCM algorithm in practice [7] [36].

The implementation of the minimum balance algorithm in [11] using YTO (with a binary heap) as the MCM calculation subroutine has an overall time complexity of $O(|V||E|\log|V|)$. On the other hand, Schneider and Schneider used a modified version of Karp's algorithm [1] to solve the balancing problem in $O(|V|^2|E|)$ time complexity [3]. In this work, we have used YTO as the MCM calculation subroutine to implement the Schneider-Schneider algorithm. Our implementation that runs in time $O(|V||E|\log|V|)$ can be viewed as a faster implementation of the Schneider-Schneider algorithm.

It is important to note that these MCM algorithms produce not just the MCM (λ^*). Each of them also identifies the corresponding minimum-mean cycle in $G(V, E, w)$ (or zero-weight and minimum-mean cycle in $G(V, E, w - \lambda^*)$), as well as the vector of shortest distances of $G(V, E, w - \lambda^*)$. The MCM, the corresponding minimum-mean cycle, and the shortest distance vector all play an important role in the graph update operation.

3.1.2 Graph update operation in [3]

The graph update operation involves two steps: the re-weighting of edges and the contraction of a minimum-mean cycle.

3.1.2.1 Re-weighting of edges

Schneider and Schneider's algorithm [3] re-weights every edge in the graph using shortest path distances of the downshifted graph $G(V, E, w - \lambda^*)$, as outlined in the pseudocode **re-weight_$\boldsymbol{G}$_using_$\boldsymbol{\pi}$**.

We use $\pi[v]$ to denote the shortest path weight to node v from an arbitrary source node used in the MCM algorithm. The weight of every edge $(u, v) \in E$ in $G(V, E, w)$ (see Fig. 3.2(a)) is adjusted to $w'(u, v) = w(u, v) + \pi[u] - \pi[v]$ in the re-weighting

Re-weight G using π

```
re-weight_G_using_π($G(V, E, w)$, $\pi$)
  for each edge $e = (u, v) \in E$ do
    $w(u, v) = w(u, v) + \pi[u] - \pi[v]$
  end for
  return $G$
end re-weight_G_using_π
```

process, yielding $G'(V, E, w')$, as shown in Fig. 3.2(b). This re-weighting strategy is similar to that of Johnson's algorithm for the shortest path problem [36], where all negative weights are transformed to non-negative ones. The re-weighting operation takes $\Theta(|V| + |E|)$ time.

3.1.2.2 Contraction of minimum-mean cycle

We now elaborate on how the minimum-mean cycle in G and the shortest-path tree in $G(V, E, w - \lambda^*)$, both provided by the MCM algorithm, allow us to contract the cycle to produce a new graph G'' as shown in Fig. 3.2(c). The main purpose of cycle contraction is to reduce the graph size so that the next iteration of MCM calculation and graph update can be more efficient. However, it is important that the re-weighting and cycle contraction do not change the next MCM.

Let us first explain the process of cycle contraction process using Fig. 3.2. All nodes in the minimum-mean cycle C_1 are collapsed into one representative node. The topology of the contracted graph $G''(V'', E'', w')$, where $V'' \subset V$ and $E'' \subset E$, can be easily obtained: all edges that form the cycle are removed, all edges leaving nodes in C_1 now leave the representative node and all edges entering nodes in C_1 now enter the representative node. In Fig. 3.2, $C_1 = v_0 \rightarrow v_1 \rightarrow v_4 \rightarrow v_0$ is the minimum-mean cycle of G and it has been collapsed into the representative node v_0. For example, consider an incoming edge (u, v) to the cycle C in G', the edge becomes (u, c) in G'' with c being the representative node. For an outgoing edge (u, v) from the cycle C in G', the edge becomes (c, v) in G''. The edges (v_1, v_2) and (v_3, v_4) in G' become respectively edges (v_0, v_2) and (v_3, v_0) in G''.

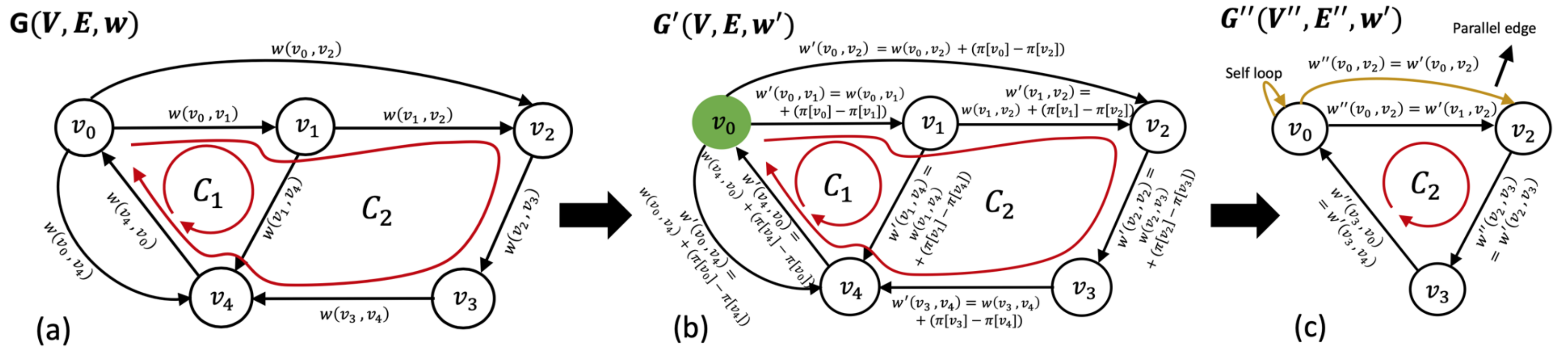

G(V, E, w)
w(v0, v2)
w(v0, v1)
w(v1, v2)
v0
v1
v2
v3
v4
C1
C2
w(v4, v0)
w(v0, v4)
w(v1, v4)
w(v2, v3)
w(v3, v4)
(a)
G′(V, E, w′)
w′(v0, v2) = w(v0, v2) + (π[v0] − π[v2])
w′(v0, v1) = w(v0, v1) + (π[v0] − π[v1])
w′(v1, v2) = w(v1, v2) + (π[v1] − π[v2])
w′(v4, v0) = w(v4, v0) + (π[v4] − π[v0])
w′(v0, v4) = w(v0, v4) + (π[v0] − π[v4])
w′(v1, v4) = w(v1, v4) + (π[v1] − π[v4])
w′(v2, v3) = w(v2, v3) + (π[v2] − π[v3])
w′(v3, v4) = w(v3, v4) + (π[v3] − π[v4])
(b)
G″(V″, E″, w′)
Parallel edge
Self loop
w″(v0, v2) = w′(v0, v2)
w″(v0, v2) = w′(v1, v2)
w″(v3, v0) = w′(v3, v4)
w″(v2, v3) = w′(v2, v3)
(c)

Fig. 3.2.: The graph update operation in [3]. (a) The original graph $G(V,E,w)$ with minimum-mean cycle C_1. (b) The weight of every edge $(u,v) \in E$ in $G(V,E,w)$ is adjusted to $w'(u,v) = w(u,v) + \pi[u] - \pi[v]$ in the re-weighting process, yielding $G'(V,E,w')$, where π is the vector of shortest path distances in $G(V,E,w-\lambda^*)$. (c) Node v_0 on C_1 is the representative node for cycle contraction. All edges in C_1 are removed. After contraction, (v_0,v_4) in G' turns into a self loop (v_0,v_0) in $G''(V'',E'',w')$ and (v_0,v_2) and (v_1,v_2) in G' become parallel edges in G''.

In the process of cycle contraction, (v_0, v_4) in G' has turned into a self loop (v_0, v_0) in $G''(V'', E'', w')$. The edges (v_0, v_2) and (v_1,v_2) on the other hand have turned into parallel edges (v_0,v_2) (Fig. 3.2(c)). Self loops can be removed from the graph (of course, we should record their corresponding cycle means). However, we must keep the parallel edges [14] [11]. Note that if the parallel edges appear in any iteration, they would eventually turn into self loops once the nodes involving those edges have formed a minimum-mean cycle. Again, if we are also interested in the corresponding cycle means, appropriate bookkeeping can be performed to record them. We observe that the removal of self loops from the graph results in speed-up of the minimum balance algorithm. A pseudocode for cycle contraction (with self-loop removal) is presented below. The complexity of the operation is $O(|V| + |E|)$ because only a subgraph induced by the cycle contributes to the computation.

Cycle Contraction

```
cycle_contraction($G(V, E, w)$, $C$, $\pi$, $c$)
  /* remove edges of $C$ and self loops from $G$ */
  for each edge $e(u, v) \in E$, where $u \in V(C)$ and $v \in V(C)$ do
    $E \leftarrow E - e$
  end for
  /* re-direct incoming edges of $C$ in $G$ */
  for each edge $(u, v) \in E$ where $v \in V(C)$ and $v \neq c$ do
    $E \leftarrow E - (u, v)$
    $E \leftarrow E + (u, c)$
  end for
  /* re-direct outgoing edges of $C$ in $G$ */
  for each edge $(u, v) \in E$ where $u \in V(C)$ and $u \neq c$ do
    $E \leftarrow E - (u, v)$
    $E \leftarrow E + (c, v)$
  end for
  /* remove nodes of $C$ from $G$ */
  for each node $v \in V(C)$ and $v \neq c$ do
    $V \leftarrow V - v$
  end for
  return $G$
end cycle_contraction
```

3.1.2.3 Correctness of graph update operation

We will now show that the re-weighting and cycle contraction steps in the Schneider-Schneider algorithm preserve the MCM of the graph, i.e., the next MCM of G is the same as the MCM of G''. We first use the example in Fig. 3.2(c) to illustrate that. Observe that the total edge weight of the cycle C_2 in G'' is

$$w'(C_2) = w(v_1, v_2) + w(v_2, v_3) + w(v_3, v_4) + (\pi[v_1] - \pi[v_4]).$$

Recall that π is the vector of shortest path distances of the graph $G(V, E, w - \lambda^*)$, where λ^* is the cycle mean of the cycle C_1 (i.e. MCM_1). As C_1 is also a minimum-mean cycle and a zero-weight cycle of $G(V, E, w - \lambda^*)$, every pair of consecutive nodes u and v in the cycle satisfies the condition that $\pi[v] = \pi[u] + w(u, v) - \lambda^*$ [1]. Therefore,

$$\pi[v_4] = \pi[v_1] + w(v_1, v_4) - \lambda^*,$$

$$\pi[v_1] - \pi[v_4] = -(w(v_1, v_4) - \lambda^*).$$

Since C_1 is a zero-weight cycle in $G(V, E, w - \lambda^*)$,

$$-(w(v_1, v_4) - \lambda^*) = (w(v_0, v_1) - \lambda^*) + (w(v_4, v_0) - \lambda^*).$$

By replacing $(\pi[v_1] - \pi[v_4])$ in $w'(C_2)$, we obtain

$$\begin{aligned} w'(C_2) &= w(v_1, v_2) + w(v_2, v_3) + w(v_3, v_4) + \\ &\quad (w(v_0, v_1) - \lambda^*) + (w(v_4, v_0) - \lambda^*), \end{aligned}$$

which is the total weight of C_2 in G after deducting λ^* from the weights of edges in C_1 (and dropping the parameters from these edges). In other words, the numerator for MCM_2 remains intact, and that the next MCM does not change as a result of contraction. The correctness of the approach relies on the fact that π is the shortest path distance of the graph $G(V, E, w - \lambda^*)$, where C_1 is a zero weight cycle. However, in the Schneider-Schneider algorithm [3], there is no need to explicitly derive the graph $G(V, E, w - \lambda^*)$; π is a byproduct of the MCM algorithm as we have mentioned earlier.

We now present a more general argument that the graph update operation of the Schneider-Schneider algorithm preserves the MCM of the graph. Again, C_1 is the minimum-mean cycle and it is a zero-weight cycle in $G(V, E, w - \lambda^*)$, where λ^* is the cycle mean of C_1. Let C_2 be any cycle in G. The objective is to show that the re-weighting of edges using π and the contraction of cycle C_1 does not change the total edge weight of C_2.

Let $P_s = \{p_1, p_2, ..., p_k\}$ be the set of shared paths of C_1 and C_2. In Fig. 3.2, there is only one shared path between C_1 and C_2, i.e., $p_1 = v_4 \to v_0 \to v_1$. Let the originating node and ending node of a shared path p_i be v_i^o and v_i^e, respectively. Assume that the shared paths are ordered in the way they appear in C_2, C_2 is of the form $v_1^o \overset{p_1}{\leadsto} v_1^e \leadsto v_2^o \overset{p_2}{\leadsto} v_2^e \leadsto \cdots \leadsto v_k^o \overset{p_k}{\leadsto} v_k^e \leadsto v_1^o$, where for $1 \leq i \leq k$, $v_i^o \overset{p_i}{\leadsto} v_i^e$ is a shared path and $v_i^e \leadsto v_j^o$, where $j = (i \bmod k) + 1$, is not. Note that it is possible that a shared path p_i contains only a single node, i.e., $v_i^o = v_i^e$.

When $|P_s| = 0$, the proof is trivial since the net adjustment of weight by π in C_2 is zero (all π cancel out in a cycle). We now consider the case when $|P_s| \geq 1$. The weight of C_2 in G'' is

$$\begin{aligned} w'(C_2) &= \sum_{\substack{i=1..k \\ j=(i \bmod k)+1}} w'(v_i^e \leadsto v_j^o) \\ &= \sum_{\substack{i=1..k \\ j=(i \bmod k)+1}} \left(w(v_i^e \leadsto v_j^o) + (\pi[v_i^e] - \pi[v_j^o]) \right). \end{aligned}$$

We shall examine the second term in the right hand side of the preceding equation more closely.

$$\begin{aligned} & \sum_{\substack{i=1..k \\ j=(i \bmod k)+1}} (\pi[v_i^e] - \pi[v_j^o]) \\ = \;& \pi[v_1^e] - \pi[v_2^o] + \pi[v_2^e] - \pi[v_3^o] + \cdots + \pi[v_k^e] - \pi[v_1^o] \\ = \;& \sum_{i=1..k} \pi[v_i^e] - \pi[v_i^o]. \end{aligned}$$

Now, we know that a shared path p_i lies on C_1. As C_1 is also a minimum-mean cycle and a zero-weight cycle of $G(V, E, w - \lambda^*)$, every path $p = u \rightsquigarrow v$ in the cycle satisfies the condition that $\pi[v] = \pi[u] + w(p) - |p|\lambda^*$, where $|p|$ is the number of edges in the path p. This is a straightforward extension from the fact that every pair of consecutive nodes u and v in C_1 satisfies the condition that $\pi[v] = \pi[u] + w(u, v) - \lambda^*$ [1]. Therefore, for a shared path p_i,

$$\pi[v_i^e] = \pi[v_i^o] + w(p_i) - |p_i|\lambda^*,$$

$$\pi[v_i^e] - \pi[v_i^o] = w(p_i) - |p_i|\lambda^*.$$

Therefore, the weight of C_2 in G'' can be re-written as

$$w'(C_2) = \sum_{\substack{i=1..k \\ j=(i \bmod k)+1}} w(v_i^e \rightsquigarrow v_j^o) + \sum_{i=1..k} \left(w(p_i) - |p_i|\lambda^*\right).$$

This is of course the weight of C_2 in G after deducting λ^* from all edges in C_1 (and dropping the parameters from these edges). Therefore, any cycle C_2 in G retains the same weight after the re-weighting and cycle contraction steps.

3.1.3 Graph update operation in [14] [11]

Let us now discuss the graph update operation for the algorithm in [14] [11]. As before, the graph update operation involves the two steps of re-weighting of edges and contracting the minimum-mean cycle.

3.1.3.1 Re-weighting of edges

Re-weighting in [14] [11] happens over two phases. In the first phase, every edge in the graph is downshifted by λ^* to obtain $G(V, E, w - \lambda^*)$. We have alluded to the downshifted graph $G(V, E, w - \lambda^*)$ earlier. Here, we provide the the pseudocode

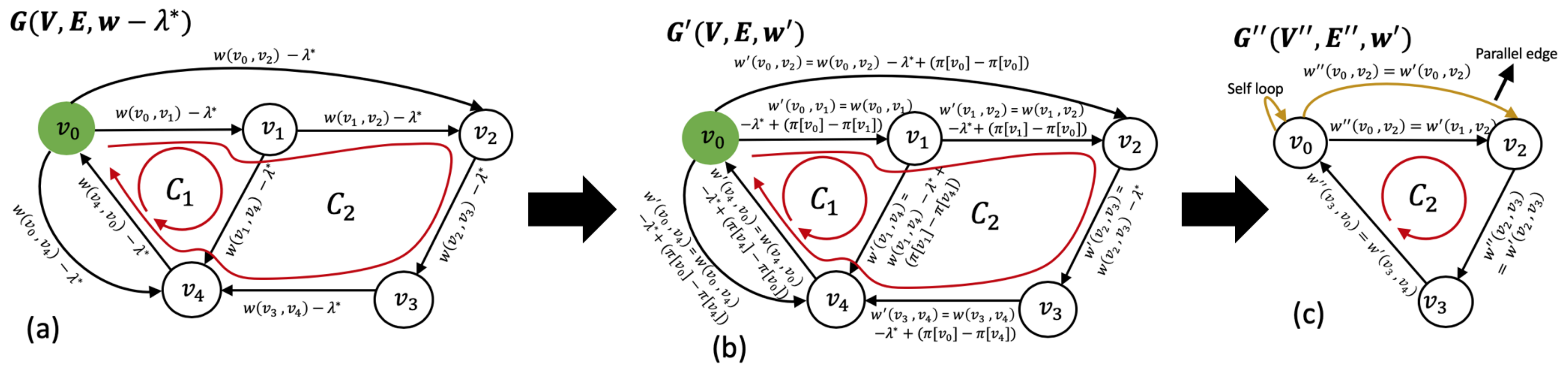

$G(V, E, w - \lambda^*)$
$w(v_0, v_2) - \lambda^*$
$w(v_0, v_1) - \lambda^*$
$w(v_1, v_2) - \lambda^*$
$w(v_0, v_4) - \lambda^*$
$w(v_4, v_0) - \lambda^*$
$w(v_1, v_4) - \lambda^*$
$w(v_2, v_3) - \lambda^*$
$w(v_3, v_4) - \lambda^*$
v_0
v_1
v_2
v_3
v_4
C_1
C_2
(a)
$G'(V, E, w')$
$w'(v_0, v_2) = w(v_0, v_2) - \lambda^* + (\pi[v_0] - \pi[v_0])$
$w'(v_0, v_1) = w(v_0, v_1) - \lambda^* + (\pi[v_0] - \pi[v_1])$
$w'(v_1, v_2) = w(v_1, v_2) - \lambda^* + (\pi[v_1] - \pi[v_0])$
$w'(v_0, v_4) = w(v_0, v_4) - \lambda^* + (\pi[v_0] - \pi[v_4])$
$w'(v_4, v_0) = w(v_4, v_0) - \lambda^* + (\pi[v_4] - \pi[v_0])$
$w'(v_1, v_4) = w(v_1, v_4) - \lambda^* + (\pi[v_1] - \pi[v_4])$
$w'(v_2, v_3) = w(v_2, v_3) - \lambda^*$
$w'(v_3, v_4) = w(v_3, v_4) - \lambda^* + (\pi[v_0] - \pi[v_4])$
(b)
$G''(V'', E'', w')$
Self loop
Parallel edge
$w''(v_0, v_2) = w'(v_0, v_2)$
$w''(v_0, v_2) = w'(v_1, v_2)$
$w''(v_3, v_0) = w'(v_3, v_4)$
$w''(v_2, v_3) = w'(v_2, v_3)$
(c)

Fig. 3.3.: Re-weighting of edges in [14] [11] in two phases. (a) The weight of every edge $(u, v) \in E$ in $G(V, E, w)$ (see Fig. 3.2(a)) is first downshifted by λ^*, i.e., $w(u, v) - \lambda^*$, to obtain $G(V, E, w - \lambda^*)$. (b) Edges adjacent to the minimum-mean cycle (except those on the cycle itself) are then adjusted to yield $G'(V, E, w')$. An edge (u, v) originating from a node on the cycle has an adjustment term of $\pi[u] - \pi[v_0]$, and an edge (u, v) entering a node on the cycle has an adjustment term of $-(\pi[v] - \pi[v_0])$, where π is the vector of shortest path distances defined on $G(V, E, w - \lambda^*)$ and node v_0 is the representative node on the minimum-mean cycle C_1. (c) C_1 in G' is contracted to representative node v_0 to obtain $G''(V'', E'', w')$.

re-weight_*G*_by_downshifting to generate such a graph. For the graph $G(V, E, w)$ in Fig. 3.2, Fig. 3.3(a) shows the downshifted $G(V, E, w - \lambda^*)$.

