Experimental Analysis of the Fastest Optimum Cycle Ratio and Mean Algorithms

ALI DASDAN
Synopsys, Inc.

Optimum cycle ratio (OCR) algorithms are fundamental to the performance analysis of (digital or manufacturing) systems with cycles. Some applications in the computer-aided design field include cycle time and slack optimization for circuits, retiming, timing separation analysis, and rate analysis. There are many OCR algorithms, and since a superior time complexity in theory does not mean a superior time complexity in practice, or vice-versa, it is important to know how these algorithms perform in practice on real circuit benchmarks. A recent published study experimentally evaluated almost all the known OCR algorithms, and determined the fastest one among them. This article improves on that study in the following ways: (1) it focuses on the fastest OCR algorithms only; (2) it provides a unified theoretical framework and a few new results; (3) it runs these algorithms on the largest circuit benchmarks available; (4) it compares the algorithms in terms of many properties in addition to running times such as operation counts, convergence behavior, space requirements, generality, simplicity, and robustness; (5) it analyzes the experimental results using statistical techniques and provides asymptotic time complexity of each algorithm in practice; and (6) it provides clear guidance to the use and implementation of these algorithms together with our algorithmic improvements.

Categories and Subject Descriptors: B.5.2 [**Register-Transfer-Level Implementation**]: Design Aids—*Automatic synthesis*; *optimization*; B.6.3 [**Logic Design**]: Design Aids—*Automatic synthesis*; *optimization*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems; G.2.2 [**Discrete Mathematics**]: Graph Theory—*Graph algorithms*; *path and circuit problems*; G.2.3 [**Discrete Mathematics**]: Applications; J.6 [**Computer-Aided Engineering**]: Computer-Aided Design (CAD)

General Terms: Algorithms, Experimentation, Performance

Additional Key Words and Phrases: Cycle mean, cycle period, cycle ratio, cycle time, data flow graphs, discrete event systems, experimental analysis, iteration bound, system performance analysis

1. INTRODUCTION

Consider a finite directed cyclic graph $\mathcal{G}$ with n nodes and m arcs. Suppose that every arc in $\mathcal{G}$ is associated with two numbers (or weights): a real number called its *cost* and a nonnegative integer called its *transit time*. Let the cost

Author's address: Synopsys, Inc., M/S: US241, 700 East Middlefield Road, Mountain View, CA 94043; email: Ali.Dasdan@synopsys.com.

(respectively, transit time) of a cycle be equal to the sum of the costs (respectively transit times) of its arcs. Then, the (cycle) *ratio* of a cycle is defined to be equal to its cost divided by its transit time. If the transit time of a cycle is equal to its length, then the ratio of the cycle is also called its *mean*. Since "cycle ratio" is more general than "cycle mean", this article focuses on the former.

For some important performance-analysis related problems in the computed-aided design (CAD) field, the solution of the following fundamental problem is needed: find the optimum (maximum and/or minimum) cycle ratio (OCR) (respectively, cycle mean) of $\mathcal{G}$ over all its cycles. This problem is called the *optimum cycle ratio* (respectively, *cycle mean*) *problem*. This article mostly focuses on the minimum cycle ratio (MCR) problem and algorithms, and uses the terms OCR and MCR *interchangeably*. The reason is that a maximum cycle ratio problem can easily be reduced to MCR, as will be shown in Section 2.

1.1 Applications of Optimum Cycle Ratio Algorithms

The OCR algorithms are *fundamental* to the performance analysis of discrete event systems, hence, digital systems. Simply stated, it can be shown that the cycle period (or cycle time) of a discrete event system modeled by such a general model as $\mathcal{G}$ is equal to the optimum cycle ratio of $\mathcal{G}$. For a proof of this fact, see Bacelli et al. [1992], Burns [1991], and Ramamoorthy and Ho [1980]. This fact holds regardless of how $\mathcal{G}$ is interpreted as a model: it can be a Petri net [Ramamoorthy and Ho 1980], an event graph [Hulgaard et al. 1995], an event-rule system [Burns 1991], a process graph [Mathur et al. 1998], a signal transition graph [Nielsen and Kishinevsky 1994], and a data flow graph [Ito and Parhi 1995].

As for the specific applications of these algorithms in the CAD field (and the digital signal processing field), the following selected applications can be listed: the performance analysis of asynchronous systems (modeled as event-rule systems and Petri nets) [Burns 1991; Ramamoorthy and Ho 1980], that of synchronous or mixed systems [Teich et al. 1997], that of latency-insensitive systems (modeled as latency-insensitive graphs) [Carloni and Sangiovanni-Vincentelli 2000], the rate analysis of embedded real-time systems (modeled as process graphs) [Mathur et al. 1998], the iteration bound (or cycle period) of data flow graphs [Ito and Parhi 1995], the time separation analysis of concurrent systems (modeled as event graphs) [Hulgaard et al. 1995], the optimal clock schedules for circuits [Szymanski 1992], cycle time and slack optimization for circuits [Albrecht et al. 1999], retiming [Shenoy and Rudell 1994], and over-constraint resolution (for scheduling, layout compaction, etc.) [Dasdan 2002].

The OCR algorithms also have important applications in graph theory; for example, see Ahuja et al. [1993], Bacelli et al. [1992], Gondran and Minoux [1984], Hartmann and Orlin [1993], Lawler [1976], and Radzik and Goldberg [1994], and systems theory; for example, see Bacelli et al. 1992.

1.2 Previous Work and Motivation

There are many OCR algorithms; a complete list can be constructed by combining the lists given in Dasdan and Gupta [1998], Dasdan et al. [1999], Gondran

Table I. The Fastest Optimum Cycle Ratio Algorithms (Alg.) Compared in this Article

	Alg.	Original Source and the latest sources	Year	Time Comp. ("float" $\omega(\cdot)$)	Time Comp. ("int" $\omega(\cdot)$)	Based upon	Final Result
1	LAW	Lawler [1976]	1976	$O(nm \lg(W/\varepsilon))$	$O(nm \lg(nWT))$		Approximate
2	SZY	Szymanski [1992]	1992	$O(nm \lg(W/\varepsilon))$	$O(nm \lg(nWT))$	LAW	Approximate
3	TAR	Tarjan [1981] (and [Cherkassky and Goldberg 1996; Tarjan 1983])	1981	$O(nm \lg(W/\varepsilon))$	$O(nm \lg(nWT))$	LAW	Approximate
4	VAL	Howard [1960]	1960	$O(nmN_2)$	$O(n^4mWT^2)$	HOW	Exact
5	HOW	Howard [1960] (and [Cochet-Terrasson et al. 1998; Dasdan et al. 1999])	1960	$O(n^3mWT/\varepsilon)$	$O(n^4mWT^2)$		Exact
6	KO	Karp and Orlin [1981]	1981	$O(nm \lg(n))$			Exact
7	YTO	Young et al. [1991] (and [Skorobohatyj 1993])	1991	$O(nm + n^2 \lg(n))$		KO	Exact

These results are for a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{A}, \omega, \tau)$ with $n = |\mathcal{V}|$ nodes, $m = |\mathcal{A}|$ arcs, arc cost function ω, arc transit time function τ, the maximum arc cost W, the maximum arc transit time T, N_2 simple cycles, and the error bound $\varepsilon = 0.01$.

Note: LAW is not compared; it is needed to explain SZY and TAR.

and Minoux [1984], and Hartmann and Orlin [1993] and this article (Table I). Currently, for an input graph like $\mathcal{G}$, the fastest strongly polynomial-time algorithms [Dasdan and Gupta 1998; Hartmann and Orlin 1993] (improvements over Karp's algorithm [Karp 1978]) have a time complexity of $O(nm)$, and the fastest weakly polynomial-time algorithm [Orlin and Ahuja 1992] has a time complexity of $O(\sqrt{n}m \lg(nW))$. As shown in Dasdan et al. [1998, 1999], the superiority of these algorithms does not hold in practice.

Previous studies on the OCR algorithms focus almost exclusively on theory, without providing any experimental justification or implementation guidelines for the performance of the algorithms they propose. However, since the existing theory focuses only on the asymptotic worst-case time complexity analysis, it is well known that a superior time complexity in theory does not mean a superior time complexity in practice or vice-versa. For example, as shown in Dasdan et al. [1998, 1999], Karp's algorithm is slower than the algorithms HOW and KO in Table I although the *known* time complexity of HOW is exponential, and the time complexity of KO is worse than that of Karp's. Therefore, to find the "best" OCR algorithm for practical use in any application (including those in the CAD field), it is important to provide experimental analysis of the OCR algorithms. Our previous study[1] in Dasdan et al. [1998, 1999] is a significant step in that direction, and this article improves on it by focusing on the fastest MCR algorithms (including three new MCR algorithms), implementing them more carefully, comparing them on larger tests, and providing a better experimental evaluation and comparison.

Our previous study in Dasdan et al. [1998, 1999] is actually the only comprehensive evaluation and experimental comparison of the MCR algorithms. A few more comparisons are reported in Cochet-Terrasson et al. [1998], Dantzig et al. [1967], Dasdan and Gupta [1998], and Young et al. [1991], but they are very limited in scope, comparing a few algorithms on a number of small graphs.

[1]"We" and "our" refer to the authors of Dasdan et al. [1998, 1999].

In Dasdan et al. [1998, 1999], we evaluated and compared almost all the existing MCR algorithms. We implemented them using the LEDA library [Mehlhorn and Naher 1995]. LEDA is a reusable library of efficient data structures and algorithms, implemented in the C++ programming language. LEDA helped us tremendously in prototyping algorithms quickly, comparing them on a uniform basis, and optimizing their parameters. We performed the experimental comparison on a test suite that consisted of random graphs (r-tests) and graphs generated from some large circuit benchmarks in the ACM/SIGDA circuit benchmark suite (c-tests). For the r-tests, the size $(n + m)$ ranged from (512 + 512) to (8,192 + 24,576), and for c-tests, the size ranged from (274 + 388) of s344 to (24,255 + 34,876) of s38417. This experimental comparison has confirmed that the "best" MCR algorithm should have the following properties:

—*low practical time complexity*, that is, low time complexity in practice, which may not be apparent from the worst-case time complexity analysis.

—*linear space complexity*, which is needed to handle large graphs.

—*robustness*, which refers to the exactness of the final result and the stability of the results across the tests in the test suite.

—*generality*, which means the algorithm can work for both cycle ratio and mean problems.

—*simplicity*, which refers to the simplicity in understanding and coding.

All of these properties are obvious ones but most of the previous experimental work does not consider them together for algorithm evaluations. I will use all of these properties to evaluate the algorithms in this article.

1.3 Scope

For the study reported in this article (referred to as "this article" or "this study"), I first selected the best MCR algorithms from our previous study in Dasdan et al. [1998, 1999] and then re-implemented them in C++ without using LEDA. The most important reason for not using LEDA was that although LEDA provided a uniform base for prototyping, evaluating, and comparing algorithms, its generality imposed a relatively large memory and running time overhead (uniformly for all the MCR algorithms). This overhead prevented us from including larger (random) graphs in our previous study. My new implementation without it not only eliminated this overhead completely but also helped me discover improvements to the implementation of all the MCR algorithms.

The test suite again consisted of r-tests and c-tests but the new tests were far larger than those in our previous study: the size ranged from (12,752 + 36,681) to (1,048,576 + 3,407,872) (see Table II). We obtained our c-tests from the ISPD98 Circuit Benchmark Suite [Alpert 1998], which is still the largest public-domain circuit benchmark suite that I know of. After some experiments, I realized that some of the selected MCR algorithms (including Burns's algorithm [Burns 1991]) performed poorly on these large tests; therefore, I made another selection that reduced the number of the MCR algorithms to six.

1.4 Contributions and Summary of Results

This article evaluates and compares the selected six MCR algorithms on larger tests (in Section 4) and presents their results (in Section 5). I used sound statistical techniques to provide more reliable results. I believe that this article provides the best OCR algorithms to use in the CAD field. It also provides detailed guidelines to implement these algorithms efficiently.

The six MCR algorithms we used in this study are presented in Table I. Their names are abbreviated as follows: SZY, TAR, VAL, HOW, KO, and YTO. LAW, the 7th algorithm, is in the table only because it is needed to explain and understand TAR and SZY. Among the seven algorithms in this table, LAW, HOW, KO, and YTO are from our previous study, and VAL, TAR, and SZY are the new algorithms that are introduced for this study. All seven algorithms are discussed in sufficient detail in the following three sections: LAW, TAR, and SZY in Section 3.2, VAL and HOW in Section 3.3, and KO and YTO in Section 3.4.

In terms of the properties of "the best" MCR algorithm, this article contains the following results (from Section 5), one for each property:

—*low practical time complexity*: YTO has the fastest running time in practice, and YTO and KO have the fastest practical time complexity (their asymptotic time complexity as derived from their performance on the test suite). The practical time complexity of YTO and KO is $O(n \lg n)$, a factor of m faster than their worst-case theoretical time complexity. (see Section 5.4 and Section 5.5).

—*linear space complexity*: All the algorithms have linear space complexity although SZY, VAL, and HOW have the smallest constant factors in their space requirements as implemented (see Section 5.1).

—*robustness*: YTO is the most robust algorithm, and SZY followed by TAR are the least robust algorithms (see Section 5.1).

—*generality*: All the algorithms are equally general (see Section 5.1).

