An Experimental Study of Minimum Mean Cycle Algorithms

UCI-ICS Technical Report # 98-32

Ali Dasdan
Department of Computer Science
University of Illinois at Urbana-Champaign
1304 W. Springfield Ave., Urbana, IL 61801
E-mail: dasdan@ics.uci.edu

Sandra S. Irani (Corresponding author) and **Rajesh K. Gupta**
Department of Information and Computer Science
444 Computer Science Building
University of California, Irvine, CA 92697
E-mails: irani@ics.uci.edu and rgupta@ics.uci.edu

Abstract

We present a comprehensive experimental study of ten leading algorithms for the minimum mean cycle problem. For most of these algorithms, there has not been a clear understanding of their performance in practice although theoretical bounds have been proved for their running times. Only an experimental study can shed light on whether changes in an algorithm that make its running time theoretically more efficient are worth the overhead in terms of their payoff in practice. To this end, our experimental study provides a great deal of insight. In our evaluation, we programmed these algorithms uniformly and efficiently. We systematically compared them on a test suite composed of random graphs as well as benchmark circuits. Above all, our experimental results provide important insights into the individual performance as well as relative performance of these algorithms in practice. One of the most surprising results of this study is that Howard's algorithm, a well known algorithm to the stochastic control community but a relatively unknown algorithm to the graph theory community, is by far the fastest algorithm on our test suite although the only known bound on its running time is exponential. We also present two new, stronger bounds on its running time.

1. Introduction

1.1. Notation and definitions.

Consider a digraph $G = (V, E)$ with n nodes and m arcs. Associate with each arc e in E two numbers: a cost (or weight) $w(e)$ and a transit time $t(e)$. The weight and transit time of a path in G is equal to the sum of the weights and transit times of the arcs on the path, respectively. Let $w(C)$ and $t(C)$ denote the weight and the transit time of a cycle C in G.

The *cost to time ratio* $\rho(C)$ of cycle C is defined as

$$\rho(C) = \frac{w(C)}{t(C)}, \ \ t(C) > 0, \tag{1}$$

which basically gives the average cost per transit time for the cycle. The *minimum cost to time ratio* ρ^* of G is defined as $\rho^* = \min_C\{\rho(C)\}$ where C ranges over all cycles in G. The problem of finding ρ^* is called the *minimum cost to time ratio problem* (MCTRP).

Suppose now we simplify this problem by setting the transit time of every arc to unity. Then, the cost to time ratio of cycle C is referred to as its *(cycle) mean* and is denoted by $\lambda(C)$. More precisely,

$$\lambda(C) = \frac{w(C)}{t(C)} = \frac{w(C)}{|C|}, \tag{2}$$

where $|C|$ is the length of C. Notice that the mean of a cycle gives its average arc cost. The *minimum cycle mean* λ^* of G is defined as $\lambda^* = \min_C\{\lambda(C)\}$ where C ranges over all cycles in G. The problem of finding λ^* is called the *minimum mean cycle problem* (MMCP). The definitions of the maximum versions of both of these problems are analogous.

1.2. Applications.

The applications of both MCTRP and MMCP are important and numerous. See [11] for the applications in graph theory. We will mostly focus on their applications in the performance analysis: these problems have fundamental importance to the performance analysis of discrete event systems [3], which are general enough to model the manufacturing systems [3] and digital systems such as synchronous [20], asynchronous [4], data flow [12], and embedded real-time [16] systems. Simply put, the algorithms for these problems are essential tools to find the cycle period of a given cyclic discrete event system. Once determined, the cycle period is used to describe the behavior of the system analytically over an infinite time period.

1.3. Related work.

There are many algorithms proposed for both MCTRP and MMCP. We give a comprehensive classification of the fastest and the most common ones in Table I. References to a few more algorithms can be found in [11]. Note that as MMCP is a special case of MCTRP, any algorithm for the latter problem can be used to solve the former problem. Conversely, it is also possible to solve MCTRP using an algorithm for MMCP [10].

In Table I, the polynomial and pseudopolynomial algorithms are respectively ordered according to their worst-case running times. Those with the same running time are presented in alphabetical order of their inventors' names. Some references are cited more than once because they contain more than

TABLE I

MINIMUM MEAN CYCLE AND MINIMUM COST TO TIME RATIO ALGORITHMS FOR A GRAPH WITH n NODES AND m ARCS. (W, THE MAX ARC WEIGHT; T, THE TOTAL TRANSIT TIME OF THE GRAPH.)