Re-weight *G* by downshifting

```
re-weight_G_by_downshifting($G(V, E, w)$, $\lambda^*$)
  /* re-weight every edge in $G$ by downshifting */
  for each edge $e = (u, v) \in E$ do
    $w(e) = w(e) - \lambda^*$
  end for
end re-weight_G_by_downshifting
```

Recall that after downshifting of G by λ^*, any minimum-mean cycle in G becomes a zero-weight cycle in $G(V, E, w - \lambda^*)$. Therefore, C_1 in Fig. 3.3(a) is now a zero-weight cycle. The second phase of re-weighting makes further adjustment to the weights of edges adjacent to the minimum-mean cycle so that the cycle can be contracted in the second step of the graph update operation. Let c be a representative node on the minimum-mean cycle C in $G(V, E, w - \lambda^*)$ and G_C be a sub-graph induced by edges that are adjacent to C, including edges on the cycle. Consider an incoming edge (u, v) to the cycle C, the edge weight becomes $w(u, v) - \lambda^* - (\pi[v] - \pi[c])$, where π is the shortest path distance of the downshifted graph (i.e. $G(V, E, w - \lambda^*)$) and c is the representative node of the cycle. For an outgoing edge (u, v) from the cycle C in $G(V, E, w - \lambda^*)$, the edge weight becomes $w(u, v) - \lambda^* + (\pi[u] - \pi[c])$. The second phase of the re-weighting is provided in the following pseudocode **re-weight_G_C_using_π**.

Re-weight G_C using π

```
re-weight_$G_C$_using_$\pi$($G(V, E, w)$, $C$, $\pi$, $c$)
  /* re-weight edges in $G_C$, subgraph induced by $C$, using $\pi$ */
  /* adjust weights of incoming edges of nodes in $C$ */
  for each edge $(u, v) \in E(G_C)$ where $v \in V(C)$ do
    $w(u, v) = w(u, v) - (\pi[v] - \pi[c])$
  end for
  /* adjust weights of outgoing edges of nodes in $C$ */
  for each edge $(u, v) \in E(G_C)$ where $u \in V(C)$ do
    $w(u, v) = w(u, v) + (\pi[u] - \pi[c])$
  end for
  return $G$
end re-weight_$G_C$_using_$\pi$
```

We show the updated edge weights from the second phase of re-weighting in Fig. 3.3(b). For the example in Fig. 3.3(a), $E(G_C)$, the edges in G_C, would consist of C_1 and the following edges: (v_0, v_2), (v_0, v_4), (v_1, v_2), and (v_3, v_4). Here, v_0 is the representative node of C_1. The updated weights of edges such as (v_3, v_4) and (v_1, v_2) are straightforward as these are "incoming" or "outgoing" edges of C_1, respectively, i.e., they are incoming or outgoing edges of some nodes on C_1. For an edge (u, v) that connects two nodes in C_1, the edge weight would be effectively updated to $w'(u, v) = w(u, v) - \lambda^* + \pi[u] - \pi[v]$. Note that the eventual adjustment of $(\pi[u] - \pi[v])$ is similar to the re-weighting performed in the Schneider-Schneider algorithm. The edges in C_1 and the edge (v_0, v_4) are updated in this manner.

Re-weight G by downshifting and using π

```
re-weight_$G$_by_downshifting_and_using_$\pi$($G$, $\lambda^*$, $C$, $\pi$, $c$)
  /* re-weight every edge in $G$ by downshifting */
  $G$ = re-weight_$G$_by_downshifting($G$, $\lambda^*$)
  /* re-weight edges in $G$ around $C$ using $\pi$ */
  $G$ = re-weight_$G_C$_using_$\pi$($G$, $C$, $\pi$, $c$)
  return $G$
end re-weight_$G$_by_downshifting_and_using_$\pi$
```

The pseudocode **re-weight_G_by_downshifting_and_using_π** combines the two phases of re-weighting. Altogether, the re-weighting of edges has $\Theta(|V| + |E|)$ time complexity, same as that of the Schneider-Schneider algorithm [3].

3.1.3.2 Contraction of minimum-mean cycle

The cycle contraction process in [14] [11] is similar to that in the Schneider-Schneider algorithm. In other words, we can use the pseudocode **cycle_contraction** in Section 3.1.2.2. Fig. 3.3(c) shows the graph $G''(V'', E'', w')$ obtained after the graph update operation.

3.1.3.3 Correctness of graph update operation

We now show that the total edge weight of any cycle in G'' is the same as its equivalent cycle in $G(V, E, w - \lambda^*)$. That is because the minimum balance algorithm by [14] [11] will look for the next MCM in $G(V, E, w - \lambda^*)$. On the other hand, the Schneider-Schneider algorithm [3], computes the next MCM in the graph G after the weight of each edge in the minimum-mean cycle has been reduced by λ^* and its parameter dropped from the edge. While the edge weights are modified differently in [3], the approach we take to prove the correctness of the graph update operation by [14] [11] is remarkably similar.

As an example, we look at C_2 in $G(V, E, w - \lambda^*)$ and its equivalent in G'' in Fig. 3.3. The total weight of the equivalent C_2 in G'' is

$$\begin{aligned} w'(C_2) &= w(v_1, v_2) - \lambda^* + w(v_2, v_3) - \lambda^* + w(v_3, v_4) - \lambda^* + \\ &\quad (\pi[v_1] - \pi[v_4]). \end{aligned}$$

We have shown in Section 3.1.2.3 that

$$(\pi[v_1] - \pi[v_4]) = w(v_0, v_1) - \lambda^* + w(v_4, v_0) - \lambda^*.$$

Therefore, $w'(C_2)$ can be re-written as

$$\begin{aligned} w'(C_2) &= w(v_1, v_2) - \lambda^* + w(v_2, v_3) - \lambda^* + w(v_3, v_4) - \lambda^* + \\ &\quad w(v_0, v_1) - \lambda^* + w(v_4, v_0) - \lambda^*, \end{aligned}$$

which is the weight of C_2 in $G(V, E, w - \lambda^*)$. This example illustrates that the cycle weight of C_2 in $G(V, E, w - \lambda^*)$ is preserved under the graph update operation.

We now present a more general argument that the graph update operation preserves the MCM of the graph $G(V, E, w - \lambda^*)$. Again, C_1 is the minimum-mean cycle and it is a zero-weight cycle in $G(V, E, w - \lambda^*)$, where λ^* is the cycle mean of C_1. Let C_2 be any cycle in G. Let $P_s = \{p_1, p_2, ..., p_k\}$ be the set of shared paths of C_1 and C_2. Recall the notation that the originating node and ending node of a shared path p_i are v_i^o and v_i^e, respectively. Assume that the shared paths are ordered in the way they appear in C_2, C_2 is of the form $v_1^o \overset{p_1}{\rightsquigarrow} v_1^e \rightsquigarrow v_2^o \overset{p_2}{\rightsquigarrow} v_2^e \rightsquigarrow \cdots \rightsquigarrow v_k^o \overset{p_k}{\rightsquigarrow} v_k^e \rightsquigarrow v_1^o$, where for $1 \leq i \leq k$, $v_i^o \overset{p_i}{\rightsquigarrow} v_i^e$ is a shared path and $v_i^e \rightsquigarrow v_j^o$, where $j = (i \bmod k) + 1$, is not. Again, a shared path p_i may contain only a single node, i.e., $v_i^o = v_i^e$.

When $|P_s| = 0$, the proof is trivial since all the edges in C_2 undergo only the first phase of the re-weighting. Therefore, the weight of C_2 in G'' is $w'(C_2) = w(C_2) - |C_2|\lambda^*$, which is the weight of C_2 in $G(V, E, w - \lambda^*)$.

We now consider the case when $|P_s| \geq 1$. The weight of C_2 in G'' is

$$w'(C_2) = \sum_{\substack{i=1..k \\ j=(i \bmod k)+1}} w'(v_i^e \rightsquigarrow v_j^o).$$

The path $v_i^e \rightsquigarrow v_j^o$, where $1 \leq i \leq k$ and $j = (i \bmod k) + 1$, originates from a node in C_1 and ends at a node in C_1. The second phase of re-weighting affects only the first and last edge of the path. Therefore, the weight of this path in G'' is

$$\begin{aligned} & w'(v_i^e \rightsquigarrow v_j^o) \\ = \quad & w(v_i^e \rightsquigarrow v_j^o) - |v_i^e \rightsquigarrow v_j^o|\lambda^* + \pi[v_i^e] - \pi[c] - (\pi[v_j^o] - \pi[c]) \\ = \quad & w(v_i^e \rightsquigarrow v_j^o) - |v_i^e \rightsquigarrow v_j^o|\lambda^* + (\pi[v_i^e] - \pi[v_j^o]), \end{aligned}$$

where $|v_i^e \rightsquigarrow v_j^o|$ is the number of edges in the path and c is the representative node of C_1. Hence, the weight of C_2 in G'' is

$$\begin{aligned} & w'(C_2) \\ = \quad & \sum_{\substack{i=1..k \\ j=(i \bmod k)+1}} \left(w(v_i^e \rightsquigarrow v_j^o) - |v_i^e \rightsquigarrow v_j^o|\lambda^* + (\pi[v_i^e] - \pi[v_j^o]) \right). \end{aligned}$$

As shown in Section 3.1.2.3, the last term in the right hand side of the preceding equation can be expressed as

$$\sum_{\substack{i=1..k \\ j=(i \bmod k)+1}} (\pi[v_i^e] - \pi[v_j^o]) = \sum_{i=1..k} \pi[v_i^e] - \pi[v_i^o].$$

We have also shown in Section 3.1.2.3 that for a shared path p_i,

$$\pi[v_i^e] - \pi[v_i^o] = w(p_i) - |p_i|\lambda^*.$$

Therefore, the weight of C_2 in G'' can be re-written as

$$\begin{aligned} w'(C_2) &= \sum_{\substack{i=1..k \\ j=(i \bmod k)+1}} \big(w(v_i^e \rightsquigarrow v_j^o) - |v_i^e \rightsquigarrow v_j^o|\lambda^*\big) \\ &\quad + \sum_{i=1..k} \left(w(p_i) - |p_i|\lambda^*\right), \end{aligned}$$

which is the weight of C_2 in $G(V, E, w-\lambda^*)$. Therefore, any cycle C_2 in $G(V, E, w-\lambda^*)$ retains the same weight after the graph update operation.

One should now realize that the second phase of re-weighting that is performed for edges adjacent to the minimum-mean cycle in [14] [11] is in fact to account for the removal of edges from the graph in cycle contraction. Therefore, re-weighting and cycle contraction indeed have no effect on the subsequent MCM of the graph except to reduce size of the problem that the MCM subroutine has to solve. Therefore, as cycle contraction is necessary to reduce the problem size, so is the re-weighting of edges adjacent to the minimum-mean cycle to apply the cycle contraction. Although the algorithm in [3] does not perform any additional re-weighting around the minimum-mean cycle, the re-weighting of the entire graph using the shortest distance vector implicitly takes care of the changes necessary for cycle contraction to happen.

3.1.4 Implementations

3.1.4.1 The Schneider-Schneider algorithm [3]

We first provide the pseudocode (**Conventional Minimum Balance Algorithm**) of the Schneider-Schneider minimum balance algorithm [3] below. Although any of the algorithms from [1] [2] [18] [41] [14] can be used for MCM calculation

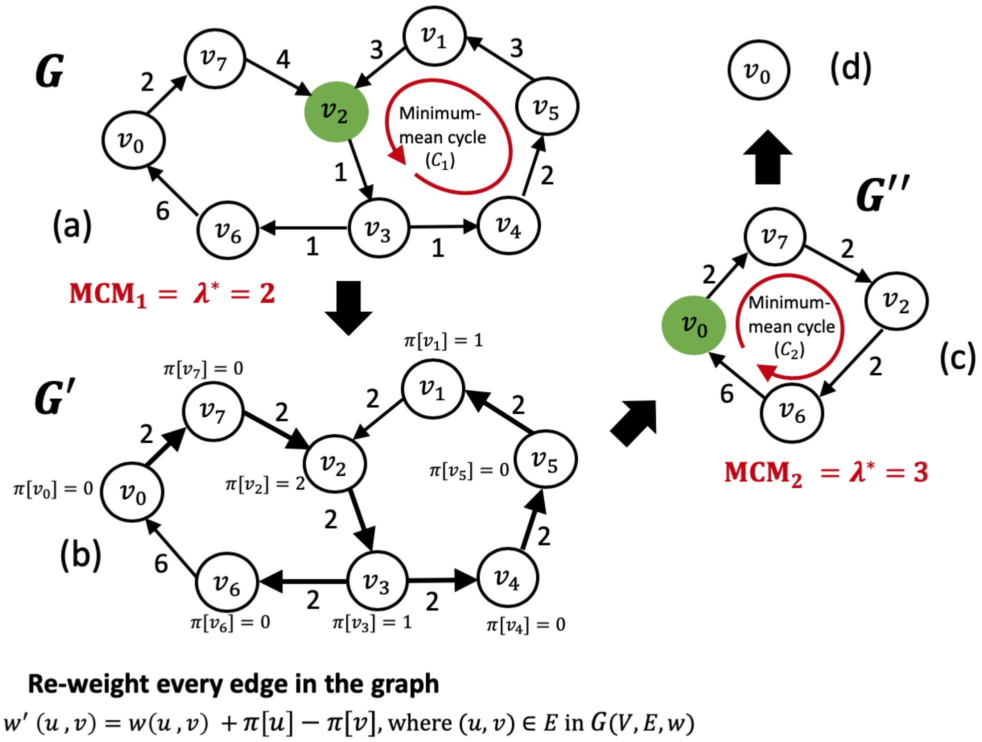

G
(a)
Minimum-mean cycle (C_1)
MCM$_1$ = λ^* = 2
G′
(b)
$\pi[v_0] = 0$
$\pi[v_1] = 1$
$\pi[v_2] = 2$
$\pi[v_3] = 1$
$\pi[v_4] = 0$
$\pi[v_5] = 0$
$\pi[v_6] = 0$
$\pi[v_7] = 0$
G″
(c)
Minimum-mean cycle (C_2)
MCM$_2$ = λ^* = 3
(d)
Re-weight every edge in the graph
$w'(u,v) = w(u,v) + \pi[u] - \pi[v]$, where $(u,v) \in E$ in $G(V,E,w)$

Fig. 3.4.: An illustration of the Schneider-Schneider minimum balance algorithm [3]. Representative nodes are colored in green. v_0 is the source node. The algorithm uses shortest distance vector π in $G(V, E, w - \lambda^*)$ to re-weight all edges. A minimum-mean cycle is collapsed into its representative node.

(referred to as **mcm_algorithm**), we have used YTO algorithm [14] [37] in our implementation. Our implementation of YTO (pseudocode not shown) is similar to that provided in [7] [38] [19], which uses binary heap. With the use of such an YTO implementation, it is possible to limit the overall time complexity of the minimum balance algorithm to $O(|V||E| \log |V|)$ because there are at most $|V|^2$ shortest path trees to be explored in total [14]. If Karp's algorithm [1] or the Hartmann-Orlin algorithm [18] is used, the complexity will be $O(|V|^2|E|)$ because the cardinality of the node set is reduced by at least 1 after each iteration of MCM calculation.

Again, we emphasize that the MCM algorithm returns three important parameters: the MCM λ^*, the minimum-mean cycle C, and the shortest distance vector π. Recall that π is defined in terms of the downshifted graph (i.e. $G(V, E, w - \lambda^*)$).

Conventional Minimum Balance Algorithm

```
   Input  : Directed graph $G(V, E, w)$, a strongly connected component (SCC)
   Output: MCMs
1  /* initialize*/
2  $itr = 0$
3  $\mathrm{MCM}_{itr} = 0$
4  /* run iterative process */
5  while $|V| > 1$ do
6    /* run MCM algorithm to compute MCM $\lambda^*$, minimum-mean cycle C,
7    shortest distance vector $\pi$ in $G(V, E, w - \lambda^*)$, and representative node $c$ */
8    $[\lambda^*, C, \pi, c]$ = mcm_algorithm($G$)
9    $\mathrm{MCM}_{itr+1} = \lambda^*$
10   $G$ = re-weight_G_using_π($G$, $\pi$) /* re-weight $G$ */
11   $G$ = cycle_contraction($G$, $C$, $\pi$, $c$) /* contract cycle $C$ */
12   $++itr$
13 end while
```

Moreover, we assume that the MCM algorithm also returns c, a representative node of the minimum-mean cycle.

We demonstrate through an example the Schneider-Schneider algorithm. A graph G is shown in Fig. 3.4(a). The minimum-mean cycle in G is C_1 and MCM_1 (i.e., λ^*) is 2. The MCM algorithm also yields the shortest distance vector π of $G(V, E, w - \lambda^*)$. The weight of every edge $(u, v) \in E$ in $G(V, E, w)$ is adjusted to $w(u, v) + \pi[u] - \pi[v]$ in the re-weighting process, as shown in Fig. 3.4(b). After contracting C_1 onto the representative node v_2, we get G'', as shown in Fig. 3.4(c). The minimum-mean cycle (C_2 with $\lambda^* = 3$) is identified again in the new graph. Note that the true MCM (i.e. MCM_2) in Schneider-Schneider algorithm is obtained directly (see line 9 of the pseudocode).

3.1.4.2 Minimum balance algorithms in [14] [11]

Let us now demonstrate through an example how the minimum balance algorithm in [14] [11] works and present necessary changes to the pseudocode **Conventional Minimum Balance Algorithm** presented earlier. A graph G is shown in Fig. 3.5(a). The minimum-mean cycle in G is C_1 and MCM_1 (i.e., λ^*) is 2. Downshifting the edge weights in $G(V, E, w)$ by λ^* yields the graph $G(V, E, w - \lambda^*)$. Further adjustments using π are made on $G(V, E, w - \lambda^*)$ for the edges adjacent to the nodes in the cycle

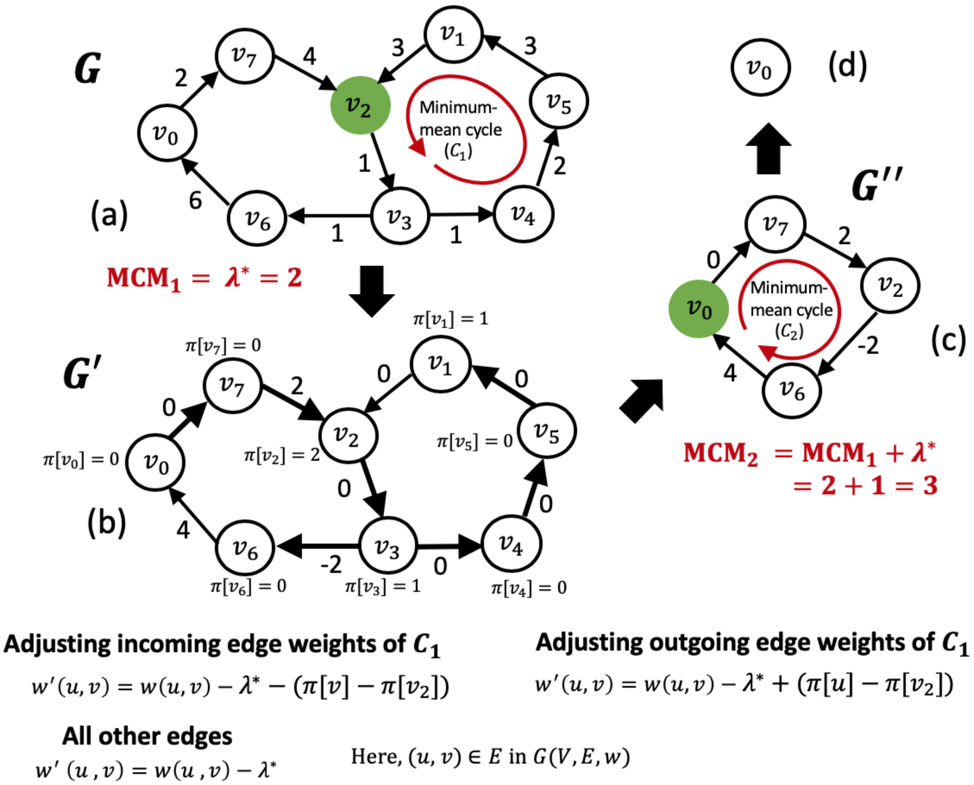

G
(a)
v_0
v_1
v_2
v_3
v_4
v_5
v_6
v_7
Minimum-mean cycle (C_1)
$MCM_1 = \lambda^* = 2$
G′
(b)
$\pi[v_0] = 0$
$\pi[v_1] = 1$
$\pi[v_2] = 2$
$\pi[v_3] = 1$
$\pi[v_4] = 0$
$\pi[v_5] = 0$
$\pi[v_6] = 0$
$\pi[v_7] = 0$
G″
(c)
Minimum-mean cycle (C_2)
$MCM_2 = MCM_1 + \lambda^* = 2 + 1 = 3$
(d)
Adjusting incoming edge weights of C_1
$w'(u,v) = w(u,v) - \lambda^* - (\pi[v] - \pi[v_2])$
Adjusting outgoing edge weights of C_1
$w'(u,v) = w(u,v) - \lambda^* + (\pi[u] - \pi[v_2])$
All other edges
$w'(u,v) = w(u,v) - \lambda^*$
Here, $(u,v) \in E$ in $G(V,E,w)$

Fig. 3.5.: An example to illustrate the minimum balance algorithm in [14] [11]. Representative nodes are colored in green. v_0 is the source node. The algorithm re-weights all edges first by downshifting the weight by λ^* and then adjusts the incoming and outgoing edge weights of the minimum-mean cycle using shortest path distance π of $G(V, E, w - \lambda^*)$. A minimum-mean cycle is collapsed into its representative node.