—*simplicity*: VAL and HOW are the simplest algorithms to code (see Section 5.1).

In our previous study, we evaluated the MCR algorithms in terms of their running time only and our data indicated HOW as the fastest MCR algorithm (although the best-known bound on its theoretical time complexity was and is still exponential). In this article, although HOW (and VAL) is almost as fast as YTO on the c-tests, our results point out YTO as the best MCR (hence, OCR) algorithm due to its superiority when all the properties above are considered together.

2. THEORETICAL PRELIMINARIES AND NOTATION

Let $\mathcal{G} = (\mathcal{V}, \mathcal{A}, \omega, \tau)$ be a finite directed graph with $|\mathcal{V}| = n$ nodes, $|\mathcal{A}| = m$ arcs, and two weight functions ω and τ that map arcs to numbers. Each arc $a = (u, v)$ is from one node u to another node v, and is mapped to an arbitrary number $\omega(a) = \omega(u, v)$ and a nonnegative integer $\tau(a) = \tau(u, v)$. Historically, the latter number for a is called its *transit time*, and the former number is called

its *cost* for minimization problems and its *profit* for maximization problems. Assume that W (respectively, T) is the largest absolute cost (respectively, transit time) value. See the references in Section 1.2 (e.g., Szymanski [1992]) on deriving the parameters of $\mathcal{G}$ if it represents a circuit.

A *path* in $\mathcal{G}$ is a forward or backward directed arc progression from an initial node to a final node. Its length is equal to the number of arcs it has. A *cycle* is a path of nonzero length in which the initial and final nodes are the same. A *simple path* is a path in which no node is repeated. If there is a path from one node u to another v, u is an *ancestor* of v, or equivalently, v is a *descendant* of u. If there is an arc $a = (u, v)$, then a (respectively, u) is a *predecessor* arc (respectively, node) of v, or equivalently, a (respectively, v) is a *successor* arc (respectively, node) of u.

A directed graph is *strongly connected* if there is a path between every pair of nodes.

2.1 Cycle Ratio and Mean

Let $\omega(\mathcal{P})$ (respectively, $\tau(\mathcal{P})$) denote the cost (respectively, transit time) of a path $\mathcal{P}$ in $\mathcal{G}$, where $\omega(\mathcal{P})$ (respectively, $\tau(\mathcal{P})$) are equal to the sum of the costs (respectively, transit times) of the arcs on $\mathcal{P}$. A cycle is positive, zero, or negative depending on the sign of its cost.

The *ratio* $\rho(\mathcal{P})$ of $\mathcal{P}$ is defined as

$$\rho(\mathcal{P}) = \frac{\omega(\mathcal{P})}{\tau(\mathcal{P})} = \frac{\sum_{a \in \mathcal{P}} \omega(a)}{\sum_{a \in \mathcal{P}} \tau(a)}, \ \tau(\mathcal{P}) > 0, \tag{1}$$

which basically gives the average arc cost per transit time for $\mathcal{P}$. When $\tau(\mathcal{P})$ is zero, $\rho(\mathcal{P})$ is defined to be infinity.

The *mean* $\lambda(\mathcal{P})$ of $\mathcal{P}$ is equal to its ratio when the transit time of each arc on $\mathcal{P}$ is unity, that is, when $\tau(\mathcal{P}) = |P|$, the length of $\mathcal{P}$. Thus, $\lambda(\mathcal{P})$ is defined as

$$\lambda(\mathcal{P}) = \frac{\omega(\mathcal{P})}{|\mathcal{P}|}, \ |\mathcal{P}| > 0, \tag{2}$$

which basically gives the average arc cost for $\mathcal{P}$.

When $\mathcal{P}$ is a cycle, its ratio (respectively, mean) is referred to as its *cycle ratio* (respectively, *cycle mean*). In the sequel, usually cycle ratios are used (see Gondran and Minoux [1984] for some applications of path ratios.)

Example 2.1. In Figure 1, the arc $3 \rightarrow 1$ has a cost of 7, a transit time of 2, and a length of 1. Thus, its ratio is $7/2 = 3.50$ and its mean is $7/1 = 7$. Similarly, the cycle $1 \rightarrow 2 \rightarrow 3 \rightarrow 1$ has a cost of 14, a transit time of 4, and a length of 3. Thus, its cycle ratio is $14/4 = 3.50$ and its cycle mean is $14/3 = 4.67$ (rounded up).

2.2 Optimum Cycle Ratio and Mean

If $\mathcal{G}$ is cyclic, it has a finite number of cycles; hence, the optimum cycle ratio (OCR) over all the cycles in $\mathcal{G}$ is well defined. The *minimum cycle ratio* (MCR) of $\mathcal{G}$ is denoted by $\rho^*_{\min}(\mathcal{G})$ (or ρ^* if $\mathcal{G}$ and "min" are clear from the context), and

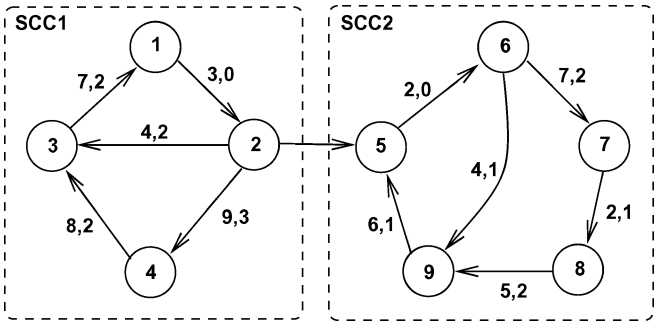

Fig. 1. An example graph with 9 nodes (circles), 12 arcs (arrows), and 2 strongly connected components (rectangles). The weight of each arc is its cost followed by its transit time.

is defined as

$$\rho^*_{\min}(\mathcal{G}) = \begin{cases} \min_{\forall \mathcal{C} \in \mathcal{G}}(\rho(\mathcal{C})) & \text{if } \mathcal{G} \text{ is cyclic} \\ \infty & \text{otherwise.} \end{cases} \tag{3}$$

The problem of finding the MCR of a given graph is called the *minimum cycle ratio problem* (the MCR problem), and an algorithm for solving it is called a *minimum cycle ratio algorithm* (a MCR algorithm). The maximum versions of these definitions are analogous. The optimum cycle mean versions of these definitions can be easily obtained by replacing "ratio" with "mean".

Example 2.2. In Figure 1, the graph has four cycles: $1 \to 2 \to 3 \to 1$, $1 \to 2 \to 4 \to 3 \to 1$, $5 \to 6 \to 9 \to 5$, and $5 \to 6 \to 7 \to 8 \to 9 \to 5$. Their ratios respectively are: 3.50, 3.86, 6.00, and 3.67 (rounded up). Their minimum is the MCR ρ^* and is 3.50.

The maximum cycle ratio can be computed using a MCR algorithm, as the following proposition shows. Its proof follows directly from Equation 1 and the fact that $\max\{x_1, x_2, \ldots, x_k\} = -\min\{-x_1, -x_2, \ldots, -x_k\}$ for any $k > 0$ real numbers. This proposition is the reason behind focusing on the MCR problem and algorithms.

PROPOSITION 2.3. *The maximum cycle ratio $\rho^*_{\max}$ of $\mathcal{G}$ is equal to the negative of its MCR when each arc cost in $\mathcal{G}$ is negated.*

2.3 Effect of Strong Connectivity

The directed graph $\mathcal{G}$ can be decomposed into its strongly connected components (SCCs) in $O(n+m)$ time [Cormen et al. 1991], which is linear in the *size* of $\mathcal{G}$. If each SCC of $\mathcal{G}$ is considered as a node in a new graph, the new graph is called the *component graph* of $\mathcal{G}$. Component graphs are necessarily acyclic.

The MCR of a graph $\mathcal{G}$ can be computed either directly or through its SCCs (by the associativity of "min"). In the latter case, the MCR of each SCC is computed, and then the minimum among these ratios is taken as the MCR of $\mathcal{G}$. This article used the latter case because it simplifies the algorithms and reduces their running times and space requirements. Moreover, decomposing into SCCs helps determine whether or not $\mathcal{G}$ is acyclic. If $\mathcal{G}$ is acyclic, there is no need to run

a MCR algorithm on $\mathcal{G}$ because its MCR is ∞ by Definition 3, and the algorithm will return the same result anyway.

Example 2.4. In Figure 1, $\rho^*(SCC1) = \min\{3.50, 3.86\} = 3.50$ and $\rho^*(SCC2) = \min\{6.00, 3.67\} = 3.67$. Thus, $\rho^* = \min\{\rho^*(SCC1), \rho^*(SCC2)\} = \{3.50, 3.67\} = 3.50$.

2.4 Bounds on Cycle Ratio

LAW, TAR, and SZY need lower and upper bounds (hence, an interval) on the MCR. The smaller the resulting interval is, the faster these algorithms are. Such bounds are derived below.

Let $\mathcal{A}_0 = \{al\ a \in \mathcal{A}$ and $\tau(a) = 0\}$. Depending on whether or not this subset of $\mathcal{A}$ is empty, the following proposition gives lower and upper bounds on the minimum and maximum cycle ratios. In both cases, it shows that the ratio of any cycle is sandwiched between these ratios. Its proof follows directly from the definition of the cycle ratio.

PROPOSITION 2.5. *Let $\mathcal{C}$ be any cycle in $\mathcal{G}$. Then, if $\mathcal{A}_0$ is empty,*

$$\min_{a\in\mathcal{A}}\left\{\frac{\omega(a)}{\tau(a)}\right\} \le \rho^*_{\min} \le \rho(\mathcal{C}) \le \rho^*_{\max} \le \max_{a\in\mathcal{A}}\left\{\frac{\omega(a)}{\tau(a)}\right\}; \tag{4}$$

Otherwise,

$$-mW \le -S_\omega \le \rho^*_{\min} \le \rho(\mathcal{C}) \le \rho^*_{\max} \le S_\omega \le mW, \tag{5}$$

where $S_\omega = \sum_{a\in\mathcal{A}} |\omega(a)|$, the sum of the arc costs.

Note that all the bounds in this proposition, including the bound due to the ratio of any cycle (see Section 3.1 for this), can be computed in linear time.

Remark 2.6. For my test suite, $\mathcal{A}_0$ was empty; hence, the left-hand side of Eq. (4) was used to initially bound $\rho^*_{\min}$:

$$\min_{a\in\mathcal{A}}\left\{\frac{\omega(a)}{\tau(a)}\right\} \le \rho^*_{\min} \le \rho(\mathcal{C}), \tag{6}$$

where $\rho(\mathcal{C})$ was computed as described in Remark 3.1 in Section 3.1.

2.5 Graph Transformation

Define $\mathcal{G}_r$ as $\mathcal{G}_r = (\mathcal{V}, \mathcal{A}, \omega - r\tau, \tau)$ where $(\omega - r\tau)(a) = \omega(a) - r\tau(a)$ for any arc a. This graph is the same as $\mathcal{G}$ except that its arc weight function is changed to incorporate a parameter r.

Two propositions below follow directly from the definitions above. Proposition 2.7 below gives the connection between the MCR and the existence of negative cycles. Proposition 2.8 below shows how the cycle ratio depends on the arc weight function.

PROPOSITION 2.7. *$\mathcal{G}$ has a negative cycle if $\rho^*(\mathcal{G}) < 0$, or it has no negative cycles if $\rho^*(\mathcal{G}) \ge 0$.*

PROPOSITION 2.8. *For any parameter r (a real number), $\rho(\mathcal{C}) - r = (\omega(\mathcal{C}) - r\tau(\mathcal{C}))/(\tau(\mathcal{C}))$.*

These two propositions imply the following corollary, which is used in all the MCR algorithms.

COROLLARY 2.9. *$\mathcal{G}_r$ has a negative cycle if $r > \rho^*(\mathcal{G})$, a zero cycle if $r = \rho^*(\mathcal{G})$, or a positive cycle if $r < \rho^*(\mathcal{G})$. In other words, if $r \leq \rho^*(\mathcal{G})$, no cycles in $\mathcal{G}_r$ are negative.*

2.6 Linear Programming Formulation

To explain the behavior of some of the algorithms better, the following linear programming formulation of the MCR problem is needed. It is actually the dual of the original linear program for this problem, for example, see Cuninghame-Green and Yixun [1996], Dantzig et al. [1967], and Hartmann and Orlin [1993].

This formulation of the MCR problem implies that the MCR ρ^* of $\mathcal{G}$ is the optimal solution (the optimal r) of the following linear program:

$$\begin{aligned} &\textbf{max } r \quad \textbf{subject to} \\ &\qquad d(v) \leq d(u) + (\omega(u,v) - r\tau(u,v)), \\ &\qquad \textbf{for every arc } (u,v) \in \mathcal{A}, \end{aligned} \tag{7}$$

where $d(v)$ is called the *distance* of v. For TAR, SZY, KO and YTO, the distance of v corresponds to the cost of a minimum-cost path from some source node to v; for VAL and HOW, the distance of v corresponds to the cost of a minimum-cost path from v to other nodes. The meaning will further be clarified in the context of the algorithms.

In the sequel, one update of $d(v)$ to satisfy the above distance constraint is called a *tightening* (or relaxing [Cormen et al. 1991]).