	Minimum mean cycle algorithms					
	Name	Source	Year	Running time	Result	Complexity
1	DG	Dasdan & Gupta [8]	1997	$O(nm)$	Exact	Polynomial
2	HO	Hartmann & Orlin [11]	1993	$O(nm)$	Exact	Polynomial
3	Karp's	Karp [13]	1978	$\Theta(nm)$	Exact	Polynomial
4		Hartmann & Orlin [11]	1993	$O(nm + n^2 \lg n)$	Exact	Polynomial
5	YTO	Young, Tarjan, & Orlin [22]	1991	$O(nm + n^2 \lg n)$	Exact	Polynomial
6		Karp & Orlin [14]	1981	$\Theta(n^3)$	Exact	Polynomial
7	KO	Karp & Orlin [14]	1981	$O(nm \lg n)$	Exact	Polynomial
8	OA1	Orlin & Ahuja [19]	1992	$O(\sqrt{n}m \lg(nW))$	Approximate	Pseudopoly.
9	OA2	Orlin & Ahuja [19]	1992	$O(\sqrt{n}m \lg^2(nW))$	Approximate	Pseudopoly.
10		Cuninghame-Green & Yixun [7]	1996	$O(n^4)$	Exact	Polynomial
	Minimum cost to time ratio algorithms					
	Name	Source	Year	Running time	Result	Complexity
11	Burns'	Burns [4]	1991	$O(n^2 m)$	Exact	Polynomial
12		Megiddo [17]	1979	$O(n^2 m \lg n)$	Exact	Polynomial
13		Hartmann & Orlin [11]	1993	$O(Tm)$	Exact	Pseudopoly.
14	Lawler's	Lawler [15]	1976	$O(nm \lg(nW))$	Approximate	Pseudopoly.
15		Ito & Parhi [12]	1995	$O(Tm + T^3)$	Exact	Pseudopoly.
16		Gerez et al. [9]	1992	$O(Tm + T^3 \lg T)$	Approximate	Pseudopoly.
17		Gerez et al. [9]	1992	$O(Tm + T^4)$	Exact	Pseudopoly.
18	Howard's	Cochet-Terrasson et al. [6]	1997	$O(Nm)$	Exact	Pseudopoly.

one algorithm. The notation W is the largest integer arc weight, T is the sum of the transit times in the input graph, which is assumed to be an integer for the algorithms in rows 13 and 15, and N is a bound on the number of iterations that Howard's algorithm does. The previously known bound on N is the product of the out-degrees of all the nodes [6]. In this paper, we give two improved bounds.

Let ϵ denote the amount of error that we want to tolerate when doing a comparison of two numbers. Lawler's algorithm converges when the interval that contains the exact ρ^* is smaller than ϵ, which is why its result is approximate. The algorithms in rows 8, 9, and 16 can produce approximate results because they use the same convergence criterion as that of Lawler's algorithm. We also note that Burns' algorithm, the algorithm in row 6, and Howard's algorithm produce exact results but they may have precision problems. For example, Burns' algorithm needs to perform many equality tests between two floating point numbers, and such tests are controlled by ϵ.

1.4. Scope, motivations, and contributions.

This paper reports the results of our experimental study of ten leading minimum mean cycle algorithms and the minimum mean cycle versions of the minimum cost to time ratio algorithms from Table I, all of which are named in the table. The remaining algorithms in this table are not included in our study because they are very similar to the chosen ones.

We implemented each algorithm in a uniform and efficient manner. We tested them on a series of random graphs, obtained using one generator from [5], and real benchmark circuits, obtained from logic synthesis benchmarks. The running time as well as representative operation counts, as advocated in [2], are measured and compared.

The main contribution of this paper is a comprehensive empirical study comparing the ten leading algorithms for the minimum mean cycle problem. For many of these algorithms, there has not been

a clear understanding of their performance in practice, although theoretical bounds have been proven for their running times. Only an empirical analysis can shed light on whether changes in the algorithm which make the running time theoretically more efficient are worth the overhead in terms of their payoff in practice. To the best of our knowledge, this is the first study that systematically compares their performance as well as that brings a comprehensive study of these algorithms to the attention of the diverse group of research communities, e.g., graph theory, discrete event systems, computer-aided design of digital systems, to which this problem is very important.