C_1 to get G', as shown in Fig. 3.5(b). After contracting C_1 onto the representative node v_2, we get G'' (Fig. 3.5(c)). The minimum-mean cycle (C_2 with $\lambda^* = 1$) is identified again in the new graph. Since the MCM is calculated on the downshifted graph G'', the true MCM_2 of C_2 is obtained by adding together the current λ^* and MCM_1. Therefore, our implementation of the algorithms in [14] [11] replaces lines 9−10 of the pseudocode **Conventional Minimum Balance Algorithm** with the following two lines.

```
$\text{MCM}_{itr+1} = \text{MCM}_{itr} + \lambda^*$
$G$ = re-weight_$G$_by_downshifting_and_using_$\pi$($G$, $\lambda^*$, $C$, $\pi$, $c$).
```

3.1.5 Data structures for cycle contraction

In our implementation, a graph $G(V, E, w)$ is implemented using two adjacency lists; for each node u, we maintain a list of outgoing edges $(u, v) \in E$ and a list of incoming edges incoming edges $(v, u) \in E$. There are two reasons for such an implementation. First, the state-of-the-art implementation of the YTO algorithm uses incoming and outgoing adjacency lists [37]. Second, the two adjacency lists facilitate efficient cycle contraction.

We shall elaborate on the steps involved in handling the outgoing edges of the cycle; the incoming edges of the cycle can be handled similarly. First, we go through the outgoing adjacency list of each node, except the representative node, on the minimum-mean cycle. Each (destination) node in these outgoing adjacency lists are stored in a temporary list if it is not a node on the cycle (as we do not want to create self loops). Second, for each unique (destination) node in the temporary list, we process its incoming adjacency list by replacing all (source) nodes that are on the cycle by the representative node. Third, we traverse the outgoing adjacency list of the representative node. All (destination) nodes that are in the cycle are removed from the list. Last, we concatenate the temporary list with the trimmed outgoing adjacency list to form the new outgoing adjacency list of the representative node.

We observe that such an implementation is very effective in practice. Overall, the cycle contraction step runs several times faster than the re-weighting step. This strengthens our argument that the bottleneck in the graph update operation as stated in [14] is indeed due to the re-weighting of *all* edges in the graph.

3.1.6 Floating point arithmetic in re-weighting

In the two implementations of the minimum balance algorithm we have presented, i.e., [3] and [14] [11], the steps of MCM calculation and cycle contraction are identical and incur similar runtime cost. However, the two implementations incur different runtime overheads in the re-weighting step.

The re-weighting of $w(u, v)$ to $w(u, v) + \pi[u] - \pi[v]$ in [3] involves one subtraction and one addition operation. However, there is only one subtraction operation in the re-weighting of $w(u, v)$ to $w(u, v) = w(u, v) - \lambda^*$ by downshifting in [14] [11]. Therefore the re-weighting step in [3] is twice (i.e., $4|E|$, considering two lists) as costly as the cost of downshifting (i.e., $2|E|$, considering two lists) in [14] [11].

However, because of the additional arithmetic operations for re-weighting edges adjacent to the minimum-mean cycle C using π, the actual cost of re-weighting in [14] [11] is $2(|E| + 2|E(G_C)|)$ (considering two lists), where $E(G_C)$ is the set of edges in G_C. Since $|E|$ is in general much larger than $|E(G_C)|$, the algorithm in [3] is likely to perform the graph update operation slower than the algorithm in [14] [11] as the cycle contraction operation is identical in these algorithms. Of course, we can save a little computational cost by not re-weighting edges forming the cycle since we are removing them in the cycle contraction step. However, we can still conclude that the Schneider-Schneider minimum balance algorithm [3] is likely to be slightly less efficient than the algorithm in [14] [11].

3.2 Proposed algorithm

The flow of our proposed approach to minimum balancing is shown in Fig. 3.1(b). The proposed minimum balance algorithm does not require re-weighting of the entire graph as opposed to the algorithms in [3] [14] [11]. However, the re-weighting process of edges adjacent to the minimum-mean cycle in our algorithm follows the same strategy as in the second phase of the re-weighting step in the minimum balance algorithm from [14] [11].

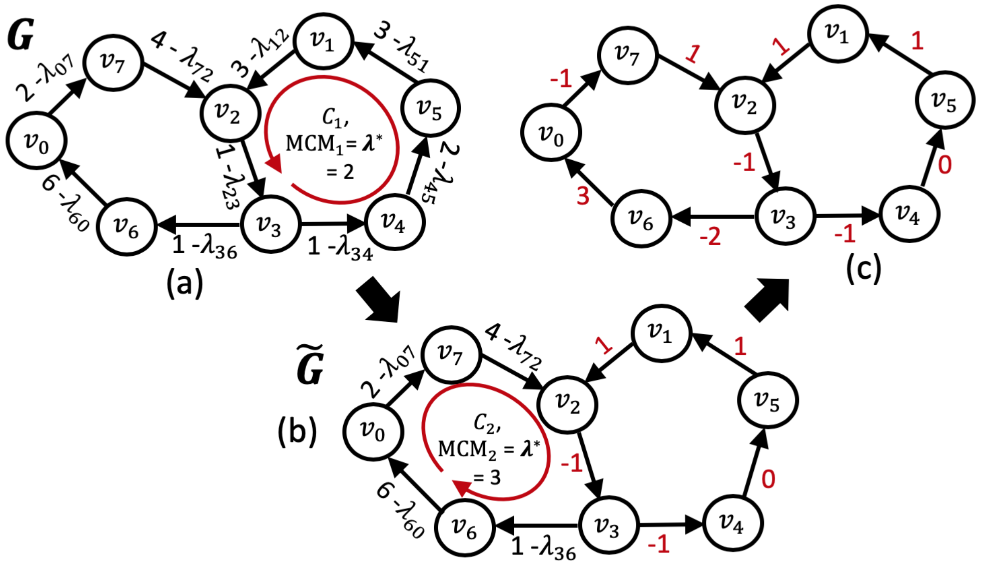


Fig. 3.6.: (a) $\lambda(e)$ for $e \in E(C_1)$ is 2. (b) Edges of the cycle C_1 are downshifted by 2 and the parameters $\lambda(e)$ are removed. (c) The edges with parameters in C_2 are downshifted and their parameters are removed to form a minimum-balanced graph.

We will explain the proposed method in Fig. 3.6 using the same example in Fig. 4.8. In this example, the first MCM is 2 and the first minimum-mean cycle is C_1. We downshift all edges in C_1 by the MCM and remove their parameters $\lambda(e)$. We refer to such a partially downshifted graph as $\tilde{G} = G(V, E, w(C) - \lambda^*)$, where C is the minimum-mean cycle and λ^* is minimum cycle mean. In contrast, a fully downshifted graph is $G(V, E, w - \lambda^*)$. For example, Fig. 3.6(b) shows $\tilde{G} = G(V, E, w(C_1) - 2)$. (We have alluded to $\tilde{G}$ earlier in Section 3.1.2.3 when we presented the correctness of the graph update operation in the Schneider-Schneider algorithm.)

For the cycle C_2 in Fig. 3.6(b), the total weight is 12. Although there are five edges, only four of them have parameters. Therefore, the mean of the cycle is $\frac{12}{4} = 3$. Now, we downshift the weights of the edges in the cycle by $\lambda^* = 2$ and drop the parameter As the downshifting operation is limited to only edges with parameters in the minimum-mean cycle, we can obtain the MCM directly, instead of accumulating all MCMs computed in the previous iterations. Therefore, we can apply the MCM algorithm to find the MCM of a graph that is partially downshifted.

However, directly applying the MCM algorithm on $\tilde{G}$ may not be efficient. Now, we shall show that it is possible to perform contraction of zero-weight cycle in $\tilde{G}$. Recall that given G, the MCM algorithm computes the MCM λ^*, its corresponding minimum-mean cycle C, π, the shortest distance vector in $G(V, E, w - \lambda^*)$, and c, a representative node of C. C is again a zero-weight cycle in $\tilde{G}$. Now, we apply the second phase of the re-weighting step in the algorithm from [14] [11] to obtain $\tilde{G}'(V, E, w')$. An incoming edge (u, v) to the cycle C has a new weight $w'(u, v) = w(u, v) - (\pi[v] - \pi[c])$. An outgoing edge (u, v) from the cycle C in $\tilde{G}$ has a new weight $w'(u, v) = w(u, v) + (\pi[u] - \pi[c])$. An edge (u, v) on the cycle has a new weight $w'(u, v) = w(u, v) - \lambda^* + (\pi[u] - \pi[v])$.

Now, we apply the cycle contraction step to $\tilde{G}'$ and obtain $\tilde{G}''(V'', E'', w')$. By a similar argument as in Section 3.1.3.3, we can show that the total edge weight of any cycle in $\tilde{G}''$ has the same weight as its equivalent cycle in $\tilde{G}$. In other words, we can still take advantage of cycle contraction in the proposed minimum balance algorithm. In fact, the construction of $\tilde{G}''$ does not even require the construction of $\tilde{G}$. The re-weighting of edges adjacent to the minimum-weight cycle and cycle contraction can be performed directly on G using the shortest distance vector π in $G(V, E, w - \lambda^*)$, as shown in Fig. 3.7. We reiterate that our algorithm exploits the fact that π in $G(V, E, w - \lambda^*)$ is a byproduct of the MCM algorithm.

The pseudocode of our proposed minimum balance algorithm is therefore similar to the pseudocode **Conventional Minimum Balance Algorithm** presented earlier, except that the call to **re-weight_G_using_π** in line 10 is replaced by a call to **re-weight_G_C_using_π** as follows:

```
10  $G$ = re-weight_$G_C$_using_$\pi$($G$, $C$, $\pi$, $c$).
```

To summarize, the proposed algorithm is similar to the Schneider-Schneider algorithm as in both *implicitly* operates on $\tilde{G}(V, E, w(C) - \lambda^*)$. Consequently, the proposed algorithm calculates the true MCM values iteratively. Therefore, there is

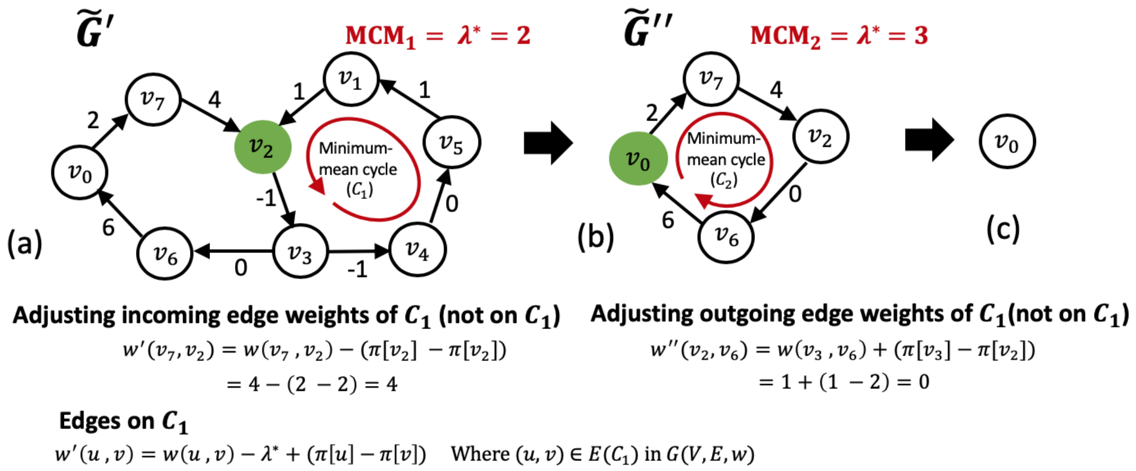

$\widetilde{G}'$
$\mathrm{MCM}_1 = \lambda^* = 2$
v_7
v_1
v_2
v_5
v_0
v_6
v_3
v_4
2
4
1
1
-1
0
6
0
-1
Minimum-mean cycle (C_1)
(a)
$\widetilde{G}''$
$\mathrm{MCM}_2 = \lambda^* = 3$
v_7
v_2
v_0
v_6
2
4
0
6
Minimum-mean cycle (C_2)
(b)
v_0
(c)
Adjusting incoming edge weights of C_1 (not on C_1)
$w'(v_7, v_2) = w(v_7, v_2) - (\pi[v_2] - \pi[v_2])$
$= 4 - (2 - 2) = 4$
Adjusting outgoing edge weights of C_1 (not on C_1)
$w''(v_2, v_6) = w(v_3, v_6) + (\pi[v_3] - \pi[v_2])$
$= 1 + (1 - 2) = 0$
Edges on C_1
$w'(u, v) = w(u, v) - \lambda^* + (\pi[u] - \pi[v])$ Where $(u, v) \in E(C_1)$ in $G(V, E, w)$

Fig. 3.7.: An example to illustrate our minimum balance algorithm. Representative nodes are colored in green. v_0 is the source node. The algorithm uses shortest distance vector π in $G(V, E, w - \lambda^*)$ to adjust incoming and outgoing edge weights of the minimum-mean cycle.

no change to line 9 of the pseudocode **Conventional Minimum Balance Algorithm**. It is implicit because the proposed algorithm (and the Schneider-Schneider algorithm) does not partially downshift the weights of edges in the cycle because the cycle will be contracted. This also avoid the first phase of downshifting all edge weights by λ^* in the algorithm from [14] [11].

To facilitate cycle contraction, the proposed algorithm uses the re-weighting of subgraph G_C as in the algorithm from [14] [11]. This avoids the re-weighting of the entire graph as in the Schneider-Schneider algorithm. Therefore, the graph update operations in the algorithms from [3] [14] [11] are $\Theta(|V| + |E|)$, whereas the graph update operation in the proposed algorithm is $O(|V| + |E|)$.

Given that MCM calculation and cycle contraction operations are identical in the algorithms from [3] [14] [11] as well as the proposed algorithms, the runtime improvement in our algorithm would essentially come from the reduced floating point arithmetic operation in partially re-weighting the graph. There are only $4|E(G_C)|$ (considering two lists) floating point operations in our algorithm as opposed to $4|E|$ and $2(|E| + 2|E(C)|)$ operations, respectively, in [3] and in [14] [11].

Table 3.1.: IWLS 2005 benchmark circuits [20] and details of the largest strongly connected component of each circuit graph.

Benchmark	Circuit name	Clock period (ns)	Largest SCC $\lvert V\rvert$	Largest SCC $\lvert E\rvert$
ISCAS89	$s5378$	0.7	136	1536
	$s13207$	0.7	231	1194
	$s38584$	1	1166	11010
	$s35932$	1	1728	7480
	$s38417$	1	1498	37478
OpenCores	*usbf*	1	1431	24522
	ac97_ctrl	10	1895	18320
	*pci_bridge*32	1	1920	46484
	dma	1	2087	102018
	eth	2	8381	151484

3.3 Experimental results

We have implemented the minimum balance algorithm from [3], the algorithm from [14] [11], and the proposed algorithm in C++. These implementations are evaluated on Intel(R) Xeon(R) CPU E5-2660 2.60 GHz Linux machine with 66GB of RAM, using graphs derived from IWLS 2005 benchmark circuits [20] and strongly connected graphs randomly generated using a technique described in [7] [8] and with SPRAND generator [42].

First, we compare the algorithms using ISCAS89 and OpenCores benchmarks from IWLS 2005 [20]. A timing constraint graph is constructed after the synthesis of a design specified in Verilog using the Synopsys tool chain and a $32nm$ technology library. We decompose the graph into strongly connected components, and apply the algorithms on these individual components. Circuit information along with the largest strongly connected component graph details are given in Table 3.1. The runtime results for the largest strongly connected components are presented in Table 3.2. Here, runtimes in all columns labeled **MCM** include running the YTO algorithm to calculate MCM and the necessary updates to the data structures associated with the

Table 3.2.: Runtime (in sec.) of the three algorithms on IWLS 2005 circuits [20].

Circuit	Schneider-Schneider [3]			The approach in [14] [11]			Our approach		
name	MCM	Update	Total	MCM	Update	Total	MCM	Update	Total
*s*5378	0.004	0.003	0.007	0.004	0.003	0.007	0.004	0.002	0.006
*s*13207	0.005	0.001	0.007	0.004	0.001	0.005	0.005	0.001	0.006
*s*38584	0.068	0.048	0.118	0.069	0.048	0.118	0.070	0.011	0.083
*s*35932	0.094	0.032	0.127	0.078	0.023	0.102	0.078	0.003	0.082
*s*38417	0.355	0.418	0.779	0.350	0.408	0.763	0.347	0.096	0.448
usbf	0.203	0.251	0.456	0.203	0.251	0.457	0.202	0.082	0.286
*ac*97_*ctrl*	0.287	0.237	0.526	0.282	0.224	0.508	0.278	0.066	0.345
*pci_bridge*32	0.473	0.596	1.072	0.474	0.562	1.038	0.470	0.164	0.638
dma	1.504	2.845	4.357	1.518	2.734	4.259	1.515	0.665	2.187
eth	15.849	17.917	33.778	15.326	16.726	32.062	15.443	3.582	19.036
Avg. runtime	1.884	2.235	4.123	1.831	2.098	3.932	1.841	0.467	2.312

Table 3.3.: Runtimes (in sec.) of the three algorithms on random graphs [7].

Graph		Schneider-Schneider [3]			The approach in [14] [11]			Our approach		
$\|V\|$	$\|E\|$	MCM	Update	Total	MCM	Update	Total	MCM	Update	Total
10000	30000	3.257	1.452	4.716	3.210	1.361	4.578	3.232	0.321	3.559
20000	60000	13.195	5.740	18.950	12.963	5.212	18.190	13.031	1.073	14.118
30000	90000	30.703	14.936	45.664	30.287	13.830	44.140	30.432	3.169	33.624
40000	120000	55.658	25.717	81.408	54.786	23.439	78.259	54.388	4.572	58.991
50000	150000	93.465	46.003	139.516	89.048	37.137	126.232	87.396	7.995	95.433
60000	180000	147.557	74.064	221.683	149.871	69.601	219.537	135.735	14.075	149.866
70000	210000	226.286	120.969	347.341	222.806	101.598	324.489	195.880	23.113	219.068
80000	240000	331.813	202.400	534.339	319.921	159.925	479.963	278.718	41.591	320.413
90000	270000	452.257	254.464	706.870	433.099	195.659	628.906	368.983	46.051	415.174
100000	300000	579.680	350.101	929.961	601.449	308.179	909.799	496.542	71.734	568.444
Avg. runtime		193.387	109.585	303.045	191.744	91.594	283.409	166.434	21.369	187.869

Table 3.4.: Runtimes (in sec.) of the three algorithms on SPRAND graphs [42].

Graph		Schneider-Schneider [3]			The approach in [14] [11]			Our approach		
$\|V\|$	$\|E\|$	MCM	Update	Total	MCM	Update	Total	MCM	Update	Total
5442	16377	1.860	0.799	2.664	1.845	0.731	2.581	1.882	0.155	2.044
11055	33308	7.844	3.457	11.314	7.775	3.097	10.883	7.829	0.624	8.467
16288	48817	17.708	8.244	25.972	17.590	7.522	25.131	17.685	1.495	19.202
21749	65224	31.823	14.360	46.211	31.604	13.088	44.718	31.699	2.198	33.928
26898	80677	49.419	21.883	71.338	50.244	21.894	72.172	50.137	3.828	54.004
32100	96437	77.155	39.431	116.632	76.244	35.497	111.785	76.471	8.044	84.565
37489	112517	110.075	56.012	166.145	110.383	54.091	164.535	104.495	11.064	115.618
42826	128446	157.093	87.219	244.387	159.360	80.423	239.858	143.397	14.019	157.492
48281	145049	212.366	120.205	332.663	212.989	110.669	323.752	190.199	21.536	211.831
53562	160787	280.198	169.592	449.910	279.981	134.866	414.957	250.077	31.336	281.523
Avg. runtime		94.554	52.120	146.723	94.802	46.188	141.037	87.387	9.430	96.867

YTO algorithm. The runtimes in all columns labeled **Update** include re-weighting and cycle contraction. On average over ten circuit graphs, our algorithm improves total runtime by 43.92% and 41.20% over the Schneider-Schneider algorithm [3] and the algorithms in [14] [11], respectively.

We also evaluate performance of the algorithms using random graphs created with a technique in [7] [8]. To generate a random graph using the technique in [7] [8], we first connect all the 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 edge connections are made. For the edge weight, random values within a range similar to that in the circuit graphs are generated and assigned. As shown in Table 3.3, our algorithm improves the total runtime. On average over ten set of random graphs (each set having five), the improvement is about 38.01% and 33.71% over the Schneider-Schneider algorithm [3] and the algorithm in [14] [11], respectively.

We also generate random graphs using SPRAND generator [42]. The SPRAND generator cannot ensure that the graph with exact $|V|$ and $|E|$ is created; $|V|$ and $|E|$ of a SPRAND graph may therefore be different from the random graph reported in Table 3.3. On average over ten set of SPRAND generated graphs (each set having five), our algorithm improves total runtime by 33.98% over Schneider-Schneider [3] and 31.32% over the algorithm in [14] [11]. The runtime results for the SPRAND graphs are presented in Table 3.4.

One can notice that the runtimes for **Update** in [14] [11] are somewhat better than those of [3]. This can be attributed to the fewer arithmetic operations that the algorithm in [14] [11] has to perform. This confirms what we presented in Section 3.1.6.