This formulation shows the close relation between the MCR problem and the shortest paths problem. For a fixed r and an arc weight function of $(\omega - r\tau)$, the distance constraints in Eq. (7) are the necessary optimality constraints for the shortest paths problem [Cormen et al. 1991]. The addition of the parameter r leads to a generalization of the shortest paths problem called *the parametric shortest paths problem*. This problem seeks to find the largest value of the parameter r below which $\mathcal{G}_r$ has no negative cycles. This largest value of r is exactly the MCR $\rho^*(\mathcal{G})$. As such, the MCR problem and the parametric shortest paths problems are equivalent. KO and YTO algorithms are parametric shortest path algorithms.

3. ALGORITHMS

This section first describes an efficient cycle detection algorithm that is used as a subroutine by the MCR algorithms. It then allocates one subsection to each group of algorithms: (1) LAW, TAR, and SZY; (2) VAL and HOW; (3) KO and YTO. Each algorithm takes a strongly connected graph $\mathcal{G}$ as an input. For each algorithm, an overview is given and important points are mentioned. It is simply impossible to describe each algorithm in detail in this article. Interested readers should consult the cited references in each subsection.

```
FIND-RATIO(G = (V, A, ω, τ), r, p)
 1. for each node v ∈ V do
 2.   visited(v) = NIL /* Nodes are not visited initially. */
 3. r' = r; handle = NIL
 4. for each node v ∈ V do
 5.   if visited(v) ≠ NIL then
 6.     continue /* v is visited before. */
      /* Search for a new cycle. Mark nodes with v: */
 7.   u = v
 8.   do
 9.     visited(u) = v; u = p(u)
10.   while visited(u) = NIL
11.   if visited(u) ≠ v then
12.     continue /* u is on an old cycle. */
      /* u is on a new cycle. Find the cycle ratio: */
13.   x = u; sum = 0; len = 0
14.   do
15.     sum = sum + ω(p(x), x); len = len + τ(p(x), x)
16.     u = p(u)
17.   while x ≠ u
      /* Find and return min{r, the cycle ratio}: */
18.   if r' > sum/len then
19.     r' = sum/len; handle = u
20. return (r', handle)
```

Fig. 2. An algorithm to compute the minimum of r and $\rho^*(\mathcal{G}_p)$. The **continue** statement behaves as in C/C++: it jumps to the next iteration of the enclosing loop statement. Line 14 assumes that $p(x)$ is a predecessor of x. If it is a successor, then $(p(x), x)$ should be written as $(x, p(x))$.

3.1 Efficient Cycle Detection

Cycle detection is essential to speed up the MCR algorithms (also the negative cycle detection algorithms [Cherkassky and Goldberg 1996]). It is used in TAR, SZY, VAL, and HOW. It can also be used in many other MCR algorithms, for example, in LAW (with BELLMAN-FORD) to find a negative cycle quickly (also see Shenoy and Rudell [1994] for a similar application), or in Karp's algorithm [Karp 1978] to detect early convergence [Hartmann and Orlin 1993].

The most important requirement for cycle detection is fast running time. It is achieved by performing cycle detection on a predecessor graph (for SZY) or a successor graph (for VAL and HOW) of $\mathcal{G}$. A predecessor (respectively, successor) graph is obtained by choosing only one of the predecessor (respectively, successor) arcs of each node in $\mathcal{G}$ (using a predecessor function p).

Denote the predecessor (respectively successor) graph of $\mathcal{G}$, which is strongly connected, induced by p using $\mathcal{G}_p$. Since every node has a predecessor node in $\mathcal{G}_p$, this graph has at least one cycle. Thus, the MCR of $\mathcal{G}_p$ can be found in $\Theta(n)$ time using the algorithm FIND-RATIO in Figure 2.

FIND-RATIO returns two values: r' and *handle*. If $\rho^*(\mathcal{G}_p)$ is larger than r, *handle* is *NIL*; otherwise, $r' = \rho^*(\mathcal{G}_p)$ and *handle* is an arbitrary node on the cycle whose ratio is equal to the MCR (called a *critical cycle*).

Remark 3.1. Suppose for each node v in $\mathcal{G}$, the graph $\mathcal{G}_p$ contains the predecessor node $p(v)$ such that $(p(v), v)$ has the smallest cost among all the

```
LAW(G = (V, A, ω, τ), r, ε)
1. Find [L,U] by Equation 6 in § 2.4.
2. if r ≤ L then
3.   return r
4. r' = U = min{U, 2 * r − L}
5. while (U − L) > ε do
6.   r' = (U + L)/2
7.   (result, r') = FIND-NEG-CYCLE(G, r')
8.   if result = "found" then
9.     U = r'
10.  else
11.    L = r'
12. return r'
```

Fig. 3. Lawler's minimum cycle ratio algorithm (LAW). Calling Tarjan's (respectively, Szymanski's) negative cycle detection algorithm TARJAN (respectively, SZYMANSKI) at line 7 leads to Tarjan's (respectively, Szymanski's) minimum cycle ratio algorithm TAR (respectively, SZY).

predecessor arcs of v. Then, the MCR of $\mathcal{G}_p$ as computed by FIND-RATIO provides a very good initial upper bound on the MCR of $\mathcal{G}$. This bound provides the upper bound in Eq. (6), and it is what we used in this study.

3.2 Lawler's Algorithm (LAW), Tarjan's Algorithm (TAR), and Szymanski's Algorithm (SZY)

LAW [Lawler 1976] is given in Figure 3. With the help of a negative cycle detection algorithm FIND-NEG-CYCLE, LAW performs a binary search over the possible values of ρ^* in the (search) interval $[L, U]$. LAW iterates lines 5–11 until the search interval is small enough, that is, it is smaller than a user-defined value ε. In my experiments, $\varepsilon = 0.01$ for a reasonable precision and fast running times. In each iteration (called a pass), LAW first designates the middle value r' of the search interval as a possible candidate for ρ^* (line 6). LAW then invokes FIND-NEG-CYCLE to transform $\mathcal{G}$ to $\mathcal{G}_{r'}$ and check for a negative cycle (line 7). By Corollary 2.9, if $\mathcal{G}_{r'}$ has a negative cycle, r' is larger than ρ^*, so the next search interval is set to $[L, r']$ (line 9); Otherwise, r' is smaller than ρ^*, and so the next search interval is set to $[r', U]$ (line 11).

Note that, by Proposition 2.5 and Remark 3.1, the ratio of the negative cycle found by FIND-NEG-CYCLE provides an upper bound on ρ^*; hence, r' can be set to that ratio to speed up the search (line 7). This was what I did in my implementation.

LAW can use any negative cycle detection algorithm to implement FIND-NEG-CYCLE. In our previous study [Dasdan et al. 1998, 1999], we used the Bellman–Ford algorithm (BELLMAN-FORD) [Cormen et al. 1991] (which the reader is assumed to be familiar with). However, there are more efficient negative cycle detection algorithms in practice than BELLMAN-FORD: Tarjan's and Szymanski's negative cycle detection algorithms are among the fastest (see Cherkassky and Goldberg [1996] for another fast algorithm called the Goldberg-Radzik algorithm). Tarjan's (respectively, Szymanski's) negative cycle detection algorithm TARJAN (respectively, SZYMANSKI) is explained in Section 3.2.1

```
FIND-NEG-CYCLE(G = (V, A, ω, τ), r) /* TARJAN */
1. for each node v ∈ V do
2.   d(v) = ∞
3. d(s) = 0 /* s is arbitrary but fixed. */
4. T_p(s) = {s}
5. ENQUEUE(Q, s)
6. while Q ≠ φ do
7.   u = DEQUEUE(Q)
8.   for each arc (u, v) ∈ A do
9.     if d(v) > d(u) + ω(u, v) − r ∗ τ(u, v) then
10.      d(v) = d(u) + ω(u, v) − r ∗ τ(u, v)
11.      if u ∈ T_p(v) then
12.        (r', handle) = FIND-RATIO(G, r, p)
13.        return ("found", r')
14.      Delete from Q each x ∈ T_p(v)
15.      Delete T_p(v)
16.      p(v) = u
17.      ENQUEUE(Q, v)
18.return ("not found", r)
```

Fig. 4. Tarjan's negative cycle detection algorithm TARJAN. This algorithm combined with LAW constitutes Tarjan's minimum cycle ratio algorithm TAR.

(respectively, Section 3.2.2). LAW together with TARJAN (respectively, SZYMANSKI) make up Tarjan's (respectively, Szymanski's) MCR algorithm TAR (respectively, SZY).

The performance of LAW (hence, TAR and SZY) depends on the size of the search interval (line 1). The best search interval that I could devise is given by Remark 2.6.

The time complexity analysis of LAW, TAR, and SZY are the same; it is equal to the time complexity of FIND-NEG-CYCLE multiplied by the number of passes over the search interval. Currently, the best time complexity for FIND-NEG-CYCLE is $O(mn)$. The following theorem gives the time complexity of all three algorithms.

THEOREM 3.2 (Lawler 1976; Szymanski 1992; Tarjan 1983). *LAW, TAR, and SZY all run in* $O(nm \lg(W/\varepsilon))$ *time for arbitrary arc costs, and in* $O(nm \lg(nWT))$ *time for integral arc costs.*

3.2.1 *Tarjan's Negative Cycle Detection Algorithm (TARJAN).* Tarjan's negative cycle detection algorithm TARJAN [Tarjan 1981] is given in Figure 4. Cherkassky and Goldberg [1996] cites Tarjan [1981] as the source of TARJAN and shows that it is one of the fastest negative cycle detection algorithms. My use of TARJAN to implement FIND-NEG-CYCLE in LAW and create Tarjan's minimum cycle ratio algorithm TAR is in part due to A. Goldberg (personal communication), Mehlhorn and Naher [2000], and Tarjan [1983]. An algorithm similar to TAR is also given in Chandrachoodan et al. [1999].

Like BELLMAN-FORD [Cormen et al. 1991], TARJAN first defines the distance $d(u)$ as an upper bound on the cost of the shortest path from some arbitrary source node s to node u. Initially, the distance of s is zero (line 3), and

```
FIND-NEG-CYCLE(G = (V, A, ω, τ), r) /* SZYMANSKI */
1. /* 0th iteration */
2. for each node v ∈ V do
3.   d(v) = ∞; p(v) = s
4. d(s) = 0 /* s is arbitrary but fixed. */
5. for each iteration k = 1 to |V| do
6.   for each node u ∈ V do
7.     if d(u) is tightened in iteration k − 1 (of line 5) do
8.       for each arc (u, v) ∈ A do
9.         if d(v) > d(u) + ω(u, v) − r ∗ τ(u, v) then
10.          d(v) = d(u) + ω(u, v) − r ∗ τ(u, v); p(v) = u
11.  if d(s) < 0 then
12.    return ("found", r)
13.  if no d is tightened at line 10 then
14.    return ("not found", r)
15.  if every 10 iterations then
16.    r' = FIND_MEAN(G, r, p)
17.    if r' < r then
18.      return ("found", r')
```

Fig. 5. Szymanski's negative cycle detection algorithm SZYMANSKI. This algorithm combined with LAW constitutes Szymanski's minimum cycle ratio algorithm SZY.

that of any other node is infinity (line 2). TARJAN then goes over the nodes tightening their distances (line 10). The nodes are stored in a FIFO queue Q, which provides ENQUEUE and DEQUEUE operations for node insertion and deletion, respectively.

Unlike BELLMAN-FORD, TARJAN maintains an explicit shortest paths tree T_p induced on $\mathcal{G}$ by the predecessor function p (as defined in Section 3.1). The tree T_p is rooted at s, and $T_p(v)$ denotes the subtree rooted at any node v. After $d(v)$ is tightened in line 10, BELLMAN-FORD simply enqueues v in Q. Instead, TARJAN utilizes the following two facts [Mehlhorn and Naher 2000].

First, once a shorter path to v is discovered, shorter paths exist for all nodes in $T_p(v)$, and there is no need to update the distances of these nodes. Then, they are deleted from T_p and Q (lines 14–15) (actually marked to be ignored at line 7), and v is inserted into Q to reach these nodes later (line 17). Second, if u is an ancestor of v in T_p, a negative cycle is detected. Then, r can be improved using the ratio of the negative cycle found (by Proposition 2.5); FIND-RATIO is used for that purpose (line 12).

3.2.2 *Szymanski's Negative Cycle Detection Algorithm (SZYMANSKI).* Szymanski's negative cycle detection algorithm SZYMANSKI is given in Figure 5. My implementation of SZYMANSKI and its use to implement FIND-NEG-CYCLE in LAW and create Szymanski's MCR algorithm SZY are due to T. Szymanski (personal communication), and Szymanski [1992].

Like BELLMAN-FORD [Cormen et al. 1991], SZYMANSKI first defines the distance $d(u)$ as an upper bound on the cost of the shortest path from some arbitrary source node s to node u. Initially, the distance of s is zero (line 4), and that of any other node is infinity (line 3). SZYMANSKI then goes over the nodes tightening their distances (line 10).

```
HOW(G = (V, A, ω, τ), r, ε)
1. for each node v ∈ V do
2.   d(v) = ∞
3. for each arc (u, v) ∈ A do
4.   if ω(u, v) < d(u) then
5.     d(u) = ω(u, v); p(u) = v
6. while (true) do
7.   < r, handle > = FIND-RATIO(G, r, p)
8.   if handle ≠ NIL then
9.     if ∃ a path from v to handle in G_p then
10.      /* In backward BFS order from handle */
11.      d(v) = d(p(v)) + ω(v, p(v)) − r * τ(v, p(v))
12.  changed = false
13.  for each arc (u, v) ∈ A do
14.    if d(u) > d(v) + ω(u, v) − r * τ(u, v) + ε then
15.      d(u) = d(v) + ω(u, v) − r * τ(u, v)
16.      p(u) = v
17.      changed = true
18.  if not changed then
19.    return r
```

Fig. 6. Howard's minimum cycle ratio algorithm (HOW). VAL excludes lines 8-11. BFS stands for the standard Breadth First Search algorithm.