In the process of this study, we have gained a great deal of insight into the behavior of these algorithms and have found effective implementational improvements. One of the most surprising results of this study is that Howard's algorithm, an unknown algorithm to the graph theory community, was by far the fastest on the graphs tested in this study. There is no polynomial time bound on the running time of this algorithm although it has been proven that the algorithm terminates with a worst-case bound which is the product of the out-degrees of all the nodes [6]. We present two alternative time bounds which, although not polynomial, give stronger bounds on most graphs.

1.5. Paper organization.

§ 2 presents a brief description of each algorithm with our improvements and our bounds on the running time of Howard's algorithm. § 3 describes our experimental framework. § 4 presents our experimental results and observations. Finally, § 5 gives our conclusions. We also have an appendix that contains the tables of the experimental results. Although we summarize most of the experimental results before the appendix, this appendix is nevertheless included to present the actual numbers.

2. Algorithm Descriptions

2.1. Definitions.

The minimum cycle mean λ^* of a graph $G = (V, E)$ can be defined as the optimum value of λ in the following linear program:

$$\max \ \lambda \ \text{ s.t. } \ d(v) - d(u) \leq w(u, v) - \lambda, \quad \forall (u, v) \in E, \tag{3}$$

where $d(v)$ is called the *distance* (or the node potential) of v. When the inequalities are all satisfied, $d(v)$ is equal to the weight of the shortest path from s to v in G when λ^* is subtracted from every arc weight. The node s is arbitrarily chosen as the *source* in advance. Let G_λ denote the graph obtained from G by subtracting λ from its every arc weight. The minimum cycle mean λ^* is the largest value of λ for which G_λ has no negative cycles.

We say that an arc $(u, v) \in E$ is *critical* if $d(v) - d(u) = w(u, v) - \lambda$, which we refer to as the *criticality criterion*. We say that a node is *critical* if it is adjacent to a critical arc, and that a graph is *critical* if its every arc is critical. The critical subgraph of G_{λ^*} contains all the minimum mean cycles of G. This is implied by the above linear program. Note that if the critical subgraph of G_λ is acyclic, then it is the shortest path tree of G_λ.

Henceforth, we assume that the input to the algorithm, $G = (V, E)$, is cyclic and strongly connected. This assumption not only simplifies most of the algorithms and their coding but also generally improves

their running times in practice. If G is not strongly connected, it can be partitioned into strongly connected components. Its minimum cycle mean is the minimum of those of its strongly connected components. This is the way we implemented all of the algorithms.

2.2. Burns' algorithm.

Burns' algorithm [4] is given in Figure 2. This algorithm is actually the minimum mean cycle version of the original Burns' algorithm, which can also solve the minimum cost to time ratio problem. The algorithm in [7] is almost identical to this algorithm. The only "major" difference is that in the former algorithm, the node heights are positive whereas in Burns' algorithm, they are negative (lines 14-19). Note that the authors of [7] give the running time of their algorithm as $O(n^4)$, which agrees with the $O(n^2 m)$ running time of Burns' algorithm.

Burns' algorithm is based on linear programming. It is an iterative algorithm constructed by applying the primal-dual method. It solves the above linear program (Equation 3) and its dual simultaneously. In essence, the behavior of Burns' algorithm is very similar to that of the parametric shortest path algorithms such as the KO algorithm: The KO algorithm improves upon an initial acyclic critical subgraph of G until the critical subgraph becomes cyclic, at which point the minimum cycle mean is found. Burns' algorithm also operates on the critical subgraph and terminates when it becomes cyclic but every iteration, it reconstructs the critical subgraph from scratch.

Burns' algorithm produces exact results but the equality check in line 9 can create some precision problems because the variables involved in this check are floating point numbers. This equality check is used to determine if an arc is critical. In our implementation, we changed this check so that an arc $e = (u, v)$ is critical when $(d(u) - d(v)) - (w(u, v) - \lambda) < \epsilon$, where ϵ is a very small, user-defined constant.

In our implementation, lines 14-19 are performed during the topological sorting of the critical graph in line 11. Moreover, as a byproduct of the topological sorting, we know whether or not the critical graph is cyclic.