3.4 Conclusions

We have proposed an improvement to conventional minimum balance algorithms by performing fewer re-weighting of edge weights in each iteration, resulting in better efficiency. Performance has been evaluated on circuit graphs and random graphs. We see 42.56% and 34.26% average runtime improvements over state-of-the-art minimum balance algorithms for graphs derived from IWLS 2005 benchmark circuits and randomly generated graphs, respectively.

4. A SCALABLE BUFFER QUEUE SIZING ALGORITHM FOR LATENCY INSENSITIVE SYSTEMS

High performance communication channels are usually required to connect existing intellectual property (IP) cores together in system-on-chips (SoCs). However, as the device feature size scales down to ten's of nanometers, the delay of the global interconnects can limit the system performance. Because of the difficulty in having an accurate estimation of the global interconnect delay in early stages of the design, a large numbers of timing violations may occur in the late stages of the physical design process; design iterations are therefore unavoidable [21] [22] [23].

Latency insensitive design is an SoC design methodology that simplifies the assembly of IP cores by reusing pre-validated and pre-designed IP cores. These IP cores can be either soft macros, i.e., synthesizable logic blocks specified in a hardware description language or hard macros in GDSII format. An implementation of such a system requires the insertion of relay stations for the purpose of pipelining if the delay is longer than the target clock period. It helps design engineers to improve system performance, and also reduces the number of costly design iteration. However, the system throughput can be negatively affected if relay stations are improperly inserted [28] [23] [22] [21] [32].

By having a large number of buffer queues, such performance loss can be eliminated. However, the system may not have adequate area to accommodate these buffers [22]. In [22], the authors solved a buffer queue sizing problem for throughput optimization. By imposing constraints on the number of buffers that could be allowed in specific regions, the formulation took into account important physical design constraints such as the area and routing bottlenecks [45]. A mixed integer linear programming (MILP) formulation was used to solve the optimization problem of buffer queue sizing. While the proposed method was effective in obtaining an optimal solution (within a user-specified level of precision), its runtime was not scalable; there is currently no known polynomial-time algorithm for solving MILP [24].

Several heuristics to address the scalability issue have been proposed to date. In [26] [27] [28], the authors reduced the problem size by collapsing each strongly connected component (SCC) to a node. As the approach did not explore all cycles in a graph, it often led to a final solution that deviated considerably from an optimal solution [26]. Approaches such as those proposed in [29] [30] [31] [32] used some form of integer linear programming (ILP) but made approximations to reduce the problem size. In [30], the authors assumed that the channel latency increased linearly as the buffer capacity was increased, which might not be a reasonable assumption in latest technology nodes. The authors in [46] developed a technique to reduce the number of places buffers must be inserted by formulating the problem as a minimum cost arborescence problem in directed graphs. However, to solve the throughput optimization problem for many reduced size SCCs, they also relied on an MILP formulation similar to that in [22] for every one of those SCCs [31] [32].

In addition to the runtime scalability issue, optimally solving an MILP problem often requires careful formulation strategy and parameter tuning [25]; these could be obstacles to developing a robust methodology. In [33] [34], the authors avoided the issues of queue sizing by scheduling the activation of circuit blocks. Planning for such schedules requires that each circuit block has knowledge about the overall system behavior, posing additional roadblocks to the design process. Moreover, most of the approaches can only deal with small to medium-sized problems because of the dependency on solving an MILP formulation. Some critical physical design constraints such as the limited capacities of buffer regions [22] [45] are also not addressed in most of these approaches.

In this paper, we revisit the problem defined in [22], that of buffer queue sizing for maximizing throughput of a latency insensitive system subject to regional buffer constraints. Instead of solving the original MILP formulation from [22], we propose to solve a simplified buffer queue sizing problem that can be formulated as a parameterized graph optimization problem. In this formulation, for every communication channel there is a parameterized edge with buffer counts as the edge weight. We then use a minimum cycle mean algorithm [1] [2] [14] [18] to determine from which edges buffers can be removed safely. The use of a minimum cycle mean algorithm that is of polynomial-time complexity allows us to overcome the runtime scalability issue of an integer linear program solver. We evaluate the performance of the proposed approach on randomly generated latency insensitive systems of various sizes. In particular, we generate large size examples that represent the near future system complexities [47]. Experimental results suggest that the proposed approach is scalable. The quality of the solutions produced by the proposed approach, in terms of the throughput and the size of buffer queues, is as good as that of the MILP-based approach from [22].

The rest of the paper is organized as follows. In the next two sections, we review the work presented in [22]. In particular, we present graph models of latency insensitive systems in Section 4.1 and a throughput optimization problem through buffer sizing in Section 4.2. We propose a simplified buffer sizing problem in Section 4.3 and present its application to the throughput optimization problem in Section 4.4. Experimental results are detailed in Section 4.5. We conclude the paper in Section 4.6.

4.1 Modeling a latency insensitive system

A system on chip consists of many circuits units that exchange data on the communication channels [22]. Because of the long interconnect delays between circuit units and process variation in them, timing violation is a major issue in latest technology nodes. To take care of the timing violation, wire pipelining is a common strategy in which a long interconnect is partitioned into short segments by inserting relay stations, which are essentially clocked buffers with capacity to store [27]. Fig. 4.1(a) shows a simple system-on-chip made of three circuit units. The long interconnect between units 1 and 3 are divided into two shorter communication channels by a relay station as shown in Fig. 4.1(b). While this eliminates the timing violation in the long interconnect, the latency of the channel between two units may become multiple cycles; mismatch in latency can therefore occur between various units [23] [22] [27] [43].

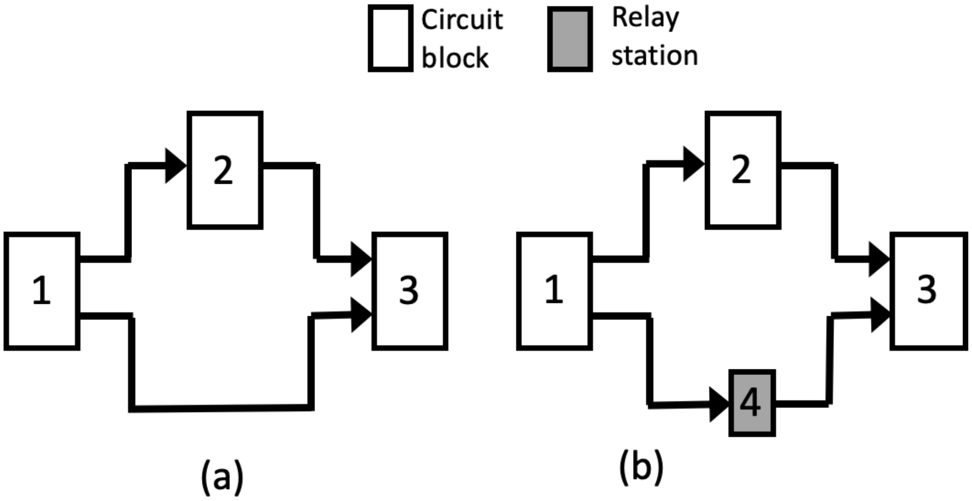


Fig. 4.1.: (a) A simple system-on-chip. Blocks 1, 2 and 3 represent circuit units. Communication channels connect the individual units. The long interconnect between the units 1 and 3 can cause timing violation. (b) A simple wire pipelining strategy that inserts a relay station (unit 4) between units 1 and 3 can solve the timing violation problem. Relay stations are essentially the clocked buffer with storage capacity [27].

Because of the latency mismatch, there can be data losses. The latency insensitive protocol allows a system with arbitrary communication latencies between different units to function properly. It uses a signaling scheme that directs a unit to stall through back pressure [22] [27]. Stallability property can be realized through clock gating for any number of clock cycles while holding its internal state [23] [27]. In a latency insensitive system, data exchanged by the units can be marked as informative (or valid) data and non-informative (or void) data. When any of its input data is not available, a unit is stalled and in that situation, non-informative or void data is supplied to the output port of the stalled unit [23] [22] [27] [43].

We shall use the system in Fig. 4.1(b) to demonstrate the stalling mechanism. Each communication channel is associated with a queue. First, we assume that every channel has a minimum size buffer queue, i.e., it can store only one piece of data at a time. We show a sequence of informative and non-informative data produced by the

Table 4.1.: Progressive trace of the LIS from Fig. 4.1 with minimum size buffer queues, unlimited buffer queues and optimized queues.

Cycle	Circuit/relay station unit											
	Minimum				Unlimited				Optimized			
no.	1	2	3	4	1	2	3	4	1	2	3	4
1	1	1	1	τ	1	1	1	τ	1	1	1	τ
2	2	2	τ	1	2	2	τ	1	2	2	τ	1
3	3	τ	2	2	3	3	2	2	3	3	2	2
4	τ	3	3	3	4	4	3	3	4	4	3	3
5	4	4	4	τ	5	5	4	4	5	5	4	4

units 1, 2, 3, and 4 in the LIS, called progressive trace [23] [22] in Table 4.1 under the block labeled "Minimum queues" for the first five cycles. The integer value i in the table is the i-th piece of informative data being produced by the circuit unit when $(i-1)$-th informative data are fed into it. Relay station samples data from the input channel at every clock cycle and passes it to the output channel. We assume that the primary inputs can always supply data to unit 1 and the primary outputs can always consume data produced by unit 3.

Assume that units 1, 2, and 3 have valid data at their input channels, they produce their first informative data in the first clock cycle. On the other hand, unit 4, as a relay station, does not have informative data in its input channel and therefore provides non-informative data (denoted as τ) to its output channel in the first cycle. Because of the non-informative data in the communication channel from unit 4 to unit 3, unit 3 would be stalled in clock cycle 2. In clock cycle 3, unit 2 is stalled since its output channel is already filled with data generated by the same unit in the previous cycle but not yet consumed by unit 3. This in turn would stall unit 1 in clock cycle 4. In cycle 5, units 1, 2, and 3 produce the 4th piece of data while unit 4 provides non-informative data. The behavior of the units repeats every 4 cycles. The throughput of the LIS with minimum queues is 3/4 or 0.75 as only three sets of informative data are generated in every four clock cycles.

The throughput can be restored to 1 when the communication channels have unlimited or infinite queues. The progressive trace of such a system is shown in Table 4.1 under the block labeled "Unlimited queues". The behavior of the system with unlimited queues is identical to that of the system with minimum queues in the first two clock cycles. However, in clock cycle 3 and beyond, unit 1 and unit 2 would keep producing informative data as their output channels have infinite queue capacity, and units 3 and 4 are always one cycle behind units 1 and 2. Other than the first two clock cycles, the primary outputs receive one piece of informative data every cycle.

Although infinite queues provide a sufficient condition for the throughput of a latency insensitive system to be optimal, it is not necessary to have infinite queues for optimal throughput. For the system in Fig. 4.1(b), the optimal throughput can also be achieved with two queue buffers for the channel between units 2 and 3, while other channels have minimum queues. For the purpose of synchronization, the first stalling of unit 3 effectively screens out the non-informative data forwarded by the relay station. If there is only one buffer in the channel between units 2 and 3, unit 2 must be stalled to prevent loss of data. However by having two buffers in the output channel, unit 2 can now continue to produce informative data. Since unit 2 never stalls, unit 1 also never stalls. Therefore, the behavior of this system is identical to that of the system with unlimited queues, as shown in the progressive trace in Table 4.1 under the block labeled "Optimized queues".

To formally capture the relationship between throughput performance and buffer queue sizing of a latency insensitive system, two graph models have been introduced in [22] [43]. An LIS can be modeled by a lis-graph $G_l(V, E, w)$, which is a weighted connected directed graph where V is the set of all circuit and relay station units. $(v_i, v_j) \in E$ refers to the communication channel from unit v_i to unit v_j, and is called a channel edge. $w(v_i, v_j) \in \{0, 1\}$ and the weights of all outgoing edges of a relay station unit are 1, or equivalently $w(v_i, v_j)$ is 1 if v_i is a relay station and 0 otherwise. Example of an lis-graph is shown in Fig. 4.2.

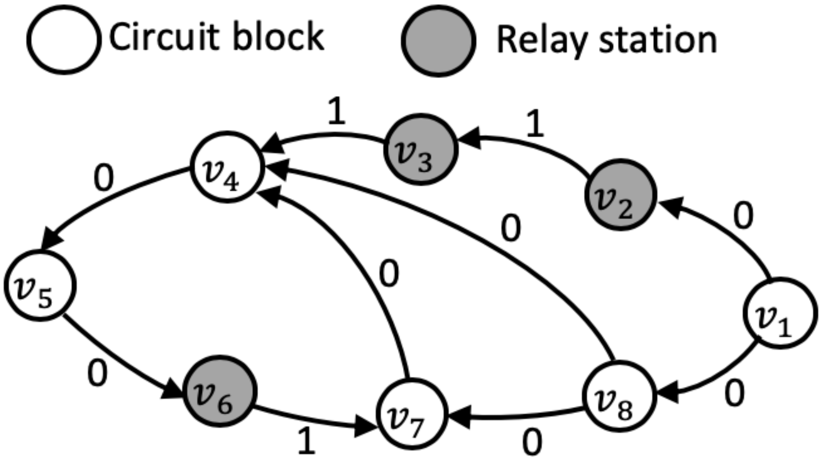


Fig. 4.2.: An example lis-graph $G_l(V, E, w)$ [22]. The graph is constructed by nodes that represent both circuit and relay station units. Directed edges represent communication channels that connect various units. The weights of all outgoing edges of a relay station are 1, for other edges the weight is 0.

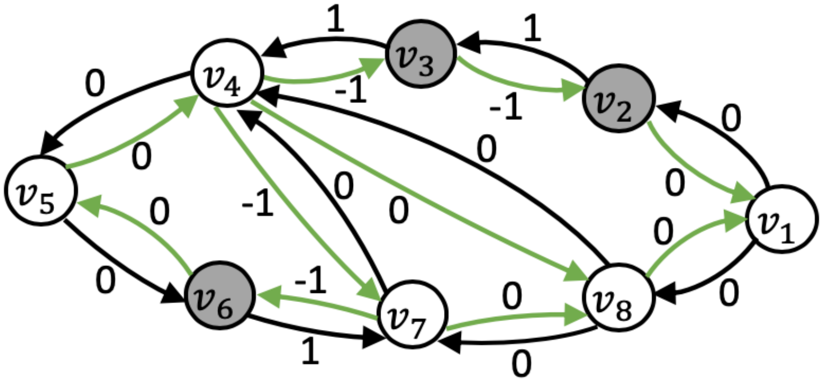


Fig. 4.3.: Extended lis-graph $G_e(V, E_e, w_e)$ for the example lis-graph in Fig. 4.2 [22]. Extended lis-graph is a weighted connected directed graph by adding into its lis-graph $G_l(V, E, w)$ a mirror edge (v_j, v_i) (to represent back pressure) with weight $w(v_j, v_i) = 1 - Q(v_i, v_j) - w(v_i, v_j)$ for every channel $(v_i, v_j) \in E$. $Q(v_i, v_j)$ is the buffer queue size of the communication channel (v_i, v_j). In the example, all channels have minimum number of buffers (i.e. one) except the channel (v_7, v_4), which has two buffers.

To model the queue size of a latency insensitive system, an extended lis-graph [22] is defined as follows. An extended lis-graph $G_e(V, E_e, w_e)$ of an lis-graph $G_l(V, E, w)$

is a weighted connected directed graph by adding into its lis-graph a mirror edge (v_j, v_i) (to represent back pressure) with the following weight.

$$w(v_j, v_i) = 1 - Q(v_i, v_j) - w(v_i, v_j), \tag{4.1}$$

for every channel $(v_i, v_j) \in E$. $Q(v_i, v_j)$ is the buffer queue size of the communication channel (v_i, v_j). For a minimum size buffer queue channel (i.e. one buffer), the mirror edge weight is exactly the opposite of the channel edge weight, in the extended lis-graph. Fig. 4.3 shows an extended lis-graph for the corresponding lis-graph in Fig. 4.2. In the extended lis-graph, all channels have minimum number of buffers except the channel (v_7, v_4), which has two buffers [22].

4.1.1 Throughput of a latency insensitive system

As more relay stations are added, more void data may begin to circulate in the system, thereby degrading the throughput performance. However, the throughput performance can be improved by allocating buffers to hold informative or valid data already generated by the source unit but not yet consumed, as demonstrated with the example in Fig. 4.1. A strong correlation therefore exists between the size of buffer queues and the throughput performance as having a larger queue would allow the system to keep producing the informative data and minimize stalling of circuit blocks. A larger queue, however, comes at the cost of area overhead and routing bottleneck.

In the extended lis-graph G_e, consider $w(C) = \sum_{e \in C} w(e)$ and $\tau(C)$ to denote respectively the 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:

$$\lambda(C) = \frac{w(C)}{\tau(C)}, \tag{4.2}$$

which is the total edge weight of the cycle divided by the number of edges. The maximum cycle mean λ^* of G_e is defined as $\lambda^*(G_e) = \max_{C \in \mathcal{C}}(\lambda(C))$ where $\mathcal{C}$ is the set of all cycles in G. The cycle which has the maximum cycle mean is the maximum-mean cycle. (Note that for a graph that has no cycles, the maximum cycle mean is by definition $-\infty$ [1].) Assuming that there exists some non-negative weight cycle in G_e, the maximum possible throughput of an LIS is [43]

$$1 - \lambda^*(G_e). \tag{4.3}$$

For an LIS with unlimited queues, it is possible that all cycles in G_e are negative. In order to handle such cases, the maximum possible throughput of an LIS can be expressed more generally as:

$$1 - \max(0, \lambda^*(G_e)). \tag{4.4}$$

Computing the maximum cycle mean is very similar to computing the minimum cycle mean, for which many efficient algorithms exist [1] [2] [14] [18].

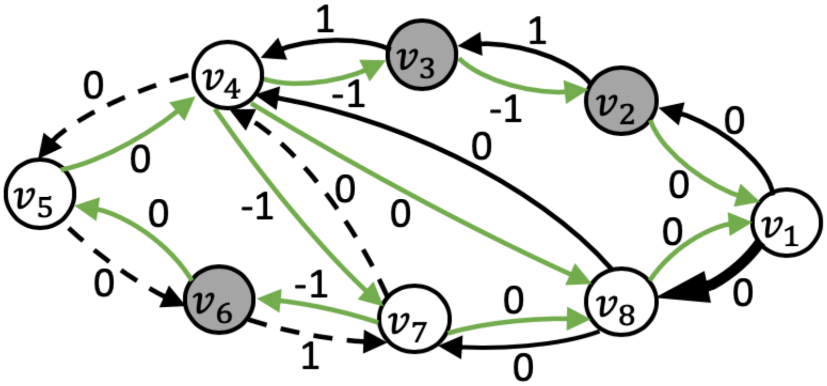


Fig. 4.4.: The cycle C_3 (dashed edges) has no mirror edges. Its cycle mean would therefore limit the throughput of the system [22]. Mean of the cycles C_1 (i.e. $v_1, v_2, v_3, v_4, v_8, v_1$) and C_2 (i.e. $v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_1$) are more than the mean of cycle C_3. Their means can be lowered by increasing number of buffers by one for the channel (v_1, v_8) (bold edge), which is actually the optimal choice for this example.

4.2 Buffer queue sizing for performance optimization

The expression in (4.4) suggests that to improve the throughput performance of an LIS, we may size the buffer queues in the communication channels of an LIS in order to change the weights of the corresponding mirror edges, and therefore reduce the maximum cycle mean of the corresponding G_e. Let us first analyze how far we can push the performance of a latency insensitive system by proper buffer queue sizing.

For an LIS with unlimited queues, we can in fact express its maximum possible throughput in terms of the maximum cycle mean of the corresponding lis-graph as follows:

$$1 - \max(0, \lambda^*(G_l)), \tag{4.5}$$

which is very similar to (4.4). It has been discussed in [22] that the maximum cycle mean of an extended lis-graph can never be smaller than that of the corresponding lis-graph. In other words, no matter how we size the buffer queues in the communication channels of an LIS and therefore change the weights of the mirror edges in its extended lis-graph G_e,

$$1 - \max(0, \lambda^*(G_e)) \leq 1 - \max(0, \lambda^*(G_l)).$$

The maximum cycle mean of the lis-graph therefore imposes a performance limit that could be reached by buffer queue sizing.

To explain this through an example, we redraw Fig. 4.3 in Fig. 4.4 where some of the edges have been highlighted. We also note the following cycles with positive means:

- $C_1 = (v_1, v_2, v_3, v_4, v_8, v_1)$, $\lambda(C_1) = 2/5 = 0.4$;

- $C_2 = (v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_1)$, $\lambda(C_2) = 3/8 = 0.375$;

- $C_3 = (v_4,\, v_5,\, v_6,\, v_7,\, v_4)$, $\lambda(C_3) = 1/4 = 0.25$;
- $C_4 = (v_4,\, v_5,\, v_6,\, v_7,\, v_8,\, v_4)$, $\lambda(C_4) = 1/5 = 0.20$;
- $C_5 = (v_1,\, v_2,\, v_3,\, v_4,\, v_7,\, v_8,\, v_1)$, $\lambda(C_5) = 1/6 = 0.167$.

Out of all the preceding five cycles, C_3 (dashed edges) is the only cycle without mirror edges, i.e., this is a cycle of only channel edges and it is also present in the lis-graph. Therefore, the maximum throughput that one could possibly get from the system is $(1 - 0.25)$ or 0.75. There are two cycles C_1 and C_2 whose means are more than the mean of the cycle C_3. To improve the overall throughput, we must lower the means of these two cycles.