Unlike BELLMAN-FORD, SZYMANSKI maintains an explicit predecessor graph induced on $\mathcal{G}$ by the predecessor function p. SZYMANSKI does not use a node queue either.[2] Instead, SZYMANSKI utilizes the following facts [Szymanski 1992] (see Cherkassky and Goldberg [1996], and Lawler [1976] for more "facts").

First, if $d(u) < 0$, there is a negative cycle (lines 11–12). For efficiency, this is checked only for the source node s. Second, if $d(u)$ is not tightened in the current iteration (of line 5), u cannot affect the distances of its successors in the next iteration (line 7). Third, if no distances are tightened in the current iteration, all the distance constraints are satisfied, and there is no negative cycle (lines 13–14). Finally, the predecessor graph contains cycles whose ratios can be used to improve r (lines 15–18) (by Proposition 2.5); FIND-RATIO is used for that purpose (lines 16) at certain intervals. The original implementation in T. Szymanski (personal communication) invokes FIND-RATIO once every 10 passes and once at the end of the passes. These were "the times for cycle detection" (line 15) in my implementation too.

3.3 Howard's Algorithms VAL and HOW

HOW is given in Figure 6. VAL has the same pseudocode except the lines 8–11. Howard's original algorithms are proposed in Howard [1960] for Markov decision processes. His algorithm that was the basis of VAL (respectively, HOW) is called a "value iteration algorithm" (respectively, a "policy iteration algorithm").

[2]One experiment has showed that SZYMANSKI with a node queue can perform fewer number of iterations (of line 5) than SZYMANSKI without one; yet, it performed worse probably because the loops at lines 5–10 are easier for the compiler to optimize without a node queue.

His policy iteration algorithm is first adapted to the MCR problem in Cochet-Terrasson et al. [1998], and then improved in Dasdan et al. [1998, 1999]. As these improvements were not well documented in Dasdan et al. [1998, 1999], they will be explained more clearly in the sequel. HOW and VAL in this article are based on the improved version in Dasdan et al. [1998, 1999]. The focus below is on HOW as VAL is almost identical.

HOW maintains an explicit successor graph $\mathcal{G}_p$ (induced by the successor function p). This graph is initialized in lines 1–5. Let a pass refer to one iteration of the loop of line 6. In the beginning of a pass (line 7), HOW uses r as a candidate for the MCR ρ^*. In Cochet-Terrasson et al. [1998], r is initialized with the MCR of an *arbitrary* cycle in $\mathcal{G}_p$; in Dasdan et al. [1998, 1999], we improved this by initializing r with the MCR of $\mathcal{G}_p$ (which is what FIND-RATIO returns at line 7). This improvement ensures that r decreases in larger strides; together with line 8, it also leads to another improvement: r is made to monotonically decrease towards ρ^*.

The rest of HOW is similar to BELLMAN-FORD. In fact, HOW itself can be considered as BELLMAN-FORD with a cycle detection routine. In lines 9–11, HOW treats *handle* as a source node and sets the distance of every other node v (or only those that reach *handle*) to the cost of the actual path from v to *handle* on $\mathcal{G}_p$. This step is missing in VAL: it is needed for faster convergence (as HOW performs faster than VAL) rather than for correctness. After this step, HOW runs lines 13–17, during which each distance constraint in Eq. (7) is checked for satisfiability. Note that these lines are identical to the inner loop of BELLMAN-FORD. If every distance constraint is satisfied (line 18), HOW returns r as the MCR ρ^* (line 19). Otherwise, unsatisfied distance constraints are tightened and $\mathcal{G}_p$ is updated (lines 15–16).

With respect to the above paragraph, we had two more improvements in Dasdan et al. [1998, 1999] over the algorithm in Cochet-Terrasson et al. [1998]: (1) we implemented lines 9–11 using a backward breadth first search whereas Cochet-Terrasson et al. [1998] uses a more complicated scheme; (2) we used the updated distance values in the remaining iterations of the loop of line 13 whereas Cochet-Terrasson et al. [1998] uses them only after the loop of line 13 ends. These two improvements led to simplicity and efficiency.

Although r decreases monotonically, it does not decrease every pass; it decreases whenever a negative cycle is detected (implying that the current value of r is too large), and it takes at most n passes to detect one. This fact is stated in Lemma 3.3 and is proved in Dasdan et al. [1998]. Intuitively, it holds because if r does not change, the lines 8–11 do not change the distances and what remains in a pass is the loop of line 13. Since this loop is identical to the inner loop of BELLMAN-FORD, n iterations of this loop will find a negative cycle (e.g., see Cormen et al. [1991] for a proof for BELLMAN-FORD). This fact holds for both HOW and VAL.

LEMMA 3.3. *In HOW and VAL, the cycle ratio r decreases every n iterations of line* 6 *(every n passes).*

To find the time complexity of HOW and VAL, we also need to find how much r decreases each time it decreases. The following lemma is for that purpose and

is proved in Dasdan et al. [1998]. The basic idea in the proof goes as follows: Suppose r decreases due to a negative cycle $\mathcal{C}$ and an arc (u, v) of this cycle has passed the test at line 14. Then, the difference $d(u) - (d(v) + \omega(u, v) - r\tau(u, v))$ is larger than ε. But, for the other arcs that have failed this test, this difference is zero due to line 11. Summing these differences for all the arcs of $\mathcal{C}$ eliminates the d values and gives its ratio. Then, its ratio is smaller than r by at least $\varepsilon/\tau(\mathcal{C})$ (or by at least $\varepsilon/(nT)$ in the worst case).

LEMMA 3.4. *In HOW, the cycle ratio r decreases by at least $\varepsilon/(nT)$ every n iterations of line* 6. *In other words, r decreases by at least $\varepsilon/(nT)$ in $O(nm)$ time (as each iteration of line* 6 *takes $\Theta(m)$ time).*

For this lemma to hold, line 11 has to be performed for every node of $\mathcal{G}_p$, that is, paths must exist from every node of $\mathcal{G}_p$ to *handle*. Since VAL lacks the lines 8–11, I have been unable to prove that Lemma 3.4 also holds for VAL. I will leave this as an open problem.

Using this lemma and noting that by Proposition 2.5, r is at most nW, r has to decrease as many as $nW/(\varepsilon/(nT)) = n^2WT/\varepsilon$ times until convergence. Consequently, HOW runs in $O(n^3mWT/\varepsilon)$ time. This bound is exponential in the worst case if any of W, T, and ε is exponential in the size of $\mathcal{G}$. Unfortunately, all the other known bounds on HOW are exponential too. All the known bounds are listed in the following theorem.

THEOREM 3.5. *Let W (respectively, T) be the absolute value of the maximum arc cost (respectively, transit time) in $\mathcal{G}$.*

(1) *The time complexity of HOW and VAL is $O(N_1)$ where N_1 is the total number of successor graphs on $\mathcal{G}$.*
(2) *The time complexity of HOW and VAL is at most $O(nmN_2)$ where N_2 is the number of simple cycles in $\mathcal{G}$.*
(3) *The time complexity of HOW is at most $O(n^3mWT/\varepsilon)$ for arbitrary arc costs.*
(4) *The time complexity of HOW and VAL is at most $O(n^4mWT^2)$ for integral arc costs.*

In Theorem 3.5, the first two bounds are trivial bounds from Cochet-Terrasson et al. [1998] and Dasdan et al. [1998], respectively. The third bound follows from Lemma 3.4. The fourth bound is a *new bound* this article introduces. Its proof follows from a fact stated in Lawler [1976]: $\mathcal{G}$ has at most $2n^3WT^2$ distinct cycle ratios when the arc costs are integral.

Note that the third bound does not hold for VAL. The experimental results suggest that VAL and HOW both run in $O(nm)$. This is stated as a conjecture (for HOW only) in Cochet-Terrasson et al. [1998]. The proof of this conjecture is still open (but see Madani [2000] for some ideas for VAL).

3.4 Karp and Orlin's Algorithm (KO) and Young, Tarjan, and Orlin's Algorithm (YTO)

YTO [Young et al. 1991] is given in Figure 8. YTO is almost identical to KO [Karp and Orlin 1981] as YTO was originally proposed as an asymptotically efficient

```
FIND-ARC-KEY(a = (x, y), G = (V, A, ω, τ))
1. Δt = t(x) + τ(a) − t(y)
2. if Δt > 0 then
3.    Δd = d(x) + ω(a) − d(y)
4.    return Δd/Δt
5. else
6.    return ∞
```

Fig. 7. A subroutine of YTO to compute the key of a given arc a.

implementation of KO. Since Young et al. [1991] does not give a pseudocode for YTO, a pseudocode for YTO is given Figure 8. A pseudocode for KO can be found in Karp and Orlin [1981].

YTO is a parametric shortest path algorithm. As explained in Section 2.6, YTO tries to find the largest value of r below which $\mathcal{G}_r$ has no negative cycles. When it finds that value, r is maximum and equal to $\rho^*(\mathcal{G})$.[3]

YTO associates a node key nk with every node and an arc key ak with every arc. The node key of a node v is the arc with the minimum arc key among the predecessor arcs of v. The arc key of an arc $a = (u, v)$ is computed by FIND-ARC-KEY in Figure 7.

YTO operates on a finite sequence of shortest paths trees. Each tree uses the predecessor function p. The initial tree is constructed by first adding a new node s to $\mathcal{G}$ and second adding n arcs from s to every other node in $\mathcal{V}$ (lines 1–3 in Figure 8). My implementation actually does not add these arcs but only simulates their effect. At an iteration of line 14 (called a pass), YTO tries to create a new shortest path tree T_p by replacing the predecessor of v in T_p by (u, v). Here, (u, v) is the arc with the minimum arc key in H (line 15), which is a priority queue data structure for maintaining node and arc keys. If YTO discovers a cycle (line 16), then the ratio of this cycle, which is equal to r of line 15, is minimum, that is, it is equal to ρ^*. Otherwise, YTO constructs the new tree at lines 18–34. The shortest path tree T_p at a pass stays as a shortest path tree in $\mathcal{G}_r$ for all r between r of this pass and that of the next pass.

During the construction of the new shortest path tree, lines 18–20 updates the distance and transit time of each node in $T_p(v)$, the subtree rooted at v. Line 21 changes the predecessor of v to u. Lines 22–28 updates the keys of each arc entering $T_p(v)$, and lines 29–34 updates the keys of each arc leaving $T_p(v)$. In lines 22–24, node keys can also get updated if necessary (lines 27 and 33).

As mentioned above, YTO uses a priority queue data structure H, which is keyed on arc keys. The original YTO uses a Fibonacci heap to implement H. In fact, the use of Fibonacci heaps together with node keys was what created YTO out of KO and reduced the time complexity of KO. In our previous study [Dasdan et al. 1998, 1999], we implemented YTO using a Fibonacci heap but realized that this heap was not fast enough in practice (also observed in

[3]Burns's algorithm [Burns 1991] behaves similar to YTO; however, it builds $\mathcal{G}_r$ from scratch every iteration whereas YTO updates this graph incrementally. This, in part, explains the poorer performance of Burns's algorithm. The other reason is the fact that it uses lots of floating-point arithmetic operations.

```
YTO(G = (V, A, ω, τ))
1. V = V ∪ {s}
2. A = A ∪ {(s, u)|ω(s, u) = 0}
3. T_p(s) = {(s, v)|(s, v) ∈ A}
4. for each node v ∈ V do
5.   d(v) = 0; t(v) = 1; nk(u) = NIL
6. t(s) = 0
7. for each arc a ∈ A do
8.   ak(a) = FIND-ARC-KEY(a, G)
9.   if nk(v) = NIL or ak(a) < ak(nk(v)) then
10.    nk(v) = a
11.PQ-INSERT(H, < ∞, − >)
12.for each node v ∈ V do
13.  PQ-INSERT(H, < ak(nk(v)), nk(v) >)
14.while (true) do
15.  < r, (u, v) >= PQ-FIND-MIN(H)
16.  if r = ∞ or u ∈ T_p(v) then
17.    return r
18.  for each node x ∈ T_p(v) do
19.    d(x) = d(x) + (d(u) + ω(u, v) − d(v))
20.    t(x) = t(x) + (t(u) + τ(u, v) − d(v))
21.  p(v) = u
22.  for each node x ∈ T_p(v) do
23.    ak(nk(x)) = ∞
24.    for each arc a = (x, y) entering T_p(v) do
25.      ak(a) = FIND-ARC-KEY(a, G)
26.      if ak(a) ≤ ak(nk(y)) then
27.        nk(y) = a
28.    PQ-UPDATE(H, < ak(nk(y)), nk(y) >)
29.  for each node x ∈ T_p(v) do
30.    for each arc a = (x, y) leaving T_p(v) do
31.      ak(a) = FIND-ARC-KEY(a, G)
32.      if ak(a) < ak(nk(y)) then
33.        nk(y) = a
34.        PQ-UPDATE(H, < ak(nk(y)), nk(y) >)
```

Fig. 8. Young-Tarjan-Orlin's minimum cycle ratio algorithm YTO. FIND-ARC-KEY is described in the next figure. Karp-Orlin's algorithm KO is similar.