2.3. Karp's algorithm and its variants.

Define $D_k(v)$ to be the weight of the shortest path of length k from s, the source, to v; if no such path exists, then $D_k(v) = +\infty$. Karp's algorithm [13], which is given in Figure 3, is based on his observation that

$$\lambda^* = \min_{v \in V} \max_{0 \leq k \leq n-1} \frac{D_n(v) - D_k(v)}{n - k}, \tag{4}$$

which is called Karp's theorem. Karp's algorithm computes the quantities $D_k(v)$ by the recurrence $D_k(v) = \min_{(u,v) \in E}\{D_{k-1}(u) + w(u, v)\}$, $k = 1, 2, \ldots, n$, with the initial conditions that $D_0(s) = 0$ and $D_0(v) = +\infty$, $v \neq s$. As observed in [8], [11], [22], this recurrence, which is not implemented recursively, makes the best and worst cases of Karp's algorithm the same, which is why it runs in $\Theta(nm)$.

We have three variants of Karp's algorithm: the DG algorithm, the HO algorithm, and the Karp2 algorithm. To summarize, the first algorithm improves Karp's algorithm by processing the nodes and arcs of the input graph more efficiently. The second algorithm checks to see if any of the cycles found before and during certain iterations in Karp's algorithm are critical. If so, it terminates. The

third algorithm improves the space complexity of Karp's algorithm. The remainder of this subsection describes these three variants in more detail.

The DG algorithm [8] improves upon Karp's algorithm by eliminating unnecessary work introduced by the above recurrence. It is given in Figure 4. It works in a breadth-first manner in that starting from the source, it visits the successors of nodes rather than their predecessors, as done in the recurrence. This process creates an unfolding of G, and when the algorithm is implemented using linked lists, its running time becomes equal to the size of the "unfolded" graph. Depending on the structure of G, the running time ranges from $\Theta(m)$ to $O(mn)$.

The HO algorithm [11] also improves upon Karp's algorithm. It is given in Figure 5. It helps to terminate Karp's algorithm early without changing its structure, i.e., it still uses the above recurrence. It is based on the following observation: Suppose $D_0(v), \ldots, D_k(v)$ are computed for all $v \in V$ and for some $k < n$. At this point, many of these paths will contain cycles. If one of these cycles, say C, is critical in G_λ when $\lambda = \lambda(C)$, then the minimum cycle mean is found and is equal to $\lambda(C)$. Note that $d(v) = \min_{0 \leq k \leq n-1}\{D_k(v) - k\lambda\}$, which is needed to check the criticality of C. We check for early termination when k becomes a power of 2, as suggested in [11]. If the early termination is not possible, this algorithm can add an overhead of $O(n^2 + m \lg n)$ in total to the running time of Karp's algorithm although it does not change the running time asymptotically.

In our implementation of the HO algorithm, we changed line 33 in Figure 5 to a comparison with ϵ so that an arc $e = (u, v)$ is critical when $(d(u) - d(v)) - (w(u, v) - \lambda) < \epsilon$. The reason for this change is exactly the same as the similar change for Burns' algorithm.

The Karp2 algorithm is a space efficient version of Karp's algorithm[1]. Karp's algorithm takes up $\Theta(n^2)$ space in order to store the D-values. The Karp2 algorithm reduces this space requirement to linear in the number of nodes n. The Karp2 algorithm performs two passes. In the first pass, it computes $D_n(v)$ for each node v without storing $D_k(v)$ for $k < n$. In the second pass, it computes the fraction in Karp's theorem as it computes each $D_k(v)$, $k < n$. The DG and HO algorithms also suffer from this large space complexity problem. Fortunately, the technique used in the Karp2 algorithm is also applicable to them.

2.4. Parametric shortest path algorithms.

The KO algorithm [14] and the YTO algorithm [22] are in this category. They are given in Figure 6 and Figure 7, respectively. The YTO algorithm is essentially an efficient implementation of the KO algorithm. These algorithms are based on the observation that the minimum cycle mean λ^* is the largest λ such that G_λ does not have any negative cycles. Thus, these algorithms start with a $\lambda = -\infty$ and always maintain a tree of shortest paths to a source node s. These algorithms change λ incrementally so that the shortest path tree changes by one arc in each iteration. When a cycle of weight zero is detected in G_λ, that cycle is the cycle with the minimum mean.

In our implementation, we slightly improved the KO algorithm. First, we obtain the initial tree by creating arcs with zero weight from the source s, a node not in the input graph, to all the other nodes, as suggested in [22]. These arcs exist in the shortest path tree only; We do not insert these arcs to the graph. In [14], the use of a shortest path algorithm is recommended to find this initial tree but our

[1] Suggested by S. Gaubert of INRIA, France.

implementation reduces the time to find it to $\Theta(n)$. Note that the shortest path tree for λ is a critical subgraph of G_λ. Second, line 26 in Figure 6 ensures that an update for an arc is performed only when *exactly* one of its endpoints is in the shortest path tree. In [14], these updates are performed when *at least* one of its endpoints is in the shortest path tree. It is easy to prove that our improvement is correct. The first improvement is also applied to the YTO algorithm.