For cycle C_1, we can increase from one to two the buffer numbers of either channel (v_8, v_4) or channel (v_1, v_8) to make $\lambda(c_1) = 0.2$. For the cycle C_2, we can increase from one to two the number of buffers of channel (v_1, v_8) in order to reduce $\lambda(C_2)$ to 0.25. Therefore, to lower both $\lambda(C_1)$ and $\lambda(C_2)$, we can increase either buffer queue sizes of both (v_8, v_4) and (v_1, v_8), which will cost us two buffers, or buffer queue size of only (v_1, v_8), which will cost us only one buffer. That was the motivation for the original work on the problem of buffer queue sizing for throughput performance optimization in [22].

The buffer queue sizing problem in [22] considered two types of constraints, namely, the performance constraints and the physical design constraints. The performance constraints deal with the system throughput that is to be maximized and the physical design constraints are essentially the regional buffer constraints that restrict the number of buffers that can be inserted in the respective regions. We first discuss the performance constraints.

4.2.1 Performance constraints

Going by the definition of maximum cycle mean [3] [1], If the maximum cycle mean (i.e. $\lambda^*(G_e)$ or λ^* in short) is subtracted from every edge in G_e, it would make all cycles negative except the maximum-mean cycles. A maximum-mean cycle turns into a zero-weight cycle after such subtraction.

Since there are no positive cycles in the graph, all longest path distances (from an arbitrary source node) are valid. We can therefore write the following two constraints respectively for a channel edge (v_i, v_j) and its mirror edge (v_j, v_i) [22].

$$r(v_j) - r(v_i) \geq w(v_i, v_j) - \lambda^*, \forall (v_i, v_j) \in E, \tag{4.6}$$

$$r(v_i) - r(v_j) + Q(v_i, v_j) - 1 \geq -w(v_i, v_j) - \lambda^*, \forall (v_i, v_j) \in E. \tag{4.7}$$

Here, $r(v_i)$ and $r(v_j)$ are respectively the longest path distances (from an arbitrary source node) to nodes i and j. By replacing $Q(v_i, v_j) - 1$ with $q(v_i, v_j)$, we re-write (4.7) as follows:

$$r(v_i) - r(v_j) + q(v_i, v_j) \geq -w(v_i, v_j) - \lambda^*, \forall (v_i, v_j) \in E, \tag{4.8}$$

where $q(v_i, v_j)$ is the number of buffers inserted in addition to the minimum buffer size of 1 for channel (v_i, v_j). A graph that represents the longest path constraints in (4.6) and (4.8) is shown in Fig. 4.5.

In a valid queue sizing solution, $q(e)$ for channel edge e is a non-negative integer value, i.e.,

$$q(e) \in Z^{\geq}, \forall e \in E, \tag{4.9}$$

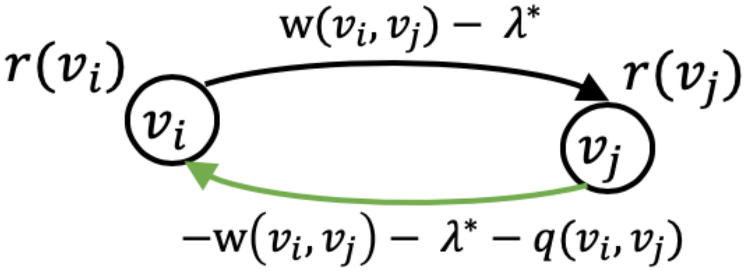

$w(v_i, v_j) - \lambda^*$
$r(v_i)$
$r(v_j)$
v_i
v_j
$-w(v_i, v_j) - \lambda^* - q(v_i, v_j)$

Fig. 4.5.: The graph captures the longest path constraints (4.6) and (4.8). $r(v_i)$ and $r(v_j)$ are respectively the longest path distances to nodes i and j.

where $Z^{\geq}$ is the set of non-negative integers. The constraints in (4.6) and (4.8) involve variables that are integers ($q(e)$) and variables that are real ($r(v)$); these are mixed-integer linear constraints.

4.2.2 Physical design constraints

As the insertion of relay stations is performed in the physical design stage, there may be very little room available for buffer insertion. At this stage of the design, the layout of a system is almost fixed, with only some regions reserved for buffer insertion. Each candidate buffer region has a limited capacity and is shared by several channels that pass through it. This is captured by the following regional buffer constraints [22]:

$$\sum_{m(e)=cb_t} q(e) \leq C(cb_t), \forall cb_t \in CB, \tag{4.10}$$

where CB is the set of candidate buffer regions; $m : E \rightarrow CB$ is the mapping of channel edges to candidate buffer regions, i.e., $m(e) = cb_t$ if channel e passes through candidate buffer region cb_t; and $C(cb_t)$ is the maximum number of buffers that can be inserted in cb_t.

4.2.3 A binary search for the maximum throughput

In the problem of buffer queue sizing for optimizing throughput performance, the objective is to maximize the throughput $(1 - \max(0, \lambda^*))$, which is equivalent to minimize the maximum cycle mean λ^*. The mixed-integer linear program (MILP) formulation is as follows [22]:

$$\text{Minimize } \lambda^*, \tag{4.11}$$

subject to (4.6), (4.8), (4.9), and (4.10).

However, solving this formulation directly with an MILP solver has been found to be very inefficient [22]. Instead, a binary search framework can be used as follows: For every λ^* under consideration in the binary search, the feasibility of the constraints in (4.6), (4.8), (4.9), and (4.10) can be checked using an MILP solver. Instead of just checking for the feasibility of these constraints, the binary search approach in [22] solved the following MILP for each λ^* under consideration in order to minimize the cost of buffering:

$$\text{Minimize } \sum_{e \in E} q(e), \tag{4.12}$$

subject to (4.6), (4.8), (4.9), and (4.10). The approach is detailed in the pseudocode **Binary search for maximum throughput [22]** below.

The binary search for λ^* is performed within a range $[\lambda^*_{\min}, \lambda^*_{\max}] = [\lambda^*(G_l), \lambda^*(G_{e,\min})]$. Here, $\lambda^*(G_l)$ is the minimum possible value of λ^*, which is equivalent to the throughput of the LIS with unlimited buffer queues, and $\lambda^*(G_{e,\min})$ is the maximum possible value of λ^*, which is equivalent to the throughput of the LIS with minimum buffer queues. For every λ^*_{tgt} under consideration in the binary search, we solve for (4.12) subject to the constraints in (4.6), (4.8), (4.9), and (4.10) with an MILP solver. If λ^*_{tgt} is feasible, $\lambda^*_{\max}$ is lowered to λ^*_{tgt} as we have found a higher lower bound of the throughput; otherwise, $\lambda^*_{\min}$ is raised to λ^*_{tgt} as we have found a more realistic up-

Binary search for maximum throughput ([22])

```
Input  : $G_l(V, E, w)$, $CB$, $C(cb_t)\ \forall cb_t \in CB$, $m(e)\ \forall e \in E$, $PREC$
Output: $\lambda^*$
/* Initialization */
$\lambda^*_{\max} = \lambda^*(G_{e,\min})$, $\lambda^*_{\min} = \lambda^*(G_l)$, $\lambda^*_{tgt} = \lambda^*_{min}$
/* Binary search for maximum throughput */
while ($\lambda^*_{\max} - \lambda^*_{\min} \geq PREC$)
  /* Call the MILP solver for buffer queue sizing */
  solve (4.12) subject to (4.6), (4.8), (4.9), and (4.10) for the target $\lambda^*_{tgt}$
  if($\lambda^*_{tgt}$ is feasible)
    $\lambda^*_{\max} = \lambda^*_{tgt}$
  else
    $\lambda^*_{\min} = \lambda^*_{tgt}$
  end if
  $\lambda^*_{tgt} = (\lambda^*_{\max} + \lambda^*_{\min})/2$
end while
$\lambda^* = \lambda^*_{\max}$
```

per bound of the throughput. The binary search terminates when the gap between $\lambda^*_{\max}$ and $\lambda^*_{\min}$ is within the user-specified precision $PREC$. Although the pseudocode does not show it, a feasible buffer sizing solution is a by-product of the optimization process.

4.2.4 A minor improvement

One may realize that it is only important to solve the MILP formulation (4.12) subject to (4.6), (4.8), (4.9), and (4.10) exactly when there is chance for the λ^*_{tgt} under consideration to be feasible. We propose a minor improvement to the binary search framework as follows. Instead of solving an MILP, we solve a relaxed linear program (LP) (4.12), subject to (4.6), (4.8), (4.10), and $q(e) \geq 0\ \forall e \in E$, where the last constraint replaces the integer constraint (4.9). If the relaxed LP formulation is not feasible for the λ^*_{tgt} under consideration, the tighter MILP formulation is also infeasible. Only when the relaxed LP formulation is feasible for the λ^*_{tgt} under consideration, we attempt to solve the MILP formulation to check for its feasibility. Compared to an MILP formulation, an LP formulation can be solved in polynomial time [24]. The pseudocode **Binary search for maximum throughput (our implementation**

of [22]) below incorporates this improvement. Again, a feasible buffer sizing solution is a by-product of the optimization process.

Binary search for maximum throughput (our implementation of [22])

```
Input  : $G_l(V, E, w)$, $CB$, $C(cb_t)\ \forall cb_t \in CB$, $m(e)\ \forall e \in E$, $PREC$
Output: $\lambda^*$
/* Initialization */
$\lambda^*_{\max} = \lambda^*(G_{e,\min})$, $\lambda^*_{\min} = \lambda^*(G_l)$, $\lambda^*_{tgt} = \lambda^*_{min}$
/* Binary search for maximum throughput */
while ($\lambda^*_{\max} - \lambda^*_{\min} \geq PREC$)
  /* Call the LP solver for feasibility check */
  check feasibility of $\lambda^*_{tgt}$ by solving for (4.12) subject to (4.6), (4.8), and (4.10) with $q(e) \geq 0\ \forall\ e \in E$
  if($\lambda^*_{tgt}$ is feasible)
    /* Call the MILP solver for buffer queue sizing */
    solve (4.12) subject to (4.6), (4.8), (4.9), and (4.10) for the target $\lambda^*_{tgt}$
    if($\lambda^*_{tgt}$ is feasible)
      $\lambda^*_{\max} = \lambda^*_{tgt}$
    else
      $\lambda^*_{\min} = \lambda^*_{tgt}$
    end if
  else
    $\lambda^*_{\min} = \lambda^*_{tgt}$
  end if
  $\lambda^*_{tgt} = (\lambda^*_{\max} + \lambda^*_{\min})/2$
end while
$\lambda^* = \lambda^*_{\max}$
```

4.3 Proposed buffer queue sizing algorithm

In the solution by [22] presented in Section 4.2.3, the main bottleneck lies in solving the MILP formulation. There is no known polynomial time algorithm for solving MILP [24]. In this work, we propose to solve the buffer queue sizing problem indirectly by solving a graph optimization problem. It involves solving a different form of buffer queue sizing problem, which is related to a graph optimization problem.

4.3.1 A new form of buffer queue sizing problem

Let us first present this new form of buffer sizing problem. We re-write (4.8) as follows:

$$r(v_i) - r(v_j) \geq -w(e) - \lambda^* - q(e), \forall e = (v_i, v_j) \in E. \tag{4.13}$$

In this new form of buffer sizing problem, we assume that each channel e allows at most $q_m(e)$ buffers on it, in addition to the minimum queue size of 1 buffer. In other words, $q(e) \leq q_m(e)$. We can therefore include $q_m(e)$ in (4.13) as follows:

$$r(v_i) - r(v_j) \geq -w(e) - \lambda^* - q_m(e), \forall e = (v_i, v_j) \in E. \tag{4.14}$$

(4.14) basically captures the fact that at the outset we want to assign maximum number of buffers to every channel e. As we are in general more familiar with a shortest path formulation, we also convert the longest path formulation to a shortest path formulation by negating (4.6) and (4.14) to obtain

$$r(v_i) - r(v_j) \leq -w(e) + \lambda^*, \forall e = (v_i, v_j) \in E, \tag{4.15}$$

$$r(v_j) - r(v_i) \leq w(e) + \lambda^* + q_m(e), \forall e = (v_i, v_j) \in E. \tag{4.16}$$

The variable $r(v)$ now denotes the shortest path distance to node v instead of the longest path distance from an arbitrary source node. A graph that represents the shortest path constraints in (4.15) and (4.16) is shown in Fig. 4.6. As we have negated (4.6) and (4.14) the directions of the edges are opposite of those in Fig. 4.5. In other words, an edge (v_i, v_j) in Fig. 4.5 (and in G_e) becomes an edge (v_j, v_i) in Fig. 4.6.

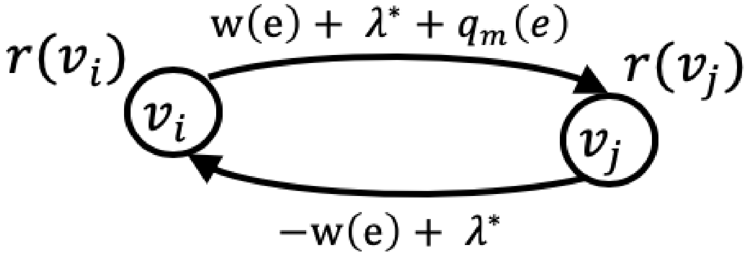


Fig. 4.6.: The graph captures the shortest path constraints (4.15) and (4.16). $r(v_i)$ and $r(v_j)$ are respectively the shortest path distances to nodes i and j. The directions of the edges are opposite to those in G_e and Fig. 4.5.

In this new form of buffer queue sizing problem, we want to remove as many buffers from each channel edge e. Let $\lambda_p(e)$, $0 \leq \lambda_p(e) \leq q_m(e)$ and $\lambda_p(e) \in Z^{\geq}$, denotes the number of buffers we may remove from channel edge e.

$$r(v_j) - r(v_i) \leq w(e) + \lambda^* + q_m(e) - \lambda_p(e), \forall e = (v_i, v_j) \in E. \tag{4.17}$$

In the new buffer queue sizing problem, we want to maximize $\sum_{e \in E} \lambda_p(e)$, subject to the constraints that $0 \leq \lambda_p(e) \leq q_m(e)$, $\lambda_p(e) \in Z^{\geq}$, and shortest path constraints (4.15) and (4.17).

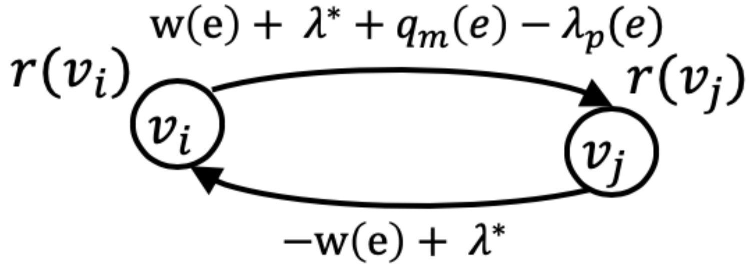


Fig. 4.7.: The parameterization of the graph in Fig. 4.6 by including a parameter $\lambda_p(e)$ on the mirror edge of channel edge e. The parameter $\lambda_p(e)$ is to be maximized in the minimum balance problem.

4.3.2 A parametric graph and its minimum balance

We can view the graph that models the shortest path constraints (4.15) and (4.17) (Fig. 4.7) as a parameterization of the graph that models the shortest path constraints (4.15) and (4.16) (Fig. 4.6). Every mirror edge of a channel edge e has a parameter $\lambda_p(e)$ that is to be optimized.

For a given throughput λ^* and the maximum buffer queue sizes of all communication channel $q_m(e)$, $\forall e \in E$, let $G_{pe}^{(\lambda^*,q_m)}(V, E_{pe}, w_{pe}, E_{upe}, w_{upe})$ denote the parametric graph obtained from a lis-graph $G_l(V, E, w)$. For every channel edge (v_i, v_j) in E, there is a parameterized edge (v_i, v_j) in E_{pe} that has an associated weight $w_{pe}(v_i, v_j) = w(v_i, v_j) + \lambda^* + q_m(v_i, v_j)$ and a parameter $\lambda_p(v_i, v_j)$, such that the parameterized weight of the edge is $w_{pe}(v_i, v_j) - \lambda_p(v_i, v_j)$. It also has a un-parameterized edge (v_j, v_i) in E_{upe} that has an associated weight $w_{upe}(v_j, v_i) = -w(v_i, v_j) + \lambda^*$.

For such a parametric graph, the problem of finding the minimum balance of the graph is related to the new buffer queue sizing problem presented in the preceding subsection. The minimum balance problem is that of finding $\lambda_p(e)$, $e \in E_{pe}$, by simultaneously maximizing $\lambda_p(e)$ for all edges without creating any negative cycle [3] [14].

Consider a generic parametric graph (not derived from a lis-graph) in Fig. 4.8(a), where every edge e has a numeric weight and a parameter λ_e. To find the minimum balance of the graph, i.e., the maximum λ_e for each edge e without inducing a negative cycle, we find a minimum-mean cycle (the cycle that has the minimum cycle mean or MCM) and assign the minimum cycle mean to all edges in that cycle. For a parametric graph, the mean of a cycle is obtained by dividing the sum of edge weights by the number of parameterized edges in the cycle, and the MCM is the minimum of all cycle means in the graph. The minimum cycle mean of $\frac{1+2+3+3+1}{5} = 2$ is from cycle C_1. Therefore, $\lambda(e) = 2$ for $e \in E(C_1)$. Otherwise, C_1 would turn negative for larger λ_e. Now, all edges have their weights reduced by 2. Moreover, all edges in C_1 now lose their parameters λ_e since they have been found (see Fig. 4.8(b)).

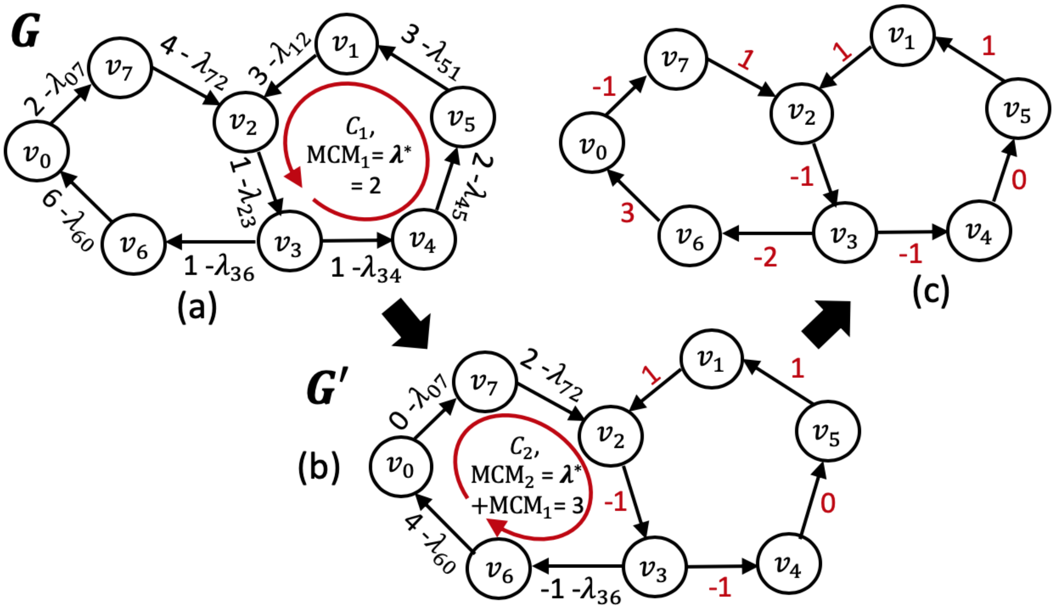


Fig. 4.8.: (a) Every edge in the graph has a parameter λ_e. Largest value of λ_e for $e \in E(C_1)$ is 2, since beyond that value cycle C_1 is negative. (b) Given that $\lambda_e = 2$ for $e \in E(C_1)$, largest value of λ_e for $e \in E(C_2)$ and $\notin E(C_1)$ is 3, since beyond that value cycle C_2 is negative. (c) Minimum-balanced graph.

Next, we find the next minimum-mean cycle C_2, which has a minimum cycle mean of $\frac{4+0+2+(-1)+(-1)}{4} = 1$; the denominator is 4 because four out of the five edges in C_2 are parameterized. Since all the edge weights have been reduced by 2 earlier because of C_1, $\lambda_e = 3$ for $e \in E(C_2)$ and $\notin E(C_1)$. Fig. 4.8(c) shows the minimum-balanced graph where every edge (e) has got its weight downshifted by λ_e (from its original numeric weight).

The minimum balance problem is related to the new buffer queue sizing problem as follows. First, a valid solution to that problem, $\lambda_p(e), \forall e \in E$ must satisfy the shortest path constraints (4.15) and (4.17). In other words, a feasible assignment of $\lambda_p(e)$ must not introduce negative cycles in the corresponding graph in Fig. 4.7. A minimum balance solution of the parametric graph $G_{pe}^{(\lambda^*, q_m)}$ meets that requirement. Second, the minimum balance solution of the parametric graph $G_{pe}^{(\lambda^*, q_m)}$ simultaneously maximizes $\lambda_p(e)$. If $\lambda_p(e)$ is non-negative and not larger than $q_m(e)$, the number of buffers allocated to the channel edge e, i.e., $q_m(e) - \lambda_p(e)$ is minimized.