Goldberg [2001]). In this study, I implemented H using a binary heap; thus, my implementation of YTO has the same time complexity as KO. Lines 11, 13, and 15 in the pseudocode contain two standard operations of H (e.g., see Cormen et al. [1991]): PQ-INSERT and PQ-FIND-MIN; PQ-UPDATE at lines 28 and 34 is a notational shorthand for calling PQ-DELETE followed by PQ-INSERT on the same key.

4. EXPERIMENTAL ENVIRONMENT

Our previous study in Dasdan et al. [1998, 1999] is the first comprehensive experimental study of the MCR problem. To the best of my knowledge, I do not know of any other comprehensive study since then. Therefore, I discuss the experimental environment in detail so that it can serve as a better reference for future studies.

4.1 Computer Setup

I built all the programs using the GNU g++ compiler (version 2.95.2). I performed all my experiments on one CPU of a 4-CPU SUN workstation computer running the UNIX operating system (SunOS version 5.7). Each CPU was a 400 MHz UltraSPARC-II processor. This processor has 16 KB instruction and 16 KB data caches, both of which seem small. The computer had 4 gigabytes of main memory (more than enough for my experiments to fit in main memory) and 5 gigabytes of swap space.

4.2 Running Time Measurement

I measured the running time using the `times()` system call with a resolution of 1/100 seconds. The running time is the sum of the user and system times returned by this call.

I report only the running time of the function that corresponds to each MCR algorithm. That is, the function takes in the component graph of $\mathcal{G}$ in memory and returns the MCR of $\mathcal{G}$. To give an idea about the total running time, note the following figures: the time to read the largest c-test and find its SCCs took less than 10 s, and the time to generate the largest r-test and find its SCCs took less than 30 s.

4.3 Test Suite

I used graphs obtained from circuit benchmarks (c-tests) and graphs generated randomly (r-tests). The r-tests were strongly connected, and in our study, they provided upper bounds on the performance of the MCR algorithms. I used the `drand48()` function as my (pseudo-)random number generator.

I generated the c-tests from the ISPD98 circuit benchmark suite [Alpert 1998]. This test suite is newer than the ACM/SIGDA test suite (which also contains the ISCAS85 and ISCAS89 benchmarks that we used in our previous study [Dasdan et al. 1998, 1999]). The ISPD98 test suite also contains the largest circuit benchmarks.[4] As detailed in Section 1.2, the largest circuit in this suite is almost two orders of magnitude larger than the largest benchmark in the ACM/SIGDA test suite.

Table II show the properties of the c-tests and r-tests. The rows marked 01-18 are c-tests (or c01–c18), where cN (N = 01−18) was obtained from the ibmN in the ISPD98 test suite (note that "cN corresponds to ibmN"). The rows marked 01–27 are the r-tests (or r01–r27). Note that r01–r18 are identical to c01–c18 in terms of the number of nodes (n), the number of arcs (m), and the average degree ($2m/n$) (note that "cN corresponds to rN"). I derived the average degree of r19–r27 from those of r01–r18 using linear regression analysis. The number of nodes for these tests ranged from 2^{18} to 2^{20}, going up by factors of $2^{0.25}$.

I generated the c-tests as follows (i.e., graphs out of circuits). Each test in the ISPD98 test suite contains nets connecting modules. Therefore, I mapped each

[4]Although the ISPD98 test suite was originally proposed for the circuit partitioning problem, there was no reason that could prevent me from using it for the MCR problem.

Table II. Properties of the c-Tests i01–i18 (from the ISPD98 Test Suite) and r-Tests r01–r27 (Random Graphs) in the Test Suite. The Average Degree is the Average Node Degree and is Equal to $2m/n$

Test Names	lg (#nodes) (lg(n))	#nodes (n)	#arcs (m)	Average Degree	Test Names	lg (#nodes) (lg(n))	#nodes (n)	#arcs (m)	Average Degree
c01, r01	13.64	12,752	36,681	5.75	c15, r15	17.30	161,570	529,562	6.56
c02, r02	14.26	19,601	61,829	6.31	c16, r16	17.49	183,484	589,253	6.42
c03, r03	14.50	23,136	66,429	5.74	c17, r17	17.50	185,495	671,174	7.24
c04, r04	14.75	27,507	74,138	5.39	c18, r18	17.68	210,613	618,020	5.87
c05, r05	14.84	29,347	98,793	6.73	r19	18.00	262,144	851,968	6.50
c06, r06	14.99	32,498	93,493	5.75	r20	18.25	311,744	1,013,166	6.50
c07, r07	15.49	45,926	127,774	5.56	r21	18.50	370,728	1,204,865	6.50
c08, r08	15.65	51,309	154,644	6.03	r22	18.75	440,879	1,432,834	6.50
c09, r09	15.70	53,395	161,430	6.05	r23	19.00	524,288	1,703,936	6.50
c10, r10	16.08	69,429	223,090	6.43	r24	19.25	623,487	2,026,333	6.50
c11, r11	16.11	70,558	199,694	5.66	r25	19.50	741,455	2,409,729	6.50
c12, r12	16.12	71,076	241,135	6.79	r26	19.75	881,744	2,865,667	6.50
c13, r13	16.36	84,199	257,788	6.12	r27	20.00	1,048,576	3,407,872	6.50
c14, r14	17.17	154,605	394,497	5.10					

module to a node, and each net between i sources and j sinks ($i + j$-terminal nets) to $i \cdot j$ arcs (2-terminal nets) such that each arc was from one source to one sink. The number of arcs in the c-tests was about 2.8 times the number of nets in the original circuits.

I generated the r-tests as follows. I repeatedly selected two nodes randomly and inserted an arc between them until the required number of arcs was reached. I disallowed self loops and multiple arcs between the same pair of nodes because they were useless, for example, the multiple arcs between a pair of nodes could safely be replaced by the one with the minimum cost. I also connected all the nodes in a circle to make the r-test strongly connected. This way, r01–r18 were made tougher for the MCR algorithms than c01–c18 even though they had the same number of nodes and arcs. The running time on an r-test is always an upper bound on that on the corresponding c-test due to the strong connectivity of the former.

4.4 Programming Details and Data Structures

I wrote the programs in C++. I implemented each MCR algorithm as a function that created and destroyed its own data structures (other than the graph data structure) and inlined its own functions. These ensured that all the MCR algorithms were compared uniformly and fairly.

I wrote two sets of programs: one set for the MCR problem and another set for the minimum cycle mean problem. The former set (respectively, the latter set) was used when $T > 1$ (respectively, $T = 1$). Although the former set of programs were capable of solving the minimum cycle mean problem too, my implementation provided the best implementation and fairer results for this problem.

The space complexity of each MCR algorithm was the same, linear in the size of the input graph. In Table III, we list the actual space complexity with constants. Note that to improve the running time, I sometimes traded off space.

Table III. Space Requirement and Data Structures for Each MCR Algorithm (Alg.)

Alg.	Data Type of $d(\cdot)$	Space Requirement		Extra Data Structures (Beyond the Graph Data Structure)
		Implemented	Minimum	
SZY	float	$6n$	$4n$	node array
TAR	float	$8n$	$7n$	node array, node circular queue, and node binary tree
VAL	float	$6n$	$3n$	node array
HOW	float	$6n$	$3n$	node array and node queue
KO	int	$7n + m$	$7n + m$	node array, arc heap, and node binary tree
YTO	int	$10n + m$	$9n + m$	node array, node heap, and node binary tree

Table III gives two more details: The first detail is the data type of the (node) distance $d(\cdot)$, "float" for floating point and "int" for integer. Since int arithmetic is faster than float arithmetic, KO and YTO had an advantage over the other MCR algorithms. They, however, lose this advantage once the tests get very large (larger than those in my test suite). In that case, they can use float arithmetic for manipulating the distances.

The second detail given in Table III is the extra data structures needed by each MCR algorithm (i.e., those beyond the graph data structure).

Each data structure as well as the graph data structure were implemented using arrays to improve cache locality, and created and destroyed once per run of the function that contained the MCR algorithm. A binary tree was "linearized" using its pre-order traversal [Mehlhorn and Naher 2000]. Because of the simplicity of implementing each of these data structures using arrays, I will skip explaining them; instead, I refer the reader to Cormen et al. [1991] or any good data structures book.

4.5 Experiments

Each of my experiments consisted of 30 runs on each graph (the c-tests and r-tests). According to Jain [1991], to estimate the mean $\bar{x}$ of the running times with an accuracy of 10% and a confidence level of 95%, the sample size should be equal to $(1.96 * s/\bar{x})^2$ where s is the sample standard deviation. The minimum sample size suggested is 30, which ensures that the ratio $s/\bar{x}$ is at least 28%. In my experiments, although this ratio was a lot smaller, I still used the minimum suggested value of 30 in order not to introduce any bias into my results due to the sample size. Each reported running time result is the average over these 30 runs. Before each run, I randomly and uniformly reassigned the arc costs (respectively, transit times) between 1 and W (respectively, T). The upper bound W had three values: 3, 30, and 300; and T had four values: 1, 3, 30, 300. The largest value 300 was set to keep the int data type in KO and YTO; using a larger value resulted in an int overflow. Note that in sorted order, two consecutive values of W and T differ by an order of magnitude.

I performed two sets of experiments: one set to determine the running times and another set to determine the counts of certain selected operations of each MCR algorithm. The latter was performed to eliminate the effect of the

computer setup as much as possible. I ensured that the time complexity of each MCR algorithm could be expressed in terms of these counts. This helped me estimate the practical time complexity of each MCR algorithm in terms of the properties of the input graphs. The use of similar counters was advocated by Ahuja et al. [1993], and was successfully employed in Ahuja et al. [1997].

My use of operation counters was very simple. I first determined which operations (e.g., loops) determined the time complexity of each algorithm. I then defined a counter for each of these operations such that each count corresponded to a constant-time operation like a simple addition. At the end of a run, the sum of the numbers returned by the counters gave the total operation count. Because of the way these operations were identified, the running time was expected to be proportional to the total operation count. This, in fact, was the case.

For the tests in my test suite, the ratio of the number of arcs to the number of nodes is at most 4. The circuits in the ISPD98 test suite show the same behavior. That is, for real circuits, $m = O(n)$. This fact allowed me to easily represent the practical time complexity of each MCR algorithm in terms of the number of nodes or the number of arcs, whichever was more convenient.

5. EXPERIMENTAL RESULTS

I will present the experimental results in terms of the following properties: generality, simplicity, space complexity, robustness, convergence, running times, and practical time complexity. These properties subsume all those outlined in Section 1.2. At the end of each section, I will present its summary as an "observation" for easy reference.

I used sound statistical techniques to analyze my experimental results. I assume that the reader is familiar with the following concepts: sampling distributions, large- and small-sample estimation, large- and small-sample tests of hypotheses, linear regression and correlation, and the analysis of variance and covariance. I will not explain these concepts in this article. Any good book on statistics and experimental analysis explains these concepts, for instance, see Jain [1991] and Mendenhall and Beaver [1994].

5.1 Generality, Simplicity, and Space Complexity

All the MCR algorithms are equally general. They can be used without any change to solve both of the cycle ratio problems as well as both of the cycle mean problems.

I evaluate the simplicity of a MCR algorithm through its space complexity, implementation effort, and the number and complexity of its data structures. Although the simplicity is a subjective measure, it can help in choosing an algorithm for a specific application.

According to Table III, VAL and HOW have the lowest space complexity. Among the data structures in Table III, binary trees and heaps are relatively harder to implement than the others (except the graph data structure). In terms of pseudocode, KO and YTO are more complex. When all of these are taken into account together with my own implementation effort, I have the following observation.

Observation 5.1. All the MCR algorithms are equally general. They all have linear space complexity. VAL and HOW seem to be the simplest MCR algorithms.

5.2 Robustness

The MCR computed by SZY or TAR is not exact because it depends on the user-defined error bound ε, that is, it can differ from the exact MCR by ε. The other algorithms return the exact MCR. SZY and TAR can compute an exact MCR if arc costs are integral and $\varepsilon < 1/(n^2T^2)$ [Lawler 1976]. As this bound is extremely small for my test suite and could have made these algorithms significantly slower, I did not use it; instead, I set $\varepsilon = 0.01$, which was more advantageous to SZY and TAR.

The other aspect of the robustness is due to the stability of the results that an algorithm demonstrates across the tests in the test suite. This kind of robustness can easily be concluded from a visual inspection of the running time results. As for the dependence of the performance on W and ε, it seems that these parameters have a large effect on SZY and TAR, a slight effect on VAL and HOW, and no effect on KO and YTO. As for the dependence of the performance on the graph type (c-tests vs. r-tests) or size, YTO seems to be a clear winner. KO, although similar to YTO, does not show the same behavior: it is generally slower than VAL and HOW on the c-tests.

Observation 5.2. YTO is the most robust MCR algorithm, and SZY followed by TAR are the least robust MCR algorithms.