2.5. Lawler's algorithm.

Lawler's algorithm [15] is given in Figure 8. It is based on the same observation as the parametric shortest path algorithms. It also uses the fact that λ^* of G lies between the minimum and the maximum arc weights in G. Lawler's algorithm does a binary search over the possible values of λ^* and checks for a negative cycle in G_λ every iteration. If one is found, then the chosen λ is too large so it is decreased; if not, it is too small so it is increased. Lawler's algorithm terminates when the interval for the possible values of λ^* becomes too small. The size of that interval, ϵ, determines the precision of the algorithm.

In our implementation, we observed that line 5 in Figure 8 may sometimes evaluate to either H or L even though they are different. This is a precision problem due to the use of floating point numbers. When this problem occurs, the algorithm goes into an infinite loop. In order to prevent this problem, we added the checks in lines 9 and 11. They basically mean that if λ is too close to one of the bounds, then just terminate the algorithm because λ is good enough.

The performance of this algorithm depends on the width of the interval of H and L and the performance of the algorithm used in line 7 to find a negative cycle. We used the standard Bellman-Ford algorithm but optimized it for this algorithm. The time complexity of the best negative-cycle detection algorithms is the same as that of the Bellman-Ford algorithm but their performance in practice may differ considerably. Thus, using a faster negative-cycle detection algorithm in line 7 will improve the performance of Lawler's algorithm. Another improvement is to set H to a smaller value. Since the mean of any cycle is an upper bound on the minimum cycle mean, we can set H to the mean of any cycle instead of the maximum arc weight. In order to ensure that we have a small cycle mean for H, we can compute the cycle mean in the graph in which each node has one successor arc that has the minimum weight among all the successor arcs of that node. This graph has n arcs, which also speeds up the computation of the cycle mean within it.

2.6. Howard's Algorithm.

Howard's algorithm [6] is given in Figure 9. Although this algorithm is well known in the control community, the version in [6] which we use here is slightly different than the classical version known to the control community. In particular, the authors of [6] have adapted the algorithm specifically for the problems we consider in this paper, whereas the previous versions addressed a broader class of problems. In addition, the version found in [6] uses linear time per iteration, whereas previous versions used $O(n^3)$ time per iteration.

Howard's algorithm is similar to the style of the parametric shortest path algorithms except that it starts with a large λ and decreases λ until the shortest paths in G_λ are well defined. For a given λ, the algorithm attempts to find the shortest paths from every node to an chosen node w. In doing so, it either discovers that the shortest paths are well defined in which case the correct λ has been found

or it discovers a negative cycle in G_λ. In the latter case, the negative cycle has a smaller mean weight than the current λ. In this case, λ can be updated to the mean weight of the new cycle and the process continues.

Howard's algorithm computes the shortest paths in G_λ in a somewhat unusual way. A *policy graph* is maintained which is simply a subgraph of G such that the out-degree of each node is exactly one. The designated sink node w is always a node on a cycle in the policy graph. The current λ is always the mean weight of a cycle in the policy graph. The policy graph is an estimate on the current shortest paths from each node to w.

Howard's algorithm maintains a distance $d(u)$ of a node u which is an estimate on the distance to the designated node. In each iteration of Howard's algorithm, a pass is made through all the arcs and each $d(u)$ is updated as follows:

$$d(u) = \min_{(u,v)\in E} \{d(v) + w(u,v) - \lambda\}. \tag{5}$$

The policy graph is also updated so that the arc going out of u points to the node v for which the minimum was achieved. In the next iteration, λ is chosen to be the mean weight of a cycle in the policy graph, w is updated and the process continues. It is also possible to choose the cycle with the minimum mean in the policy graph. Since the policy graph contains n arcs, such a cycle can be found in linear time.

If there is an iteration in which the policy graph does not change, then Equation 5 has been satisfied for every node. This can only happen when the graph G_λ does not have a negative cycle. In this case, λ is the minimum mean cycle weight and each $d(u)$ is its distance in G_λ to some node on a zero length cycle. Note that Equation 5 is almost equivalent to Equation 3 with the difference that the places of $d(u)$ and $d(v)$ in the latter are exchanged in the former.