However, a conventional minimum balance solution of the parametric graph $G_{pe}^{(\lambda^*,q_m)}$ may not have integer values and do not fall in the valid range $[0, q_m(e)]$. This calls for an integer variant of the minimum balance solution for the new buffer queue sizing problem.

4.3.3 An integer variant of minimum balance: Iterative reduction of buffer queue sizes

As illustrated in Fig. 4.8, a conventional process of performing minimum balance of a parametric graph is an iterative approach that involves two main operations [3] [14]. In each iteration, the first operation computes the minimum cycle mean [1] [2] [14]. The second operation then subtracts the minimum cycle mean from all parameterized edges in the graph. As a minimum-mean cycle is now a zero-weight cycle, all parameterized edges in a minimum-mean cycle are un-parameterized. The iterative process continues until all edges are without parameters.

In our integer-variant minimum balance algorithm, we still compute the MCM. However, our goal is not to reduce a minimum-mean cycle to zero-weight cycle but rather to remove buffers iteratively as much as possible from every channel without turning any cycle negative. Let λ_c be the MCM computed in the current iteration and $q_r(e)$ is the number of buffers left for the channel e. Note that the initial value of $q_r(e)$ is $q_m(e)$. As λ_c is in general a real number and buffer queue sizes are integers, the number of buffers we can remove from a channel e is only $\min(q_r(e), \lfloor\lambda_c\rfloor)$.

As λ_c is the MCM, the following inequality is satisfied:

$$r(v_j) - r(v_i) \le w(e) + \lambda^* + q_m(e) - \lambda_c, \forall e = (v_i, v_j) \in E. \tag{4.18}$$

Therefore, $\forall e = (v_i, v_j) \in E$, the inequality below must also be satisfied:

$$r(v_j) - r(v_i) \le w(e) + \lambda^* + q_r(e) - \min(q_r(e), \lfloor\lambda_c\rfloor). \tag{4.19}$$

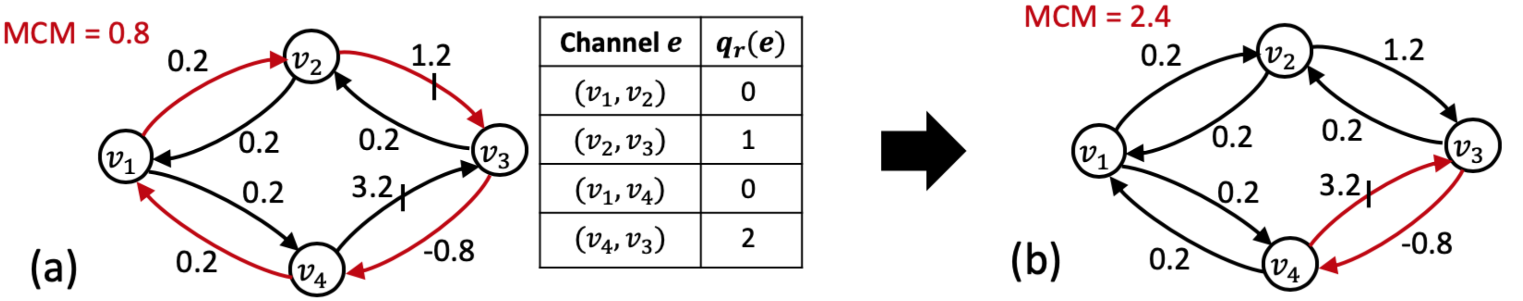
In other words, no negative cycles are introduced in the parametric graph.

Therefore, we can decrement $q_r(e)$ and the edge weight $w_{pe}(e)$ by $\min(q_r(e), \lfloor\lambda_c\rfloor)$. If that results in $q_r(e)$ being zero, we have removed all buffers from the channel. In that case, we un-parameterize the parameterized edge $e = (v_i, v_j)$ in the updated graph.

However, it is possible that $\min(q_r(e), \lfloor\lambda_c\rfloor)$ is zero. That happens when λ_c is less than 1. The implication of this is that we cannot remove buffers from any of the channels in the graph, and therefore cannot update the edge weights of parameterized edges. As there are no changes in the graph in the current iteration, the next iteration will produce the same outcome – an MCM less than 1 and no changes in the graph. This issue is illustrated in Fig. 4.9.

Fig. 4.9(a) is an example of a parametric graph $G_{pe}^{(\lambda^*,q_m)}$ after a few iterations down the run of the algorithm. The numeric weights of all edges are shown and each parameterized edge is shown with a short vertical bar that crosses the edge. We also show in a table the number of buffers left for the channels, denoted by q_r. Note that a numeric value of "0" does not imply the channel has no buffers as each channel always has a minimum buffer size of 1. The minimum-mean cycle is highlighted in red color. As the MCM is 0.8 in the current iteration, we cannot remove any buffers from the channels (and therefore make no changes to the graph).

To resolve this issue, a parameterized edge with the smallest q_r in the minimum-mean cycle is selected and un-parameterized. In this example, the edge (v_2, v_3) is un-parameterized. What this essentially means is that, at this point, we assume that the algorithm has reached the smallest buffer size for that channel, thereby allowing us to un-parameterize the edge. As we have modified the graph to have one fewer parameterized edge, the MCM in the next iteration will be higher, as demonstrated in Fig. 4.9(b). The MCM is now 2.4, which allows us to remove buffers in the remaining parameterized edge (v_4, v_3). The removal of buffers is not shown in this example, but will be demonstrated in Fig. 4.10 in the next subsection.



Channel e	$q_r(e)$
(v_1, v_2)	0
(v_2, v_3)	1
(v_1, v_4)	0
(v_4, v_3)	2

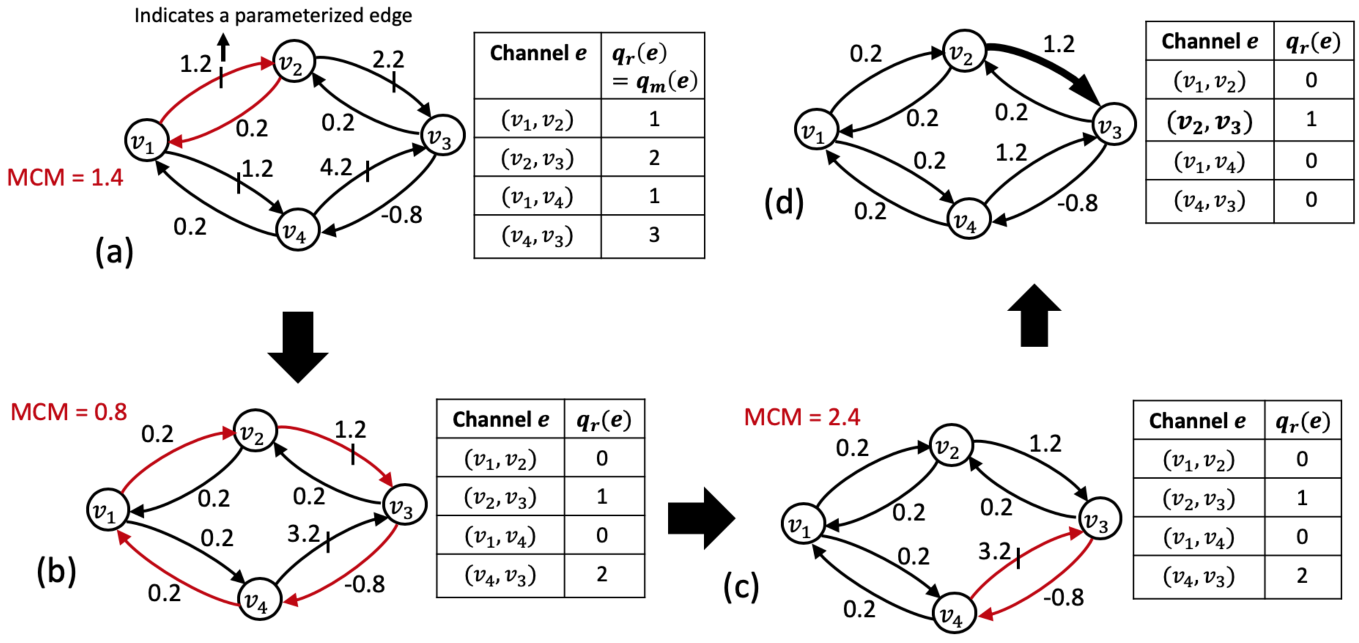
Fig. 4.9.: A parametric graph $G_{pe}^{(\lambda^*,q_m)}$ where a parameterized edge is shown with a short vertical bar crossing the edge. (a) The minimum-mean cycle (edges colored red) is identified. The minimum cycle mean is less than 1. The algorithm therefore does not remove buffers from any of the channels in the graph, and no updates to the weights of parameterized edges are performed. (b) To proceed, a parameterized edge (v_2, v_3) in the minimum-mean cycle corresponding to a channel with minimum number of buffers left is un-parameterized. The MCM (from cycle $v_4 \to v_3 \to v_4$) is now 2.4, which allows us to remove buffers from the remaining parameterized edge (v_4, v_3), which is not shown here, but will be shown in Fig. 4.10.

Of course, having an MCM value 1 or greater does not necessarily mean that buffers can be removed from the channels. Whenever there is no change to the weights of parameterized edges, i.e., no buffers are removed, we would select and un-parameterize a parameterized edge. It is possible that there are multiple parameterized edges in the minimum-mean cycle with the same minimum number of buffers. In that case, one such parameterized edge is selected randomly and un-parameterized.

4.3.4 Illustration of the algorithm through an example

Let us now illustrate the proposed algorithm through the LIS in Fig. 4.1(b). In Fig. 4.10(a), we show the parametric graph $G_{pe}^{(\lambda^*,q_m)}$, where each parameterized edge is shown with a short vertical bar crossing the edge. The numeric weights of the edges in the parametric graph are obtained based on the original edge weights in the lis-graph, the respective $q_m(e)$ values in the table in Fig. 4.10(a), and $\lambda^* = 0.2$. For example, $w(v_1, v_2) = 0$ in the lis-graph as v_1 is a circuit block; therefore, the parameterized edge (v_1, v_2) has a weight of $w(v_1, v_2) + \lambda^* + q_m(v_1, v_2) = 0 + 0.2 + 1 = 1.2$, and the un-parameterized edge (v_2, v_1) has a weight of $-w(v_1, v_2) + \lambda^* = 0 + 0.2 = 0.2$. As v_4 is a relay station, $w(v_4, v_3) = 1$ in the lis-graph; hence, the parameterized edge (v_4, v_3) has a weight of $w(v_4, v_3)+\lambda^*+q_m(v_4, v_3) = 1+0.2+3 = 4.2$, and the un-parameterized edge (v_3, v_4) has a weight of $-w(v_4, v_3) + \lambda^* = -1 + 0.2 = -0.8$.

All channels are initially assigned the maximum allowable buffers (in addition to the minimum buffer queue size of 1), i.e., $q_r(e) = q_m(e), \forall e \in E$. The minimum-mean cycle is identified using red colored edges, and the corresponding MCM is $\lambda_c = 1.4$. As $\min(q_r(e), \lfloor \lambda_c \rfloor) = 1$ for each parameterized edge e, we can remove one buffer from each channel as shown in the table in Fig. 4.10(b). The weight of each parameterized edge is also decremented by 1 accordingly. Moreover, as there are no more buffers left to be removed from channels (v_1, v_2) and (v_1, v_4), they are un-parameterized as shown in the resultant graph in Fig. 4.10(b).



(a)

Channel *e*	$q_r(e)$ $= q_m(e)$
(v_1, v_2)	1
(v_2, v_3)	2
(v_1, v_4)	1
(v_4, v_3)	3

(b)

Channel *e*	$q_r(e)$
(v_1, v_2)	0
(v_2, v_3)	1
(v_1, v_4)	0
(v_4, v_3)	2

(c)

Channel *e*	$q_r(e)$
(v_1, v_2)	0
(v_2, v_3)	1
(v_1, v_4)	0
(v_4, v_3)	2

(d)

Channel *e*	$q_r(e)$
(v_1, v_2)	0
$\boldsymbol{(v_2, v_3)}$	1
(v_1, v_4)	0
(v_4, v_3)	0

Fig. 4.10.: An illustration of the buffer queue sizing algorithm on the LIS in Fig. 4.1(b). (a) A parametric graph $G_{pe}^{(\lambda^*, q_m)}$ where a parameterized edge is shown with a short vertical bar crossing the edge. The table shows the q_m, which are assigned to q_r, and $\lambda^* = 0.2$. The MCM is 1.4, and the minimum-mean cycle is shown in red. (b) A buffer is removed from each of the parameterized edges and the weights of the parameterized edges are updated. Edges (v_1, v_2) and (v_1, v_4) become un-parameterized. This is the example in Fig. 4.9. The MCM value of the current graph is 0.8, and no buffers can be removed from any of the edges. A parameterized edge (v_2, v_3) in the minimum-mean cycle (shown in red) corresponding to a channel with minimum q_r is un-parameterized to obtain a new graph shown in (c). The MCM is 2.4, allowing us to remove two buffers from (v_4, v_3), which is then un-parameterized to give the final graph in (d). The algorithm terminates when the final graph has no more parameterized edges. The table in (d) shows that only the channel (v_2, v_3) uses one additional buffer in the final solution, while each of the remaining channels has minimum buffer queue size of 1.

We again apply the MCM algorithm on this new graph. Since the MCM is now less than 1, we cannot remove buffers from any of the parameterized edges. To continue the algorithm, a parameterized edge in the minimum-mean cycle (red colored edges) that has the lowest q_r is un-parameterized. That would be the edge (v_2, v_3), as shown in the resultant graph in Fig. 4.10(c). This is in fact the example shown in Fig. 4.9 (see the preceding subsection for more details).

Running the MCM algorithm again on the graph in Fig. 4.10(c) yields the MCM of $\lambda_c = 2.4$. As $\min(q_r(v_4, v_3), \lfloor \lambda_c \rfloor) = 2$, we can decrement both $q_r(v_4, v_3)$ and $w_{pe}(v_4, v_3)$ by 2. Since there are no more buffers left in this channel, (v_4, v_3) is un-parameterized. The final graph is shown in Fig. 4.10(d). As there are no more parameterized edges in the graph, the buffer sizing algorithm terminates. The final buffer queue sizing solution inserts only one additional buffer in channel (v_2, v_3) (on top of the minimum buffer queue size of 1). Each of the remaining channels has minimum buffer queue size of 1. This is in fact the optimal cost for this example.

4.3.5 Implementation detail

The pseudocode **Buffer queue sizing algorithm** provides the details of the proposed approach. Given a lis-graph, a target performance λ^*, and upper bounds on buffer queue sizes q_m, the algorithm first constructs the corresponding parameterized graph $G_{pe}^{(\lambda^*, q_m)}(V, E_{pe}, w_{pe}, E_{upe}, w_{upe})$. Each channel e is then initialized with $q_r(e) = q_m(e)$ buffers (in addition to the minimum queue size of 1). While there exists a parameterized edge in $G_{pe}^{(\lambda^*, q_m)}$, the algorithm performs a round of MCM calculation and graph update. An **mcm_algorithm** is invoked to calculate the minimum cycle mean λ_c and also to find the corresponding minimum-mean cycle C. The algorithm then attempts to remove buffers and update the edge weight associated with every parameterized edge e using $\min(q_r(e), \lfloor \lambda_c \rfloor)$. When the residual buffer queue size $q_r(e)$ of a parameterized edge e becomes 0, e is un-parameterized. If the current round of

Buffer queue sizing algorithm

```
buffer_queue_sizing($G_l(V, E, w), \lambda^*, q_m$)
  Construct $G_{pe}^{(\lambda^*, q_m)}(V, E_{pe}, w_{pe}, E_{upe}, w_{upe})$
  $q_r = q_m$
  while $|E_{pe}| > 1$ do
    $[\lambda_c, C]$ = mcm_algorithm($G_{pe}^{(\lambda^*, q_m)}$)
    $buffers\_removed\_flag = 0$
    for every edge $(v_i, v_j)$ in $E_{pe}$ do
      if $\min(q_r(v_i, v_j), \lfloor \lambda_c \rfloor) \neq 0$ do
        $buffers\_removed\_flag = 1$
        /* buffers to be removed */
        $q_r(v_i, v_j) = q_r(v_i, v_j) - \min(q_r(v_i, v_j), \lfloor \lambda_c \rfloor)$
        $w_{pe}(v_i, v_j) = w_{pe}(v_i, v_j) - \min(q_r(v_i, v_j), \lfloor \lambda_c \rfloor)$
        if $q_r(v_i, v_j) == 0$ do
          /* un-parameterize the edge if no buffers left in channel */
          $E_{pe} = E_{pe} - (v_i, v_j)$
          $E_{upe} = E_{upe} + (v_i, v_j)$
        end if
      end if
    end for
    if $buffers\_removed\_flag == 0$ do
      /* buffers not removed */
      /* find a parameterized edge with lowest $q_r$ in $C$ */
      find $(v_i, v_j) = \text{argmin}_{e \in E_{pe} \cap C}(q_r(e))$
      /* un-parameterize the edge with lowest $q_r$ in $C$ */
      $E_{pe} = E_{pe} - (v_i, v_j)$
      $E_{upe} = E_{upe} + (v_i, v_j)$
    end if
  end while
  return $q_r$
end buffer_queue_sizing
```

graph update does not remove any buffers, a parameterized edge in the minimum-mean cycle C with the lowest residual buffer queue size q_r is un-parameterized.

Although any MCM algorithms from [1] [2] [14] [18] can be used to implement the **mcm_algorithm** in the pseudocode, we have chosen to use the Young-Tarjan-Orlin (YTO) algorithm from [14] because of its efficiency, as observed in [7] [37]. Our implementation of the YTO algorithm uses binary heap and has a time complexity of $O(|V||E_{upe} + E_{pe}| \log |V|)$ [7]. As $|E| = |E_{upe}| = |E_{pe}|$ for the initial $G_{pe}^{(\lambda^*, q_m)}$, the time complexity of **mcm_algorithm** is effectively $O(|V||E| \log |V|)$.

The main contributor to the time complexity of the proposed buffer queue sizing algorithm is the iterative invocation of the **mcm_algorithm**. As we un-parameterize at least one parameterized edge in each iteration, the overall time complexity of the proposed algorithm is $O(|V||E|^2| \log |V|)$.

4.4 Proposed throughput optimization algorithm

In the formulation of the new buffer queue sizing problem, we assume an upper bound on the number of buffers for a channel, i.e., $q_m(e)$ for channel e. However, the original formulation of the buffer queue sizing problem in [22] has regional constraints of buffers, i.e., $C(cb_t)$, and not the channel constraint $q_m(e)$. In order to apply the proposed buffer queue sizing algorithm to solve the original problem, we have to convert regional buffer constraints to channel constraints.

We do so by solving the relaxed LP formulation presented in Section 4.2.4. Solving for (4.12), subject to (4.6), (4.8), (4.10), and $q(e) \geq 0\ \forall e \in E$ gives an non-integer buffer solution $q(e)$ for each channel e that satisfies the regional buffer constraints.

Of course, the non-integer buffer solution, denoted as $q_{LP}(e)$, from the LP formulation cannot be used directly. We use the following approximation to obtain an initial guess on $q_m(e)$ for the proposed buffer queue sizing algorithm.

$$q_m(e) = 1 + \lceil q_{LP}(e) \rceil, \forall e \in E. \tag{4.20}$$

Even though these upper bounds on the buffer queue sizes may not meet the regional constraint of buffers, they provide a good starting point for the buffer queue sizing algorithm to find a good residual buffer queue sizing solution $q_r(e)$. Since we start with initial bounds q_m that may be infeasible, we must check for the feasibility of the residual buffer queue sizing solution:

$$\sum_{m(e)=cb_t} q_r(e) \leq C(cb_t), \forall cb_t \in CB. \tag{4.21}$$

If the constraints are not satisfied, we have to lower the target throughput performance in search for a lower-cost buffer solution that meets the regional constraints.

We are now ready to present the proposed throughput optimization algorithm in the following pseudocode, which is still based on the binary search framework presented in Section 4.2.