5.3 Convergence

Each MCR algorithm in this article computes the MCR iteratively (in iterations or passes of their main loops). The first pass in each algorithm starts with an estimate on the MCR. For VAL and HOW (respectively, KO and YTO), the initial estimate is an upper (respectively, lower) bound on the MCR. For SZY and TAR, it can be both. In the remaining passes, the estimate gets closer and closer to the MCR, that is, it converges to the MCR. Figure 9 shows the convergence of the estimate for each algorithm. As this figure validates, for VAL and HOW (respectively, KO and YTO), the estimate monotonically decreases (respectively, increases), that is, it provides an upper bound (respectively, lower bound) on the MCR. For SZY and TAR, the convergence is not monotonic; the estimate is sometimes an upper bound and sometimes a lower bound on the MCR.

The number of passes needed for convergence is different in each algorithm, and ends when each algorithm verifies that the estimate is indeed the MCR. Hence, the number of passes can be thought of in two phases: “discovery phase” and “verification phase”. For SZY, TAR, KO, and YTO, these two phases are identical but SZY and TAR can discover a good approximation to the MCR more quickly than KO and YTO can. For VAL and HOW, these two phases are different and the discovery phase is considerably shorter than the verification phase: we observed that the former was approximately 1/3 of the latter. This implies some improvement opportunities: better techniques for verification to

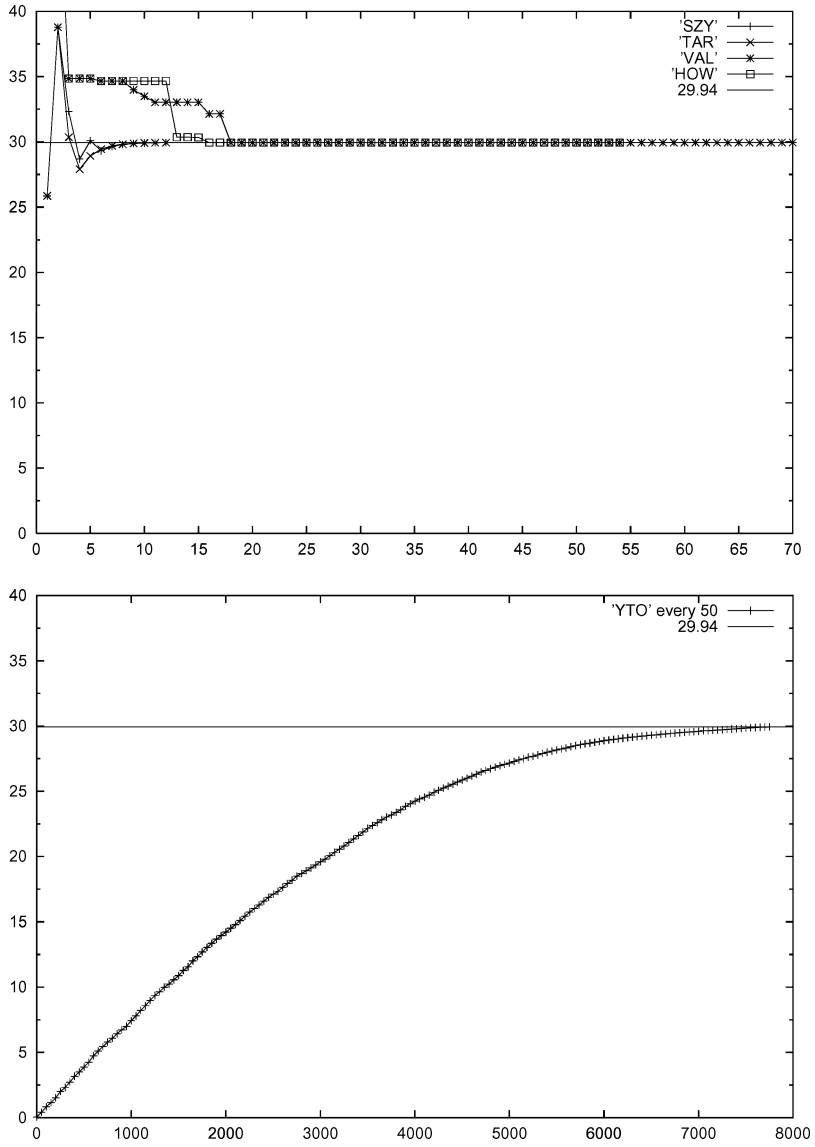

Fig. 9. The convergence of all the algorithms on one of the small r-tests. The y-axis is the range for the MCR (=29.94), and the x-axis is the pass number. The top plot is for SZY, TAR, VAL, and HOW, and the bottom plot is for YTO and KO. For the latter, every 50th data point was plotted to avoid clutter.

speed up these algorithms and the use of the discovery phase (or a number of its initial iterations) to provide better upper bounds for SZY and TAR. Recall that we used a single iteration of FIND-RATIO in a similar way. We finally note that KO and YTO can provide a better lower bound for SZY and TAR.

Observation 5.3. VAL and HOW (respectively, KO and YTO) converge to the MCR from above (respectively, below), and SZY and TAR converge within

a bounded interval that contains the MCR. VAL and HOW spend more time in verifying their result than in discovering it. For the other algorithms, the discovery of the result verifies the result.

5.4 Running Times

Due to the large number of running time results obtained, I had to be selective in my presentation of them. I present four representative plots for c-tests (respectively, r-tests) and a summary of the rest of the results. Two of them compare all the MCR algorithms for certain W and T. Figure 10 for c-tests and Figure 12 for r-tests display such plots. The other two plots show the behavior of SZY and TAR under different W and T (as the other algorithms do not show much dependence on these parameters). Figure 11 for c-tests and Figure 13 for r-tests display such plots.

The y-axis of each plot is the running time in seconds, and the x-axis is the number of nodes in the test (in logarithmic base 2 scale). The legend ALG-W-T for a curve in a plot means that the curve belongs to the algorithm ALG under the upper bounds W and T.

5.4.1 *Running Times on C-Tests.* The plots in Figure 10 show that when W is large, SZY and TAR usually perform significantly worse than the other algorithms do. The plots for 300-3 and 300-30 are almost identical to the plot 300-1 (in terms of the ranking of the algorithms). SZY and TAR perform better than the other algorithms when W-T is one of these: 3-1, 3-3, and 30-1, that is, especially when W is small as expected from their time complexity analyses.

The plots in Figure 11 validate the above conclusion. They also show that T plays some role. From these plots, it seems that the running times of SZY and TAR also increase as W/T increases. For instance, these algorithms are faster for 300-300 than for 300-3. TAR is even faster than KO for 300-300.

For the rest of the results on the c-tests, we note that the running times of HOW and KO were under 3 seconds, and those of VAL and YTO were under 1 second.

Observation 5.4. The overall result on the c-tests is that the algorithms can be ranked from the fastest to the slowest as follows: YTO, VAL, HOW, KO, TAR, and SZY. When W is very small or W/T is very small, TAR followed by SZY and VAL are the best choices.

5.4.2 *Running Times on r-tests.* The plots in Figure 12 clearly show how YTO followed by KO are faster than the other algorithms. The plots for all the other cases point to the same fact. Only for the plots for 3-1, SZY gets closer to KO, and TAR gets closer to YTO. That is, even for small W, YTO and KO are the winners. Since the r-tests provide upper bounds on the performance of these algorithms, this observation implies that for even larger tests, YTO and KO will dominate the other algorithms.

The plots in Figure 13 show that the performance of SZY and TAR with respect to W and T is similar to that in Figure 11. That is, these algorithms depend on W and T in the same way largely irrespective of the test type.

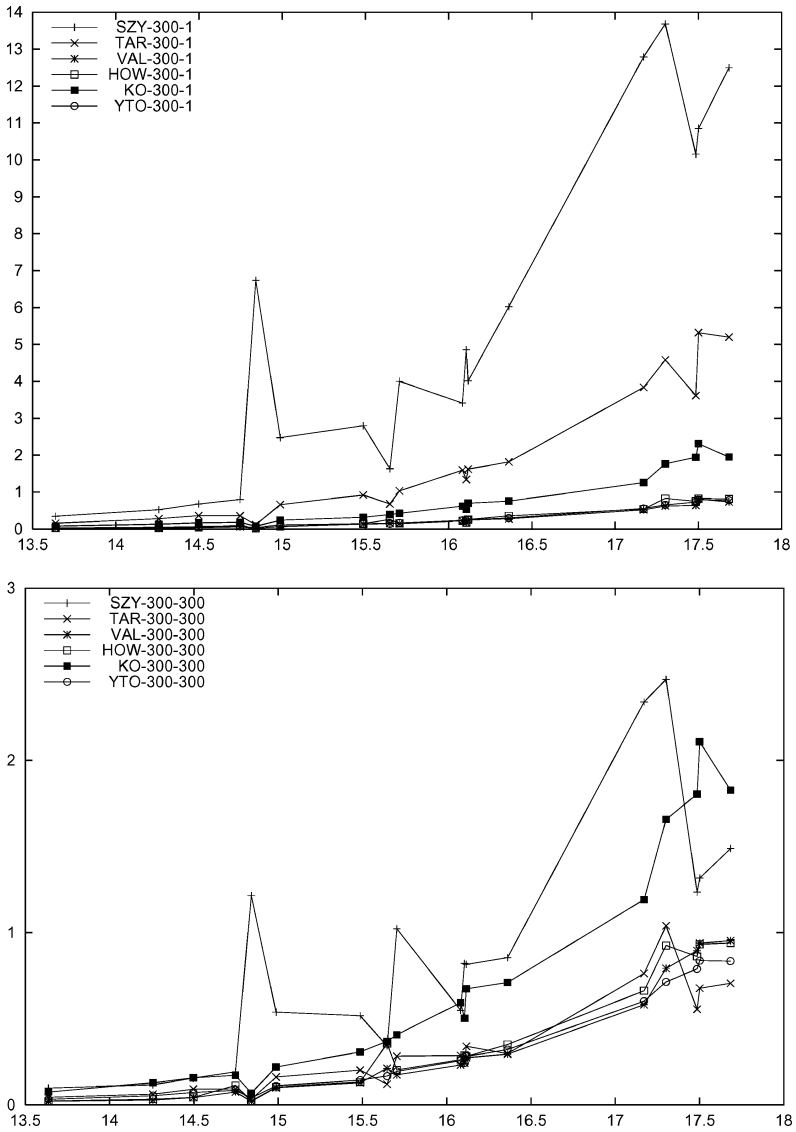

Fig. 10. The running times (the y-axis, in seconds) of all the algorithms with respect to the number of nodes (the x-axis, in logarithm base 2) of the c-tests for two W and T combinations. The legend is ALG-W-T where ALG is the algorithm. For the top plot, $W = 300$ and $T = 1$, and for the bottom one, $W = 300$ and $T = 300$.

For the rest of the results on the r-tests, we note that the running time of YTO was under 25 seconds, and that of KO was under 55 seconds. The upper bound on the running times of the other algorithms was over 200 seconds.

Observation 5.5. The overall result on the r-tests is that the algorithms can be ranked from the fastest to the slowest as follows: YTO, KO, HOW, VAL, TAR, and SZY. Even when W is very small, YTO followed by KO still leads the other algorithms.

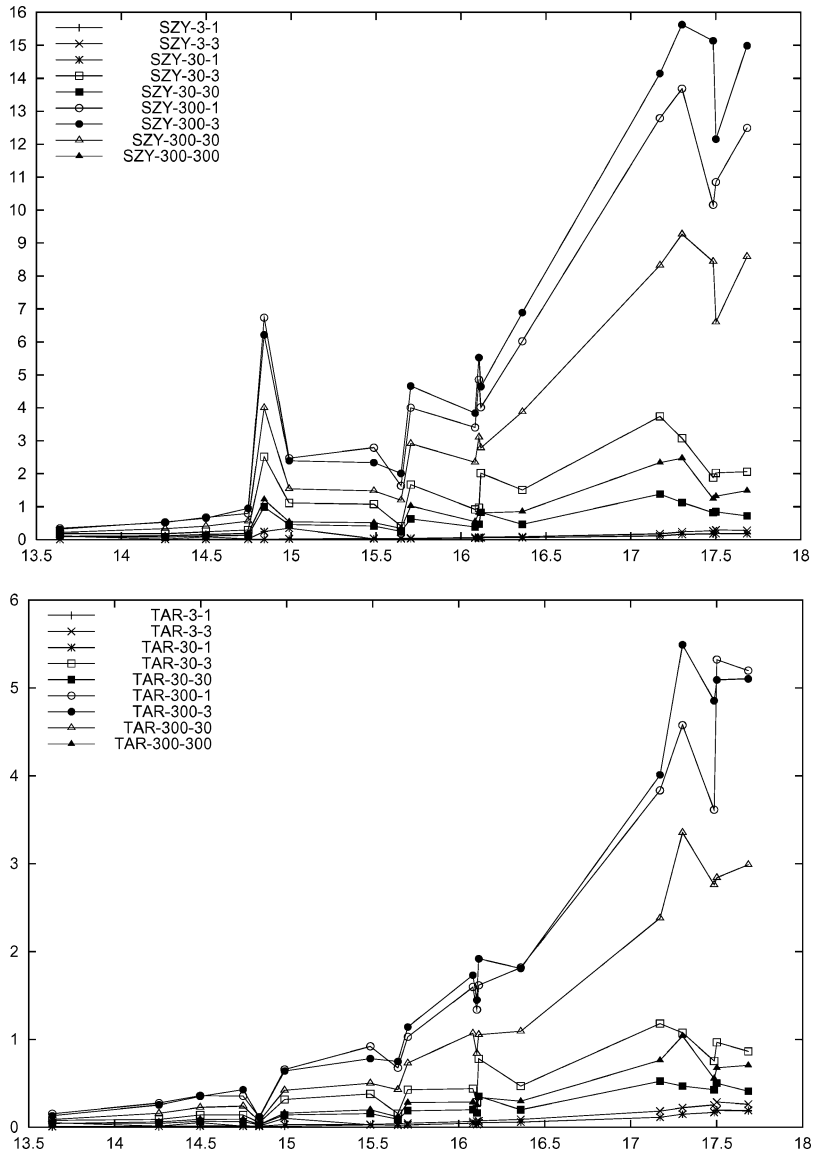

Fig. 11. The running times (the y-axis, in seconds) of SZY (the top plot) and TAR (the bottom plot) with respect to the number of nodes (the x-axis, in logarithm base 2) of the c-tests for all W and T combinations. The legend is ALG-W-T where ALG is the algorithm.

5.4.3 *Further Remarks on YTO's Superior Performance.* The major observation is the dominance of YTO over the other algorithms. In Dasdan et al. [1998, 1999], YTO was not faster than HOW. One of the main reasons for YTO's superior performance was due to how YTO was implemented. In Dasdan et al. [1998, 1999], the loops at line 22 and 29 in Figure 8 were iterating over all the arcs of $\mathcal{G}$, and the bodies of these loops were performed for those arcs that were either entering or leaving the subtree $T_p(v)$. For this article, I realized that this

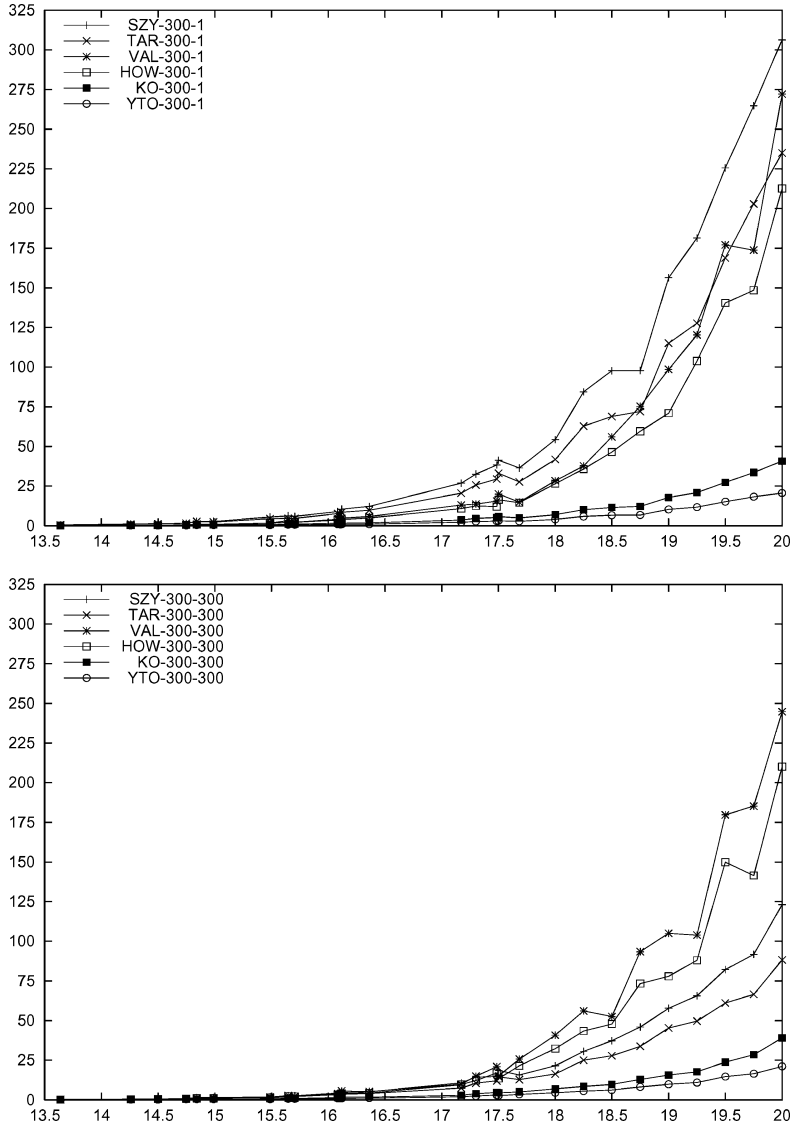

Fig. 12. The running times (the y-axis, in seconds) of all the algorithms with respect to the number of nodes (the x-axis, in logarithm base 2) of the r-tests for two W and T combinations. The legend is ALG-W-T where ALG is the algorithm. For the top plot, $W = 300$ and $T = 1$, and for the bottom one, $W = 300$ and $T = 300$.

subtree contained 2 or 3 nodes on the average and it was better to iterate over the nodes of this subtree, as in Figure 8. This change also applied to KO, and reduced their running times considerably.

5.5 Practical Time Complexity

By the practical time complexity of an algorithm, I refer to the *asymptotic time complexity* extrapolated from the running time that the algorithm exhibits on

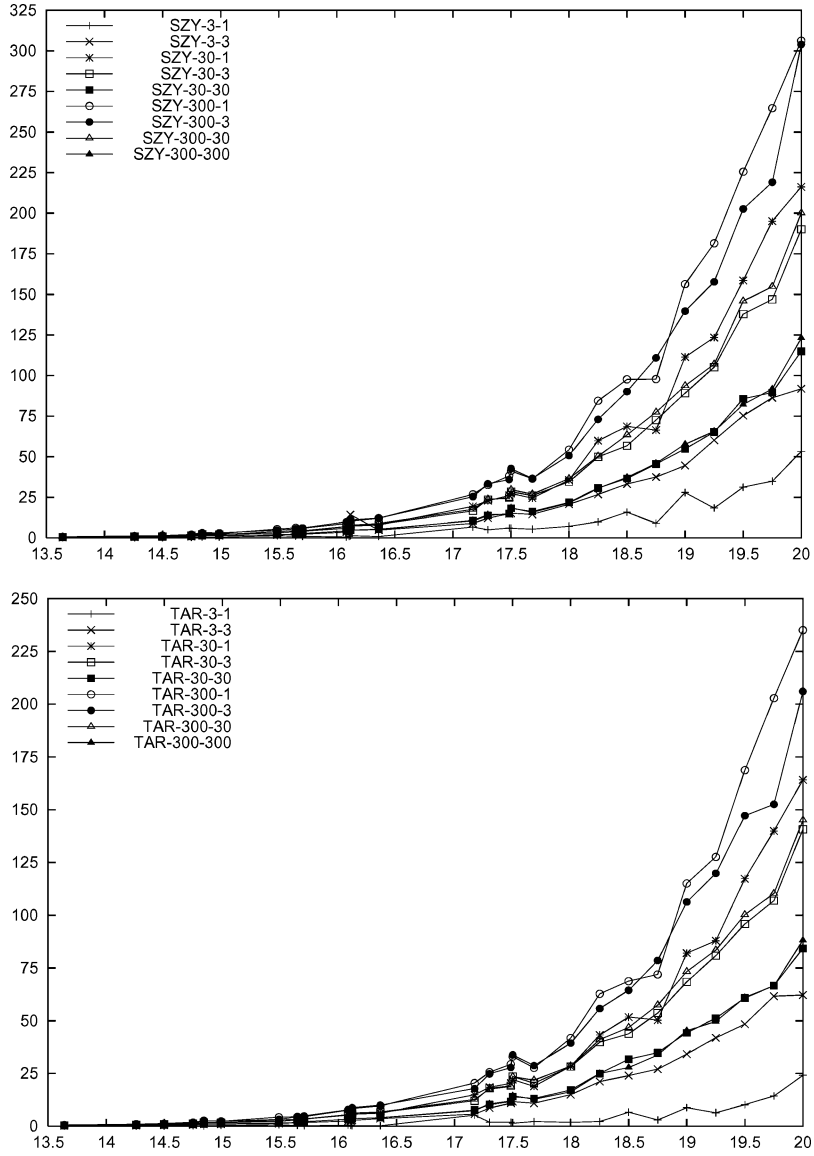

Fig. 13. The running times (the y-axis, in seconds) of SZY (the top plot) and TAR (the bottom plot) with respect to the number of nodes (the x-axis, in logarithm base 2) of the r-tests for all W and T combinations. The legend is ALG-W-T where ALG is the algorithm.

a set of tests (either c-tests or r-tests). To derive the practical time complexity of each MCR algorithm on, say, the c-tests, I first ran them on the test suite to generate sets of data: one set for their operation counts (called the counter data) and another set for their running times (called the runtime data).

Second, I expressed the practical time complexity of each algorithm in terms of a parameter α as in the following expressions: $O((nm)^{\alpha} \lg(nWT))$ for SZY and TAR, $O((nm)^{\alpha})$ for VAL and HOW, and $O((nm)^{\alpha} \lg(n))$ for KO and YTO. I derived

the form of each expression from the worst-case time complexity analysis of the corresponding algorithm. For these expressions, I also had to use another parameter for the constants hidden in the O notation. As these constants were dependent very much on my computer setup, I do not report them in this article (other than saying that they were around 10^{-8}). In the above expressions, I could have used n^α or m^α or (even $n^\alpha m^\gamma$ by adding another parameter γ) instead of $(nm)^\alpha$. I eliminated all of these choices because my test suite satisfied $m \leq 4n$ and all of these choices could easily be converted to each other with suitable exponents.

Third, I used regression analysis to fit these expressions to my two sets of data and obtained two sets of values for α (again, one set for the counter data and another set for the runtime data). For regression analysis, I used the standard method of least squares (LS).

Fourth, I computed 95% confidence intervals for each α derived for each algorithm. Since each test had 9 versions with different W and T values, I obtained 18 confidence intervals for each algorithm, 9 of them on the c-tests and 9 of them on the r-tests. To compute the final confidence interval for α of each algorithm, I took a conservative approach (inspired by Johnson et al. [1996]): I took the average of the 9 confidence intervals on each test group. To validate this approach, I also performed an analysis of variance (ANOVA) for the 9 results to see if they come from the same population, which provided an indirect way of ensuring that these versions did not have any variability due to W and T. I was able to validate this approach for all the algorithms (except for SZY and TAR) with a very high probability (close to 95%). In the case of SZY and TAR, it seemed that there was still some variability even though I had divided the results by the number of passes in each one to eliminate the $lg(nWT)$ factor from the results.

Fifth, I prepared the plots of the 95% confidence intervals and the coefficients of determination (the R^2 values). The result is in Figure 14. The x-axis is for the MCR algorithms. The y-axis is for the confidence intervals and the R^2 values. Each algorithm has two confidence intervals and two R^2 values: one computed from the counter data (marked with "counter" in the plots) and another from the running time data (marked with "running time" in the plots). The confidence intervals are the solid and vertical lines; the R^2 values are the points on the dashed and somewhat horizontal lines. The end points and the middle point of each confidence interval are also marked. Recall from statistics that an R^2 value is between 0 and 1, and a value close to 1 is preferable for the strength of the fit.

Sixth, I analyzed the plots. Figure 14 shows that for the c-tests, the confidence intervals are wide and the R^2 values are away from 1 whereas for the r-tests, the confidence intervals are very narrow and the R^2 values are very close to 1. Moreover, the confidence intervals are wider for the counter data than for the running time data although the confidence intervals from the counter data should provide the basis for the practical time complexity derivation as the operation counts are largely independent of the experimental environment (especially the computer setup). In short, these results show that the confidence intervals for the running times on the r-test are more reliable in providing the worst-case practical time complexity, and I used them for that purpose.