Proposed throughput optimization algorithm

```
Input  : $G_l(V, E, w)$, $CB$, $C(cb_t)\ \forall cb_t \in CB$, $m(e)\ \forall e \in E$, $PREC$
Output: $\lambda^*$, $q(e)\ \forall\ e \in E$
/* Initialization */
$\lambda^*_{\max} = \lambda^*(G_{e,\min})$, $\lambda^*_{\min} = \lambda^*(G_l)$, $\lambda^*_{tgt} = \lambda^*_{min}$
/* Run binary search for maximum throughput */
while ($\lambda^*_{\max} - \lambda^*_{\min} \geq PREC$)
  /* Call the LP solver */
  check feasibility of $\lambda^*_{tgt}$ and get $q_{LP}(e)$ by solving for (4.12) subject to (4.6), (4.8), and (4.10) with $q(e) \geq 0\ \forall\ e \in E$
  if($\lambda^*_{tgt}$ is feasible)
    /* Run proposed buffer queue sizing algorithm */
    for every channel edge $e$ in $E$ do
      $q_m(e) = 1 + \lceil q_{LP}(e) \rceil$
    end for
    $q_r$ = buffer_queue_sizing($G_l, \lambda^*_{tgt}, q_m$)
    if ($\sum_{m(e)=cb_t} q_r(e) \leq C(cb_t), \forall cb_t \in CB$) /* regional buffer constraint (4.10) */
      $\lambda^*_{\max} = \lambda^*_{tgt}$
    else
      $\lambda^*_{\min} = \lambda^*_{tgt}$
    end if
  else
    $\lambda^*_{\min} = \lambda^*_{tgt}$
  end if
  $\lambda^*_{tgt} = (\lambda^*_{\max} + \lambda^*_{\min})/2$
end while
$\lambda^* = \lambda^*_{\max}$
```

We perform a binary search for the best system throughput attainable as in approach from [22]. A preliminary feasibility check of a particular λ^*_{tgt} is performed by solving a relaxed LP formulation, which also yields q_{LP} if λ^*_{tgt} is feasible for the relaxed problem. If λ^*_{tgt} is feasible, the **buffer_queue_sizing** algorithm is called, using q_{LP} as a good guess of q_m as in (4.20). If the residual buffer queue sizing solution q_r satisfies the regional buffer constraints, we have found a better throughput and therefore, $\lambda^*_{\max}$ is updated. If the relaxed LP formulation is infeasible or the residual buffer queue sizing solution does not satisfy the regional buffer constraints, we have to update $\lambda^*_{\min}$ in order to look for a lower throughput solution. The binary search process terminates when the difference between $\lambda^*_{\max}$ and $\lambda^*_{\min}$ is less than a user-specified precision level. Again, even though the pseudocode does not show it, a buffer queue sizing solution that corresponds to the best throughput attainable is a by-product of the optimization process.

Further optimization (not shown in the pseudocode) is possible to remove more buffers from the channels. One can do so by removing a buffer from a channel (and also making updates to its parameterized edge weight) and checking for negative cycles using Bellman-Ford algorithm [36] or the **mcm_algorithm** used in this work. If there are no negative cycles, the buffer can be removed safely. This can be repeated for every parameterized edge e that has non-zero $q_r(e)$. As the final buffer queue sizing solution usually has a low buffer count, the runtime for this further optimization is insignificant.

4.5 Experimental results

We have implemented the MILP-based optimization approach from [22] (Section 4.2.3), our minor improvement to it (Section 4.2.4), and the proposed throughput optimization algorithm (Section 4.4) in C++. In these implementations, we have used lp_solve 5.5.2.5 [48] as the MILP solver and the LP solver. These implementations are evaluated on Intel(R) Xeon(R) CPU E5-2660 2.60 GHz Linux machine with 66GB of RAM.

We consider large size examples that represent the future system complexities [47] for the evaluation of the proposed throughput optimization algorithm against the original MILP-based approach and improved version. We vary the number of nodes (i.e. the number of blocks) in the latency insensitive system from 400 to 9100. For each combination of the number of blocks, the number of communication channels, and the number of relay stations as shown in Table 4.2, we create a lis-graph randomly. For each lis-graph, we randomly generate two sets of regional buffer constraints. One set of regional buffer constraints is quite relaxed in that the attainable highest system performance is closer to that of a system with unlimited buffer queue size. The other set of regional buffer constraints is more stringent in that the attainable highest system performance would degrade quite significantly. We essentially use a larger value of $C(cb_t)$ in (4.10) for the relaxed constraints compared to the tight constraints.

Table 4.2.: Details of lis-graphs used for the evaluation of various approaches.

	Number of blocks	Number of channels	Number of relay stations
LIS1	400	1600	100
LIS2	700	800	100
LIS3	1200	4800	300
LIS4	2000	8000	500
LIS5	2100	2400	300
LIS6	2800	11200	700
LIS7	3500	4000	500
LIS8	3600	14400	900
LIS9	4400	17600	1100
LIS10	4900	5600	700
LIS11	5200	20800	1300
LIS12	6000	24000	1500
LIS13	6300	7200	900
LIS14	7700	8800	1100
LIS15	9100	10400	1300

We first compare the proposed approach against the two MILP-based approaches using lis-graphs with relaxed regional buffer constraints. The results in Table 4.3 are obtained using $PREC = 0.0001$ as the level of precision to decide when to terminate the binary search in all three approaches. The column labeled "T_{MQ}" corresponds to the throughput when the latency insensitive systems have minimum buffer queue size of 1. The column labeled "T_{UQ}" corresponds to the throughput when the latency insensitive systems have unlimited buffer queue size. The column labeled "$N_{\min}$" shows the minimum number of buffers a system must have. It is the same as the number of channels in Table 4.2.

For each approach used to solve the throughput optimization problem, we show the highest throughput performances attainable in the column labeled "T_{CQ}". The numbers of additional buffers (on top of the minimum buffer queue of size 1) required to attain the reported throughput performances are shown in the column labeled "N_{buf}". For the two MILP-based approaches, $N_{buf} = \sum_{e \in E} q(e)$, whereas for the

Table 4.3.: Comparing the proposed approach with MILP-based approaches for relaxed regional buffer constraints.

	T_{MQ}	T_{UQ}	N_{min}	MILP-based				The proposed approach		
				T_{CQ}	N_{buf}	$T_{exec}(sec.)$ [22]	$T_{exec}(sec.) = T_{lp} + T_{milp}$ (Our implementation of [22])	T_{CQ}	N_{buf}	$T_{exec}(sec.) = T_{lp} + T_{graph}$
LIS1	0.2500	0.3077	1600	0.3077	11	6.9	1.4 + 3.1	0.3077	12	1.4 + 4.9
LIS2	0.4694	0.5200	800	0.5200	15	2.1	1.1 + 0.5	0.5200	15	1.1 + 4.3
LIS3	0.2368	0.2414	4800	0.2414	3	11.5	7.0 + 3.9	0.2414	3	7.3 + 6.2
LIS4	0.2250	0.2286	8000	0.2285	1	25.8	19.6 + 5.5	0.2285	1	20.0 + 4.6
LIS5	0.2727	0.3519	2400	0.3518	5	7.1	4.8 + 1.9	0.3518	5	5.0 + 4.6
LIS6	0.1667	0.1852	11200	0.1852	1	41.6	30.2 + 10.4	0.1852	1	30.9 + 7.6
LIS7	0.3529	0.4400	4000	0.4118	9	29.8	13.6 + 10.8	0.4118	9	13.9 + 6.6
LIS8	0.1964	0.2222	14400	0.2222	22	39238.5	86.7 + 19552.8	0.2222	22	87.3 + 7.8
LIS9	0.1667	0.1765	17600	0.1765	1	75.7	56.5 + 18.0	0.1765	1	57.8 + 6.4
LIS10	0.4054	0.4444	5600	0.4444	16	303.4	29.0 + 143.3	0.4444	16	29.3 + 6.9
LIS11	0.1667	0.2222	20800	0.2222	11	9957.7	193.3 + 4909.7	0.2222	11	194.3 + 8.4
LIS12	0.1905	0.2239	24000	0.2239	19	18560.7	518.0 + 9120.4	0.2239	19	519.6 + 11.1
LIS13	0.4235	0.4524	7200	0.4523	17	4088.1	45.3 + 2026.9	0.4523	17	45.7 + 10.0
LIS14	0.3441	0.4321	8800	−−	−−	−−	−−	0.4238	33	105.2 + 8.3
LIS15	0.2917	0.4104	10400	−−	−−	−−	−−	0.3896	41	83.5 + 14.7

proposed approach, $N_{buf} = \sum_{e\in E} q_r(e)$, based on the notation used in the respective sections describing the approaches. The actual total number of buffers for each system is $N_{\min}+N_{buf} = \sum_{e\in E} Q(e)$. As $N_{\min}$ is same for all the three approaches, highlighting N_{buf} can more clearly demonstrate the quality of the solutions produced by the three approaches. The two MILP-based approaches are equivalent; therefore, they share the same "T_{CQ}" and "N_{buf}".

The table also lists the runtimes taken to compute the solutions in the columns labeled "T_{exec}" for the three approaches. For the improved MILP-based approach, we show the time taken by the LP solver, i.e., "T_{lp}", and the time taken by the MILP solvers, "T_{milp}". For the proposed approach, we show the time taken by the LP solver, i.e., "T_{lp}", and the time taken by the graph-based buffer queue sizing algorithm, i.e., "T_{graph}". The higher the "T_{CQ}". the better is the throughput; the lower the "N_{buf}", the lower is the cost; the lower the "T_{exec}", the more efficient is the approach.

The two MILP-based approaches fail to produce solutions for two test cases LIS14 and LIS15, as indicted by "$--$", even after running the programs for a few days. For the rest of test cases (LIS1–13), the solutions from the two MILP-based approaches and the proposed approach have the same throughput performances. The buffer queue sizing solutions have identical costs except for LIS1, where the proposed approach uses one more buffer than the MILP-based approaches. As we may consider the solutions that are available from the MILP-based approaches to be optimal, we observe that the proposed approach also produces optimal throughput performances for all test cases LIS1–13. As the regional buffer constraints are relatively relaxed, the throughput performances for most test cases are exactly or close to T_{UQ} (system performance when queues are unlimited). The buffer queue sizing solutions from the proposed approach are also close to being optimal.

The runtimes of the proposed approach remain scalable even when the size of the lis-graph increases. On the other hand, the two MILP-based approaches see a sharp increase in runtimes when the size of the lis-graph increases. As expected, the improved MILP-based approach scales better than the original MILP-based approach from [22]. Based on Table 4.3, we may conclude that the proposed throughput optimization algorithm is a scalable approach in producing solutions that are optimal or close to optimal.

In order to obtain some results for LIS14 and LIS15, we also apply the improved MILP-based approach on these two test cases with lower precision of $PREC \in \{0.001, 0.01, 0.1\}$. (We use the improved MILP-based approach because it is more scalable than the original MILP-based approach.) Even with the lower precision, we manage to obtain solution only when $PREC = 0.1$, as shown in Table 4.4. The throughput performances, T_{CQ}, obtained by the improved MILP-based approach at $PREC = 0.1$ are not as good as the throughput performances obtained by the proposed approach at a higher precision of $PREC = 0.0001$. As the throughput performances are lower, the corresponding buffer queue sizing solutions have fewer buffers compared to those obtained by the proposed approach at a higher precision.

We also apply the proposed approach to LIS14 and LIS15 at lower precision of $PREC \in \{0.001, 0.01, 0.1\}$. As expected, the throughput degrades at lower precision. (Note that we report T_{CQ} with six decimal places in Table 4.4 to show more clearly the degradation in throughput.) We also observe in Table 4.4 that at $PREC = 0.1$, the solutions from the proposed approach have the same throughput and buffering cost as those of the MILP-based approach. Again, the proposed approach is significantly more efficient than the improved MILP-based approach. The results for LIS14 are also a clear indicator that the proposed buffer queue sizing algorithm is a heuristic, as the number of buffers increases from 33 to 34 even when the throughput drops from 0.423836 to 0.423828.

We now present results for the test cases with tight regional buffer constraints. As in Table 4.3, Table 4.5 reports the details of the solutions produced by the three approaches for LIS1–15 using $PREC = 0.0001$. All three approaches produce solutions that are of the same quality, in terms of throughput T_{CQ} and buffer counts N_{buf}. As we may assume that the MILP-based approaches are optimal, the proposed algorithm manages to also obtain optimal solutions for all test cases. We also observe in

Table 4.4.: Comparing the proposed approach with our implementation of [22] for relaxed regional buffer constraints with different $PREC \in \{0.0001, 0.001, 0.01, 0.1\}$.

	$PREC$	Our implementation of [22]			The proposed approach		
		T_{CQ}	N_{buf}	$T_{exec}(sec.) = T_{lp} + T_{milp}$	T_{CQ}	N_{buf}	$T_{exec}(sec.) = T_{lp} + T_{graph}$
LIS14	0.0001	−−	−−	−−	0.423836	33	105.2 + 8.3
	0.001	−−	−−	−−	0.423836	33	105.1 + 7.8
	0.01	−−	−−	−−	0.423828	34	52.7 + 6.7
	0.1	0.406250	14	25.7 + 37.5	0.406250	14	25.9 + 3.6
LIS15	0.0001	−−	−−	−−	0.389587	41	83.5 + 14.7
	0.001	−−	−−	−−	0.389160	39	57.6 + 9.6
	0.01	−−	−−	−−	0.382812	26	30.5 + 4.4
	0.1	0.382812	26	30.2 + 19339.0	0.382812	26	30.5 + 4.3

Table 4.5.: Comparing the proposed approach with MILP-based approaches for tight regional buffer constraints.

				MILP-based				The proposed approach		
	T_{MQ}	T_{UQ}	N_{min}	T_{CQ}	N_{buf}	$T_{exec}(sec.)$ [22]	$T_{exec}(sec.) = T_{lp} + T_{milp}$ (Our implementation of [22])	T_{CQ}	N_{buf}	$T_{exec}(sec.) = T_{lp} + T_{graph}$
LIS1	0.2500	0.3077	1600	0.2857	4	1.6	1.1 + 0.4	0.2857	4	1.2 + 3.4
LIS2	0.4694	0.5200	800	0.4912	3	1.3	0.7 + 0.4	0.4912	3	0.9 + 6.8
LIS3	0.2368	0.2414	4800	0.2414	3	11.6	7.1 + 3.8	0.2414	3	7.4 + 6.6
LIS4	0.2250	0.2286	8000	0.2285	1	26.6	20.6 + 5.5	0.2285	1	21.1 + 4.5
LIS5	0.2727	0.3519	2400	0.2941	1	4.2	3.0 + 1.0	0.2941	1	3.2 + 5.1
LIS6	0.1667	0.1852	11200	0.1852	1	41.0	29.7 + 10.5	0.1852	1	30.4 + 7.4
LIS7	0.3529	0.4400	4000	0.3864	3	16.8	12.4 + 4.0	0.3864	3	12.7 + 6.3
LIS8	0.1964	0.2222	14400	0.2000	1	69.9	52.4 + 16.7	0.2000	1	52.9 + 6.5
LIS9	0.1667	0.1765	17600	0.1667	0	74.9	59.0 + 14.8	0.1667	0	59.8 + 6.1
LIS10	0.4054	0.4444	5600	0.4230	3	31.7	22.3 + 8.7	0.4230	3	22.5 + 6.2
LIS11	0.1667	0.2222	20800	0.2105	6	326.3	132.6 + 125.6	0.2105	6	133.1 + 8.7
LIS12	0.1905	0.2239	24000	0.2206	15	1019.7	445.7 + 371.4	0.2206	15	445.4 + 12.0
LIS13	0.4235	0.4524	7200	0.4354	5	68.7	34.0 + 28.3	0.4354	5	34.4 + 10.2
LIS14	0.3441	0.4321	8800	0.4058	23	1039.6	70.4 + 504.9	0.4058	23	71.1 + 11.4
LIS15	0.2917	0.4104	10400	0.3333	3	84.4	58.4 + 24.8	0.3333	3	59.2 + 9.3

Table 4.5 that for very tight regional buffer constraints, it may not be always possible to reach close to the maximum throughput performance. This can be attributed to the fact that satisfying the regional buffer constraints would require one to sacrifice the throughput performance as there is a direct trade-off involved.

When it comes to runtime performance, the proposed approach certainly still scales better than the MILP-based approaches, although the advantage of the proposed approach is no longer that pronounced as compared to the results in Table 4.3. It is also interesting that the runtime performance of the MILP-based approaches for the tight constraints (Table 4.5) is considerably better than for the relaxed constraints (Table 4.3).

Summarizing the results from Table 4.3, Table 4.4, and Table 4.5, the proposed approach, which essentially combines an LP solver and a graph optimization algorithm, always runs in reasonable time. Significant portion of the runtime in the proposed approach goes to solving the LP formulation. One reason for the longer runtime is that the LP solver is called for every target λ^*_{tgt}, whereas the proposed graph algorithm runs only when a particular λ^*_{tgt} is found to be feasible. In fact, the runtimes for the LP solver in the proposed approach and in the improved MILP-based approach are similar. In other words, the scalability of the proposed approach stems from the advantage of the graph-based optimization over an MILP solver.

One may argue that the scalability is not a direct result of switching from an MILP formulation to a graph formulation; rather, it is the new form of buffer queue sizing problem in Section 4.3.1 that contributes to the efficiency of the proposed approach. Recall that we convert the regional buffer constraints proposed in the formulation in [22] into the channel buffer constraints so that we can formulate the buffer queue sizing problem as a parametric graph. This conversion allows us to use an integer-variant of the minimum balance algorithm to perform buffer queue sizing. As the MILP is a general solver that is capable of handing the new formulation in Section 4.3.1, we present a fourth approach of throughput optimization: Instead of calling **buffer_queue_sizing** in line 12 of the pseudocode **Proposed throughput**

Table 4.6.: Using an MILP solver to solve the new buffer queue sizing problem for relaxed regional buffer constraints.

	T_{CQ}	N_{buf}	$T_{exec}(sec.) =$ $T_{lp} + T_{milp}$
LIS1	0.3077	11	1.2 + 2.8
LIS2	0.5200	15	1.0 + 0.5
LIS3	0.2414	3	7.1 + 3.6
LIS4	0.2285	1	19.6 + 5.1
LIS5	0.3518	5	4.9 + 1.9
LIS6	0.1852	1	30.4 + 9.8
LIS7	0.4118	9	13.8 + 9.8
LIS8	0.2222	22	86.9 + 24722.9
LIS9	0.1765	1	56.7 + 16.7
LIS10	0.4444	16	29.1 + 104.0
LIS11	0.2222	11	193.5 + 4544.2
LIS12	0.2239	19	520.2 + 26816.4
LIS13	0.4523	17	45.6 + 2555.3
LIS14	$--$	$--$	$--$
LIS15	$--$	$--$	$--$

optimization algorithm, we use an MILP solver to solve the new form of buffer queue sizing problem of maximizing $\sum_{e \in E} \lambda_p(e)$, subject to the constraints that $0 \leq \lambda_p(e) \leq q_m(e)$, $\lambda_p(e) \in Z^{\geq}$, and the shortest path constraints (4.15) and (4.17).

We present the results of this fourth approach in Table 4.6 for the relaxed regional buffer constraints and Table 4.7 for the tight regional buffer constraints. For the relaxed constraints, the fourth approach outputs the same T_{CQ} and N_{buf} as in Table 4.3. However, the runtime in general is inferior to our implementation of [22]. For the tight constraints, the algorithm does not perform well as the throughput attainable falls below the throughput reported in Table 4.5 for the other algorithms. Therefore, it is justifiable to conclude that the parametric graph formulation of the new buffer queue sizing problem is the main reason for the scalability of the proposed throughput optimization algorithm.

Table 4.7.: Using an MILP solver to solve the new buffer queue sizing problem for tight regional buffer constraints.

	T_{CQ}	N_{buf}	$T_{exec}(sec.) =$ $T_{lp} + T_{milp}$
LIS1	0.2800	2	1.0 + 0.7
LIS2	0.4807	2	0.7 + 0.7
LIS3	0.2414	3	7.1 + 3.6
LIS4	0.2285	1	20.6 + 5.1
LIS5	0.2727	0	2.9 + 1.6
LIS6	0.1852	1	29.8 + 9.8
LIS7	0.3864	3	12.5 + 3.8
LIS8	0.2000	1	52.6 + 15.5
LIS9	0.1667	0	59.1 + 13.6
LIS10	0.4230	3	22.3 + 7.9
LIS11	0.2083	2	128.9 + 84.6
LIS12	0.2017	3	222.6 + 389.9
LIS13	0.4269	1	35.3 + 27.7
LIS14	0.3441	0	34.9 + 26.0
LIS15	0.3124	2	34.6 + 24.9

4.6 Conclusions

In latency insensitive system design, proper sizing of buffer queues can reduce the cost of implementation while achieving the same system performance. We have presented a parametric graph formulation of a new form of buffer queue sizing problem and proposed an optimization approach based on an integer-variant of the minimum balance algorithm. Experimental results show that the proposed approach scales well while quality of the solutions obtained, in terms of system throughput attainable and buffer queue sizes, is as good as that of the MILP-based approach.

5. SUMMARY AND FUTURE RESEARCH

This dissertation has explored efficient minimum cycle mean algorithms and their applications. Particularly, in chapter 2, we have presented several techniques to improve the early termination check in the Hartmann-Orlin's (HO) algorithm. These improvements allow the efficient early termination check to be performed in $O(|V|)$ time on the average empirically. Such efficiency also allows to reduce the cost of vertical relaxation. Consequently, the proposed algorithm has better runtime performance than HO and produces comparable results to YTO for circuit graphs and dense random graphs. However, when it comes memory usage, the proposed algorithm is significantly better than YTO.

Minimum balancing is an application of the minimum cycle mean algorithm. In Chapter 3, we have proposed an improvement to the conventional minimum balance algorithms by performing fewer re-weighting of edge weights in each iteration, resulting in better efficiency. We see 42.56% and 34.26% average runtime improvements over state-of-the-art minimum balance algorithms for graphs derived from IWLS 2005 benchmark circuits and randomly generated graphs, respectively.

We have also applied the minimum cycle mean algorithm in latency insensitive system design. In latency insensitive system design, proper sizing of buffer queues can reduce the cost of implementation while achieving the same throughput performance. In Chapter 4, we have presented a parametric graph formulation of a new form of buffer queue sizing problem and proposed an optimization approach based on an integer-variant of the minimum balance algorithm. The minimum balance algorithm uses the minimum cycle mean algorithm as a subroutine. Experimental results show that the proposed approach scales well while quality of the solutions in terms of the system throughput and buffer queue sizes is as good as that of the MILP-based approach.

We now present possible future research opportunities based on these works.

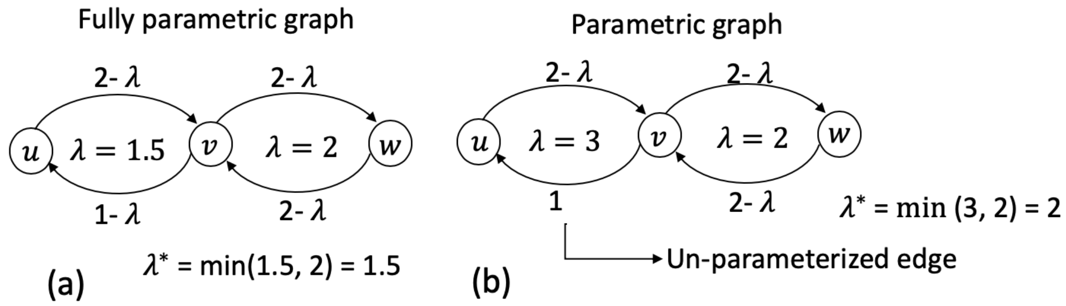


Fig. 5.1.: Showing an example of (a) fully parametric graph where all edges are parameterized, that is a parameter λ is associated with every edge, (b) parametric graph where not all edges have a parameter λ.

5.1 Hartmann-Orlin's algorithm for parametric graph

In the problem formulation for minimum cycle mean and minimum balancing in Chapters 2 and 3 we have assumed that all edges in the graph are parameterized (fully parametric graph example, see Figure 5.1) that is, a parameter $\lambda(e)$ is associated with every edge $e \in E$ in $G(V, E, w)$. But there can also be situation where do not want to maximize slack or tolerance for all the constraints. If such is the requirement, some edges in the timing constraint graph would have just a constant value of weight and no parameter associated with it. This generic formulation is closely related to the parametric shortest path problem which can be solved by YTO algorithm as efficiently as for the fully parametric case. In Chapter 4, for solving the buffer queue sizing problem, we have in fact used the version of the YTO algorithm that can handle parametric graphs. Remember in the extended lis-graph only the mirror edges are parametrized. But to solve such problem with the proposed MCM algorithm, there is a need of extra round of relaxation (which we call horizontal relaxation) to handle un-parameterized edges. Hartmann and Orlin [18] suggest using Dijkstra's shortest path algorithm for horizontal relaxation, but running shortest path algorithm as many times as vertical relaxation in a straightforward manner can be time consuming if size of the un-parameterized graph is large (i.e. there are many edges without the parameter λ). A speed-up of the horizontal relaxation is therefore necessary. To the best of our knowledge there is no existing experimental study of the HO algorithm on such a general problem.

5.2 The proposed MCM algorithm for solving the minimum balance problem

Although any MCM algorithm can be used as the subroutine for MCM calculation in the minimum balance algorithm, historically only two MCM algorithms have been used for this purpose. While Schneider and Schneider [3] uses a modified version of the Karp's algorithm [1] that repeatedly calculates maximum cycle mean from scratch on the modified graph for solving the maximum balance problem (closely related to the minimum balancing), Young, Tarjan and Orlin [14] shows a way to continue using the shortest path tree calculated by the YTO algorithm in the previous iteration for the next iteration to solve the minimum balance problem. Since the algorithm does not rebuild the shortest path tree from scratch, there is runtime benefit.

In our implementation of the minimum balance algorithm including that of the Schneider and Schneider's, we have used the YTO algorithm as the MCM subroutine. Although it is noted that any maximum (minimum) cycle mean algorithm can be used for solving the maximum (minimum) balance problem, Schneider and Schneider's study is mostly focused on introducing the problem itself and also on the other subroutine that collapses a maximum/minimum mean cycle. Given the competitive performance in runtime on circuit graphs as well as less overhead in memory usage of the proposed MCM algorithm presented in this dissertation, we think it might be worth exploring ways to extend the algorithm much the same way it is done with YTO. Note that a small improvement in one MCM calculation can result in significant overall improvement when many MCM calculations are involved. Minimum balancing does require solving MCM problem many times.

However for the buffer queue sizing problem, the proposed approach has to run the MCM algorithm from scratch in every iteration, therefore one of the benefit of using the YTO algorithm, i.e. making use of the previous shortest path tree information for building the next tree is no more applicable. We must also take into account the additional memory overhead of the YTO algorithm. It might be worth exploring parameterized version of the proposed MCM algorithm to implement the proposed buffer queue sizing approach.

5.3 Solving other integer linear programming problems

The problem of optimal slack distribution in clock network can be formulated also as a linear programming problem [56]. Since combinatorial algorithms have been observed to run faster for difference constraint problems [12], minimum balancing approach for solving the slack distribution problem has been received well by the research community. However both approaches are capable of solving the problem optimally in polynomial time.

On the contrary, the mixed integer linear programming (MILP) formulation of the buffer sizing problem cannot be solved optimally in reasonable time since MILP is known to have no polynomial time algorithm. The proposed approach to buffer sizing is scalable but it is still a heuristic technique, no optimal solution is therefore guaranteed. The reason is that often the algorithm has to make a random choice to prevent getting stuck. We have discussed the issue before. If the MCM value in any iteration is below one, no changes occur in the graph and hence the algorithm makes no progress. We resolve this by making a greedy choice since we always pick the parameterized edge with smallest number of buffers left and un-parameterize it. There could be multiple such edges. Notwithstanding, the quality of the solution of the proposed approach is as good as that of the MILP based approach over a wide range in size of latency insensitive systems as we have shown in the experimental results.

We believe the unique approach that we have developed to solve the buffer sizing problem is open for adoption in other integer linear programming problems. Particularly, in applications where there are difference constraints with some parameters to be optimized are integers, it could be worth exploring ways to apply the proposed method. One thing to note is that the buffer sizing problem that we have solved in this dissertation is of mixed integer linear programming type because of the presence of both integers and real variables. The longest/shortest path distance r captures the non-integer part in the problem since λ^* is a real variable, while buffer number captures the integer part. However, the approach can be extended to pure integer linear programming problems as well.

VITA

VITA

Supriyo Maji obtained a Bachelor's degree in Electronics and Telecommunication Engineering from IIEST, Shibpur, India in 2007 and a Master's degree in Electronics and Electrical Communication Engineering from IIT, Kharagpur, India in 2011. He worked as R&D engineer for Synopsys (via Magma) in Bangalore. He also has experience working for Qualcomm in San Diego, HAL (SLRDC) in Hyderabad and Mindtree Limited in Bangalore. His current research interest is in design automation of electronic circuits.

REFERENCES

REFERENCES

[1] R. M. Karp, “A Characterization Of The Minimum Cycle Mean In A Diagraph,” *Discrete Mathematics 23*, pp. 309–311, 1978.

[2] R. M. Karp and J. B. Orlin, “Parametric shortest path algorithms with an application to cyclic staffing,” *Discrete Applied Mathematics*, vol. 3, pp. 37–45, 1981.

[3] H. Schneider and M. Schneider, “Max-Balancing Weighted Directed Graphs And Matrix Scaling,” *Mathematics of Operation Research*, vol. 16, no. 1, pp. 208–222, 1991.

[4] J. Orlin and A. Sedeno-Noda, “An O(Nm) Time Algorithm for Finding the Min Length Directed Cycle in a Graph,” in *ACM-SIAM Symposium on Discrete Algorithms*, 2017.

[5] C. Albretch, B. Korte, J. Schietke, and J. Vygen, “Maximum mean weight cycle in a digraph and minimizing cycle time of a logic chip,” *Discrete Applied Mathematics*, pp. 103–127, 2002.

[6] A. Dasdan and R. K. Gupta, “Faster Maximum and Minimum Mean Cycle Algorithms for System-Performance Analysis,” *IEEE Transaction On Computer-Aided Design of Integrated Circuits And Systems*, vol. 17, no. 10, 1998.

[7] A. Dasdan, “Experimental Analysis of the Fastest Optimum Cycle Ratio and Mean Algorithms,” *ACM Transactions on Design Automation of Electronics System*, vol. 9, no. 4, pp. 385–418, 2004.

[8] G. Wu and C. Chu, “A Fast Incremental Cycle Ratio Algorithm,” in *International Symposium on Physical Design*, 2017, pp. 75–82.

[9] R. Lu and C.-K. Koh, “Performance analysis of latency-insensitive systems,” *IEEE Transaction On Computer-Aided Design of Integrated Circuits And Systems*, vol. 25, no. 3, pp. 469–483, 2006.

[10] ——, “Performance Optimization of Latency Insensitive Systems Through Buffer Queue Sizing of Communication Channels,” in *International Conference on Computer-Aided Design*, 2003, pp. 227–231.

[11] S. Held, “Timing closure in chip design,” Ph.D. dissertation, University of Bonn, 2008.

[12] C. Albretch, B. Korte, J. Schietke, and J. Vygen, “Cycle time and slack optimization for VLSI-chips,” in *International Conference on Computer-Aided Design*, 1999.

[13] S. Held, B. Korte, J. Massberg, M. Ringe, and J. Vygen, "Clock Scheduling And Clocktree Construction For High Performance ASICS," in *International Conference on Computer-Aided Design*, 2003.

[14] N. E. Young, R. E. Tarjan, and J. B. Orlin, "Faster Parametric Shortest Path and Minimum-Balance Algorithms," *Networks*, vol. 21, no. 2, 1991.

[15] A. B. Kahng, S. Kang, J. Li, and J. P. D. Gyvez, "An Improved Methodology for Resilient Design Implementation," *ACM Transactions on Design Automation of Electronics System*, vol. 20, no. 4, pp. 1–26, 2015.

[16] J.-L. Tsai, D. H. Baik, C.-P. Chen, and K. K. Saluja, "A yield improvement methodology using pre- and post-silicon statistical clock scheduling," in *International Conference on Computer-Aided Design*, 2004.

[17] J. M. Rabaey, A. P. Chandrakasan, and B. Nikoli, *Digital integrated circuits: a design perspective.* Pearson Education, 2003.

[18] M. Hartmann and J. B. Orlin, "Finding Minimum Cost to Time Ratio Cycles with Small Integral Transit Times," *Networks*, vol. 23, pp. 567–574, 1993.

[19] A. Dasdan, "An Experimental Study of Minimum Mean Cycle Algorithms," *UCI-ICS Technical Report*, 1998.

[20] C. Albrecht, "IWLS 2005 Benchmarks," in *International Workshop for Logic Synthesis*, 2005.

[21] L. P. Carloni, "From Latency-Insensitive Design to Communication-Based System-Level Design," *Proceedings of the IEEE*, vol. 103, no. 11, pp. 2133–2151, 2015.

[22] R. Lu and C.-K. Koh, "Performance Optimization Of Latency Insensitive Systems Through Buffer Queue Sizing Of Communication Channels," in *International Conference on Computer-Aided Design*, 2003, pp. 227–231.

[23] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli, "Theory of latency-insensitive design," *IEEE Transaction On Computer-Aided Design of Integrated Circuits And Systems*, vol. 20, no. 9, pp. 1059–1076, 2001.

[24] R. M. Karp, "Reducibility Among Combinatorial Problems," in *Complexity of Computer Computations, Plenum Press, New York*, 1972, pp. 85–103.

[25] E. Klotz and A. Newman, "Practical guidelines for solving difficult mixed integer linear programs," *Surveys in Operations Research and Management Science*, vol. 18, pp. 18–32, 2013.

[26] R. L. Collins and L. P. Carloni, "Topology-Based Performance Analysis and Optimization of Latency-Insensitive Systems," *IEEE Transaction On Computer-Aided Design of Integrated Circuits And Systems*, vol. 27, no. 12, pp. 2277–2290, 2008.

[27] C.-H. Li and L. P. Carloni, "Leveraging Local Intra-core Information to Increase Global Performance in Block-Based Design of Systems-on-Chip," *IEEE Transaction On Computer-Aided Design of Integrated Circuits And Systems*, vol. 28, no. 2, pp. 165–178, 2009.

[28] R. L. Collins and L. P. Carloni, “Topology-Based Optimization of Maximal Sustainable Throughput in a Latency-Insensitive System,” in *Design Automation Conference*, 2007, pp. 410–415.

[29] J.-D. Huang, Y.-H. Chen, and Y.-C. Ho, “Throughput Optimization for Latency-Insensitive System with Minimal Queue Insertion,” in *Asia and South Pacific Design Automation Conference*, 2011, pp. 585–590.

[30] D. Bufistov, J. Julvez, and J. Cortadella, “Performance optimization of elastic systems using buffer resizing and buffer insertion,” in *International Conference on Computer-Aided Design*, 2008, pp. 442–448.

[31] B. Xue and S. K. Shukla, “Optimization of Back Pressure and Throughput for Latency Insensitive Systems,” in *International Conference on Computer Design*, 2010, pp. 45–51.

[32] B. Xue, S. K. Shukla, and S. S. Ravi, “Optimization of Latency Insensitive Systems Through Back Pressure Minimization,” *IEEE Transaction of Computers*, vol. 64, no. 2, pp. 464–476, 2015.

[33] M. R. Casu and L. Macchiarulo, “A new approach to latency insensitive design,” in *Design Automation Conference*, 2004, pp. 576–581.

[34] ——, “Throughput-driven foorplanning with wire pipelining,” *IEEE Transaction On Computer-Aided Design of Integrated Circuits And Systems*, vol. 24, no. 5, pp. 663–675, 2005.

[35] 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 *Asia and South Pacific Design Automation Conference*, 2006.

[36] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction To Algorithms*. MIT Press, 2009.

[37] L. Georgiadis, A. V. Goldberg, R. E. Tarjan, and R. F. Wemeck, “An Experimental Study of Minimum Mean Cycle Algorithms,” in *Proceedings of the Eleventh Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2009, pp. 1–13.

[38] A. Dasdan, S. S. Irani, and R. K. Gupta, “Efficient Algorithms for Optimum Cycle Mean and Optimum Cost to Time Ratio Problems,” in *Design Automation Conference*, 1999.

[39] *https://github.com/alidasdan*.

[40] H. Schneider and M. Schneider, “Towers and cycle covers for max-balanced graph,” *Mathematics of Operation Research*, vol. 73, pp. 159–170, 1990.

[41] E. Lawler, *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart and Winston, New York, 1976.

[42] B. V. Cherkassky, A. V. Goldberg, and T. Radzik, “Shortest Paths Algorithms: Theory and Experimental Evaluation,” *Mathematical Programming*, vol. 73, pp. 129–174, 1996.

[43] R. Lu and C.-K. Koh, "Performance Analysis and Efficient Implementation of Latency Insensitive Systems," *Technical Report, TR-ECE03-06, School of Electrical and Computer Engineering, Purdue University*, 2003.

[44] ——, "Performance Analysis of Latency-Insensitive Systems," *IEEE Transaction On Computer-Aided Design of Integrated Circuits And Systems*, vol. 25, no. 3, pp. 469–483, 2006.

[45] ——, "Interconnect planning with local area constrained retiming," in *Design Automation and Test Conference*, 2003, pp. 442–447.

[46] B. Xue, S. K. Shukla, and R. S. Sekharipuram, "Minimizing back pressure for latency insensitive system synthesis," in *International Conference on Formal Methods and Models for Co-Design*, 2010, pp. 189–198.

[47] G. Reehal and M. Ismail, "A Systematic Design Methodology for Low-Power NoCs," *IEEE Transaction On Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 12, pp. 2585–1595, 2014.

[48] *http://lpsolve.sourceforge.net/5.5/.*

[49] B. V. Cherkassky and A. V. Goldberg, *Negative-cycle detection algorithms.* Springer-Verlag, 1999.

[50] M. Chaturvedi and R. M. McConnel, "A Note On Finding Minimum Mean Cycle," *Information Processing Letters*, 2017.

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

[52] E. Moore, "The shortest path through a maze," in *International Symposium on the Theory of Switching*, 1959, pp. 285–292.

[53] N. Chandrachoodan, S. S. Bhattacharyya, and K. J. R. Liu, "Adaptive negative cycle detection in dynamic graphs," in *IEEE International Symposium on Circuits and Systems*, 2001.

[54] C.-W. A. Tsao and C.-K. Koh, "UST/DME: A clock tree router for general skew constraints," *ACM Transactions on Design Automation of Electronics System*, vol. 7, no. 3, pp. 359–379, 2002.

[55] J. P. Fishburn, "Clock skew optimization," *IEEE Transactions on Computers*, vol. 39, no. 7, pp. 945–951, 1990.

[56] R. Ewetz and C.-K. Koh, "Scalable Construction of Clock Trees With Useful Skew and High Timing Quality," *IEEE Transaction On Computer-Aided Design of Integrated Circuits And Systems*, vol. 38, no. 6, 2019.

[57] R.-S. Say, "Exact zero skew," in *International Conference on Computer-Aided Design*, 1991, pp. 336–339.

[58] J. Cong, A. B. Kahng, C.-K. Koh, and C.-W. A. Tsao, "Bounded-skew clock and Steiner routing," *ACM Transactions on Design Automation of Electronics System*, vol. 3, no. 3, pp. 341–388, 1998.

[59] I. H.-R. Jiang, C.-L. Chang, and Y.-M. Yang, “INTEGRA: Fast Multi-bit Flip-Flop Clustering for Clock Power Saving,” *IEEE Transaction On Computer-Aided Design of Integrated Circuits And Systems*, vol. 31, no. 2, pp. 192–204, 2012.

[60] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli, “A methodology for correct-by-construction latency insensitive design,” in *International Conference on Computer-Aided Design*, 1999.

[61] N. Uysal and R. Ewetz, “OCV Guided Clock Tree Topology Reconstruction,” in *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2018.

[62] C. Tan, R. Ewetz, and C.-K. Koh, “Clustering of Flip-Flops for Useful-Skew Clock Tree Synthesis,” in *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2018.

[63] R. Ewetz, “A Clock Tree Optimization Framework with Predictable Timing Quality,” in *Design Automation Conference (DAC)*, 2017.

[64] R. Ewetz and C.-K. Koh, “Clock Tree Construction based on Arrival Time Constraints,” in *International Symposium on Physical Design (ISPD)*, 2017.

[65] R. Ewetz, C. Tan, and C.-K. Koh, “Construction of Latency-Bounded Clock Trees,” in *International Symposium on Physical Design (ISPD)*, 2016.

[66] R. Ewetz and C.-K. Koh, “MCMM Clock Tree Optimization based on Slack Redistribution Using a Reduced Slack Graph,” in *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2016.

[67] R. Ewetz, S. Janarthanan, and C.-K. Koh, “Construction of Reconfigurable Clock Trees for MCMM Designs,” in *Design Automation Conference (DAC)*, 2015.

[68] R. Ewetz and C.-K. Koh, “Useful Skew Tree Framework for Inserting Large Safety Margins,” in *International Symposium on Physical Design (ISPD)*, 2015.

[69] R. Ewetz, S. Janarthanan, and C.-K. Koh, “Fast Clock Skew Scheduling based on Sparse-Graph Algorithms,” in *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2015.

[70] R. Ewetz and C.-K. Koh, “Local Merges for Effective Redundancy in Clock Networks,” in *International Symposium on Physical Design (ISPD)*, 2013.

[71] G. Wu, Y. Xu, D. Wu, M. Ragupathy, Y.-Y. Mo, and C. Chu, “Flip-flop Clustering by Weighted K-means Algorithm,” in *Design Automation Conference (DAC)*, 2016.

[72] D. Eppstein, “Finding the k shortest paths,” *SIAM J. Computing*, vol. 28, no. 2, pp. 652–673, 1998.

[73] M. R. Guthaus, W. Wilke, and R. Reis, “Revisiting Automated Physical Synthesis of High-Performance Clock Networks,” *ACM Transactions on Design Automation of Electronics System*, vol. 18, no. 2, 2013.

[74] C.-K. Koh, J. Jain, and S. F. Cauley, *Synthesis of clock and power/ground networks.* Electronic Design Automation, Morgan Kaufmann, 2009.

[75] R. B. Deokar and S. S. Sapatnekar, “A Graph-theoretic Approach to ClockSkew Optimization,” in *International Symp. on Circuits and Systems (ISCAS)*, 1994.

[76] D. Blaauw, K. Chopra, A. Srivastava, and L. Scheffer, “Statistical timing analysis: From basic principles to state of the art,” *IEEE Transaction On Computer-Aided Design of Integrated Circuits And Systems*, vol. 27, no. 4, pp. 589–607, 2008.

[77] J. P. Fishburn, “Clock skew optimization,” *IEEE Transactions on Computers*, vol. 39, no. 7, pp. 945–951, 1990.

[78] G. Ramalingam and T. Reps, “On the computational complexity of incremental algorithms,” *Tech. Rep. TR-1033, Computer Sciences Department, University of Wisconsin, Madison*, 1991.

[79] R. Ewetz, S. Janarthanan, and C.-K. Koh, “Benchmark circuits for clock scheduling and synthesis,” *https://purr.purdue.edu/publications/1759*, 2015.

[80] R. Tarjan, “Enumeration of the Elementary Circuits of a Directed Graph,” *SIAM J. Computing*, vol. 2, no. 3, pp. 589–607, 1973.

[81] A. P. Hurst, P. Chong, and A. Kuehlmann, “Physical Placement Driven by Sequential Timing Analysis,” in *International Conference on Computer-Aided Design*, 2004.