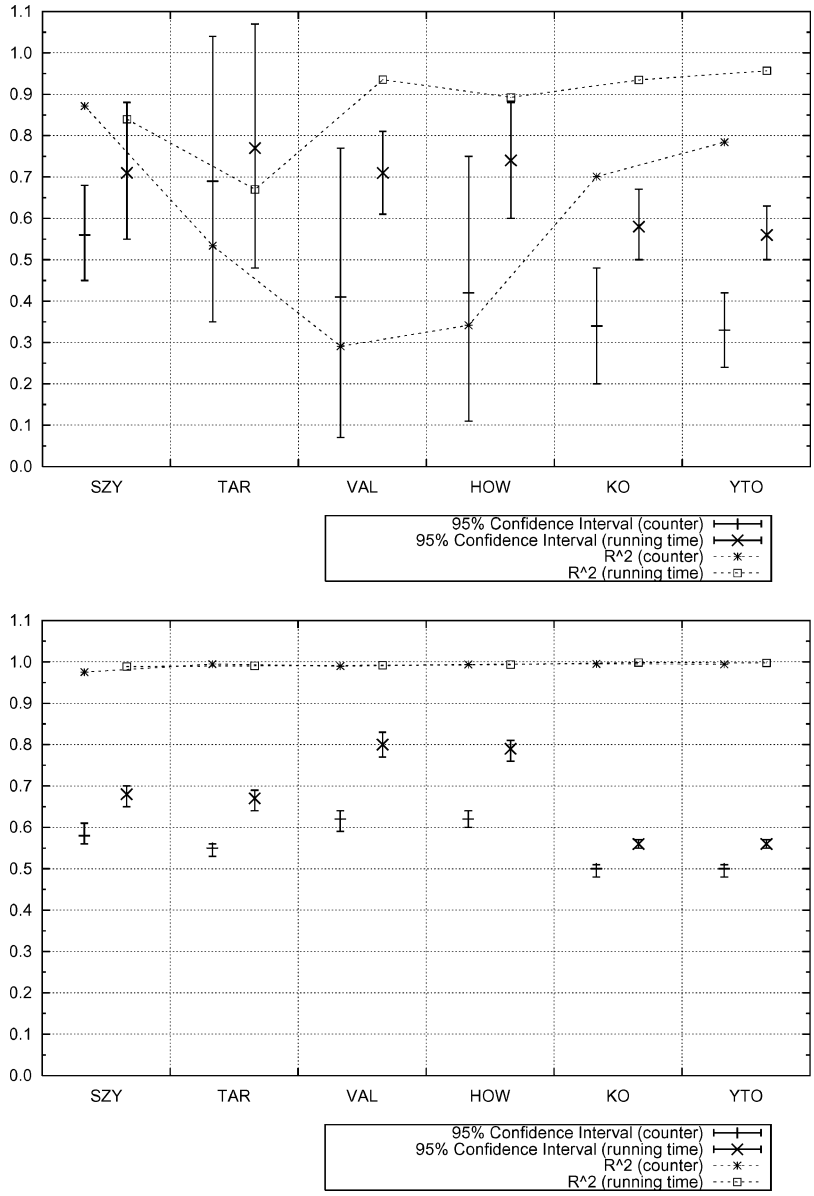

Fig. 14. The 95% confidence intervals (vertical solid lines and points) for α in the practical time complexity of all the algorithms: $O((nm)^{\alpha} \lg(nWT))$ for SZY and TAR, $O((nm)^{\alpha})$ for VAL and HOW, and $O((nm)^{\alpha} \lg(n))$ for KO and YTO. For each algorithm, there are two sets of confidence intervals and R^2 values: one set from the operation counts (marked as "counter") and another from the running times (marked as "running time"). Note that the y-axis subsumes both the confidence intervals and the R^2 values. The top plot is for the c-tests and the bottom one is for the r-tests.

Table IV. The Worst-Case Practical Time Complexity for Each Algorithm (Alg.)

Alg.	From Table I	Practical Time Comp. (on the c-Tests)	Practical Time Comp. (on the r-Tests)
SZY	$O(nm \lg(nWT))$	$(nm)^{0.71} \lg(nWT)$	$(nm)^{0.68} \lg(nWT)$
TAR	$O(nm \lg(nWT))$	$(nm)^{0.77} \lg(nWT)$	$(nm)^{0.67} \lg(nWT)$
VAL	$O(nm)$	$(nm)^{0.71}$	$(nm)^{0.81}$
HOW	$O(nm)$	$(nm)^{0.76}$	$(nm)^{0.81}$
KO	$O(nm \lg(n))$	$(nm)^{0.58} \lg(n)$	$(nm)^{0.58} \lg(n)$
YTO	$O(nm \lg(n))$	$(nm)^{0.56} \lg(n)$	$(nm)^{0.58} \lg(n)$

Finally, I verified the derived practical time complexity of each algorithm by comparing it with the actual running time of the algorithm. I wanted to ensure that the former provided an upper bound on the latter. The final results are in Table IV. They provide upper bounds on the asymptotic practical time complexity for each MCR algorithm. Note that although the worst-case practical time complexity of YTO and KO is about $O(n^{1.16} \lg n)$ (due to their running times), their practical time complexity from the counter data was $O(n \lg n)$ (as shown in Figure 14), and this is in agreement with the expected running time analysis done for YTO in Young et al. [1991] for random graphs.

6. CONCLUSIONS

Optimum cycle ratio and mean algorithms and are fundamental to the performance analysis of discrete-event systems (hence, digital systems) with cycles. They have important applications in the CAD field. In a recent published study, we have provided a comprehensive experimental analysis of all the known optimum cycle mean (OCM) algorithms. This study in this article builds upon the experience and knowledge gained from our previous study. In this article, I give a more focused and more comprehensive experimental analysis of the six fastest OCR and OCM algorithms. The test suite consisted of the largest circuit benchmarks and even larger random graphs. The latter were used to provide upper bounds on the running times. The algorithms were evaluated in terms of the following properties: operation counts, running times, convergence behavior, space requirement, generality, simplicity, and robustness. The experimental results were examined using various statistical techniques (including regression analysis). Asymptotic time complexity expressions for each algorithm were provided. This article also provides clear guidance to the use and implementation of the algorithms. When all the properties are considered, the fastest algorithm was found to be YTO (although its running time was on par with that of VAL and HOW, which was reported as the fastest algorithm in our previous study.)

ACKNOWLEDGMENTS

I greatly appreciate the contributions of Prof. Rajesh Gupta of the University of California at San Diego and Prof. Sandy Irani of the University of California, Irvine to our previous study, which carried the seeds of some of

the ideas in this article. I thank Weimin Chen, Hyun Dasdan, Jamil Kawa, Yehia Massoud, and Preeti Panda for their reviews and insightful comments. I also thank the anonymous reviewers for their comments which helped improve the presentation.

REFERENCES

AHUJA, R. K., KODIALAM, M., MISHRA, A. K., AND ORLIN, J. B. 1997. Computational investigations of maximum flow algorithms. *Europ. J. Oper. Res. 97*, 509–542.

AHUJA, R. K., MAGNANTI, T. L., AND ORLIN, J. B. 1993. *Network Flows*. Prentice Hall, Upper Saddle River, N.J.

ALBRECHT, C., KORTE, B., SCHIETKE, J., AND VYGEN, J. 1999. Cycle time and slack optimization for VLSI chips. In *Proceedings of the International Conference on Computer-Aided Design*. IEEE Computer Society Press, Los Alamitos, Calif., 232–238.

ALPERT, C. J. 1998. The ISPD98 circuit benchmark suite. In *Proceedings of the International Symposium on Physical Design*. ACM/IEEE, New York, 588–593.

BACELLI, F., COHEN, G., OLSDER, G. J., AND QUADRAT, J.-P. 1992. *Synchronization and Linearity*. Wiley, New York.

BURNS, S. M. 1991. Performance analysis and optimization of asynchronous circuits. Ph.D. dissertation, California Institute of Technology.

CARLONI, L. P. AND SANGIOVANNI-VINCENTELLI, A. L. 2000. Performance analysis and optimization of latency insensitive systems. In *Proceedings of the 37th Design Automation Conference*. IEEE Computer Society Press, Los Alamitos, Calif., 361–367.

CHANDRACHOODAN, N., BHATTACHARYYA, S. S., AND LIU, K. R. 1999. Negative cycle detection in dynamic graphs. Tech. Rep. UMIACS-TR-99-59, Institute for Advanced Computer Studies, Univ. of Maryland at College Park, College Park, Md., Sept.

CHERKASSKY, B. V. AND GOLDBERG, A. V. 1996. Negative-cycle detection algorithms. In *Proceedings of the 4th European Symposium on Algorithms*. 349–363.

COCHET-TERRASSON, J., COHEN, G., GAUBERT, S., MCGETTRICK, M., AND QUADRAT, J.-P. 1998. Numerical computation of spectral elements in max-plus algebra. In *Proceedings of the IFAC Conference on System Structure and Control*.

CORMEN, T. H., LEISERSON, C. E., AND RIVEST, R. L. 1991. *Introduction to Algorithms*. MIT Press, Cambridge, Mass.

CUNINGHAME-GREEN, R. A. AND YIXUN, L. 1996. Maximum cycle-means of weighted digraphs. *Applied Math.-JCU 11*, 225–234.

DANTZIG, B., BLATTNER, W., AND RAO, M. R. 1967. Finding a cycle in a graph with minimum cost to times ratio with applications to a ship routing problem. In *Theory of Graphs*, P. Rosenstiehl, Ed. Dunod, Paris and Gordon and Breach, New York, 77–84.

DASDAN, A. 2002. A strongly polynomial algorithm for over-constraint resolution. In *Proceedings of the 10th International Symposium on HW/SW Codesign (CODES)*. 127–132.

DASDAN, A. AND GUPTA, R. K. 1998. Faster maximum and minimum mean cycle algorithms for system performance analysis. *IEEE Trans. Computer-Aided Design 17*, 10 (Oct.), 889–99.

DASDAN, A., IRANI, S., AND GUPTA, R. K. 1998. An experimental study of minimum mean cycle algorithms. Tech. Rep. #98-32, University of California, Irvine, Calif., July.

DASDAN, A., IRANI, S., AND GUPTA, R. K. 1999. Efficient algorithms for optimum cycle mean and optimum cost to time ratio problems. In *Proceedings of the 36th Design Automation Conference.*

GOLDBERG, A. V. 2001. A practical shortest path algorithm with linear expected time. *Submitted to SIAM J. Comput.*, http://www.avglab.com/andrew/.

GONDRAN, M. AND MINOUX, M. 1984. *Graphs and Algorithms*. Wiley, New York.

HARTMANN, M. AND ORLIN, J. B. 1993. Finding minimum cost to time ratio cycles with small integral transit times. *Networks 23*, 567–574.

HOWARD, R. A. 1960. *Dynamic Programming and Markov Processes*. The M.I.T. Press, Cambridge, Mass.

HULGAARD, H., BURNS, S. M., AMON, T., AND BORRIELLO, G. 1995. An algorithm for exact bounds on the time separation of events in concurrent systems. *IEEE Trans. Comput. 44*, 11 (Nov.), 1306–1317.

ITO, K. AND PARHI, K. K. 1995. Determining the minimum iteration period of an algorithm. *J. VLSI Signal Proce. 11*, 3 (Dec.), 229–44.

JAIN, R. 1991. *The Art of Computer Systems Peformance Analysis*. Wiley-Interscience, New York.

JOHNSON, D. S., MCGEOCH, L. A., AND ROTHBERG, E. E. 1996. Asymptotic experimental analysis for the held-karp traveling salesman bound. In *Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms*. ACM, New York, 341–350.

KARP, R. M. 1978. A characterization of the minimum cycle mean in a digraph. *Disc. Math. 23*, 309–311.

KARP, R. M. AND ORLIN, J. B. 1981. Parametric shortest path algorithms with an application to cyclic staffing. *Disc. Appl. Math. 3*, 37–45.

LAWLER, E. L. 1976. *Combinatorial Optimization: Networks and Matroids*. Holt, Reinhart, and Winston, New York.

MADANI, O. 2000. Complexity results for infinite-horizon Markov decision processes. Ph.D. dissertation, University of Washington.

MATHUR, A., DASDAN, A., AND GUPTA, R. K. 1998. Rate analysis of embedded systems. *ACM Trans. Design Automat. Elect. Syst. 3*, 3 (July), 408–436.

MEHLHORN, K. AND NAHER, S. 1995. LEDA: A platform for combinatorial and geometric computing. *Commun. ACM 38*, 1, 96–102.

MEHLHORN, K. AND NAHER, S. 2000. *LEDA: A Platform for Combinatorial and Geometric Computing*. The Cambridge University Press, New York.

MENDENHALL, W. AND BEAVER, R. J. 1994. *Introduction to Probability and Statistics*. Duxbury Press, Belmont, Calif.

NIELSEN, C. D. AND KISHINEVSKY, M. 1994. Peformance analysis based on timing simulation. In *Proceedings of the 31st Design Automation Conference*. IEEE, Computer Society Press, Los Alamitos, Calif., 70–76.

ORLIN, J. B. AND AHUJA, R. K. 1992. New scaling algorithms for the assignment and minimum mean cycle problems. *Math. Prog. 54*, 41–56.

RADZIK, T. AND GOLDBERG, A. V. 1994. Tight bounds on the number of minimum-mean cycle cancellations and related results. *Algorithmica 11*, 226–242.

RAMAMOORTHY, C. V. AND HO, G. S. 1980. Performance evaluation of asynchronous concurrent systems using petri nets. *IEEE Trans. Software Eng. 6*, 5 (Sept.), 440–9.

SHENOY, N. AND RUDELL, R. 1994. Efficient implementation of retiming. In *Proceedings of the 31st Design Automation Conference*. IEEE, Computer Society Press, Los Alamitos, Calif., 226–33.

SKOROBOHATYJ, G. 1993. Code for finding a minimum mean cycle in a directed graph. WWW document, Konrad-Zuse-Zentrum fur Informationstechnik Berlin, (http://elib.zib.de/pub/Packages/mathprog/netopt/mmc/).

SZYMANSKI, T. G. 1992. Computing optimal clock schedules. In *Proceedings of the 29th Design Automation Conference*. ACM, New York, 399–404.

TARJAN, R. E. 1981. Shortest paths. Tech. Rep., AT&T Bell Laboratories, Murray Hill, N.J.

TARJAN, R. E. 1983. *Data Structures and Network Algorithms*. Society for Industrial and Applied Math., Philadelphia, Pa.

TEICH, J., SRIRAM, S., THIELE, L., AND MARTIN, M. 1997. Performance analysis and optimization of mixed asynchronous synchronous systems. *IEEE Trans. Computer-Aided Design 16*, 5 (May), 473–484.

YOUNG, N. E., TARJAN, R. E., AND ORLIN, J. B. 1991. Faster parametric shortest path and minimum-balance algorithms. *Networks 21*, 205–221.

Received June 2002; revised April 2003 and November 2003; accepted November 2003