In the implemention of the algorithm, $d(u)$ in Equation 5 and the arc going out of u in the policy graph are only updated if $d(u)$ will improve by some ϵ which is the precision of the algorithm. Note that unlike Lawler's algorithm, Howard's algorithm does not use ϵ to control the value of λ.

The beauty of Howard's algorithm is that each iteration of the algorithm is extremely simple and requires only $\Theta(m)$ time. Meanwhile, although it ensures that the value of λ is non-increasing from one iteration to another, it usually manages to make significant progress in decreasing its value in a very few number of iterations.

In [6], it is proven that Howard's algorithm does terminate by showing that the algorithm never has the same policy graph during its run. Thus, the running time is bounded by the total number of policy graphs, which is the product of the out-degrees of all the nodes in the input graph G. Below, we prove two bounds on the running time of Howard's algorithm. For our proofs, we use a slightly different version of Howard's algorithm than the one in Figure 9. The new version is given in Figure 10. The new version essentially selects the cycle with the minimum mean in the policy graph in an iteration rather than an arbitrary cycle in in the policy graph.

Theorem 1: λ decreases by at least ϵ/n at least every n iterations of the main loop of Howard's algorithm in Figure 10, where ϵ is the precision of the algorithm.

The proof of this theorem appears in the Appendix. Of course, this is a very pessimistic bound since in our experiments, λ almost always decreases and usually by much more than ϵ.

The theorem yields the following two corollaries.

Corollary 1: The running time of Howard's algorithm in Figure 10 is at most $O(nm\alpha)$, where α is the number of simple cycles in the graph G.

Corollary 2: The running time of Howard's algorithm in Figure 10 is at most

$$O(n^2 m(w_{max} - w_{min})/\epsilon),$$

w_{max} and w_{min} are the maximum and minimum arc weights in the graph G and ϵ is the precision of the algorithm.

2.7. Scaling algorithms.

The OA1 and OA2 algorithms [19] are in this category. They assume that the arc weights are integers bounded by W. If W is polynomial in n, then these algorithms are asymptotically the fastest algorithms. The OA2 algorithm applies scaling to a hybrid version of an assignment algorithm, called the auction algorithm, and the successive shortest path algorithm. It uses an approximate binary search technique. The OA1 algorithm is the same as the OA2 algorithm except that it does not use the successive shortest path algorithm. We do not give the pseudocode for the OA1 and OA2 algorithms because the pseudocode is quite long when described in as much detail as the others in this paper. For the pseudocode, we refer to [19].

3. Experimental Framework

We programmed the algorithms in C++ using the LEDA library version 3.4.1. This library is a template library for efficient data types and algorithms [18]. It may be slower than a custom implementation of these data structures for the sole purpose of comparing the algorithms in this paper but we found it quite efficient and useful. In addition, the data structures of this library are the basis of every algorithm so we expect that the comparison of the same kind of operations between two different algorithms is accurate.

In order to ensure uniformity of implementation, all the algorithms were implemented by one of us. Thus, they were programmed in the same style. In addition, the same routines to read and write the graph and to find its strongly connected components were used. We also flattened each algorithm in that we manually inlined all the functions other than the functions needed by the LEDA data types. This eliminated the overhead of function invocations. The total size of the programs is approximately 2700 lines of C++ code.

We compiled and linked each program using the Sun C++ compiler CC version 3.0.1 under the O4 optimization option. We conducted the experiments on a Sun Sparc 20 Model 512 with two CPUs, 64 MB of main memory, and 105 MB of virtual memory. The operating system was SunOS version 5.5.1.

We did two sets of experiments: one to measure the running time of each algorithm and another to count the key operations of each algorithm, as advocated in [2]. Our test suite contains random graphs, generated using SPRAND [5], and cyclic sequential multi-level logic benchmark circuits, obtained from the 1991 Logic Synthesis and Optimization Benchmarks [21]. SPRAND produces a graph with n nodes

and m arcs by first building a Hamiltonian cycle on the nodes and then adding $m - n$ arcs at random. This cycle makes the graph strongly connected. We generated 10 versions of each random graph. The experimental data reported for these graphs in this paper are the average over these 10 versions. The arc weights in the random graphs as well as the benchmark circuits were randomly assigned such that they are uniformly distributed in $[1, 10000]$, which is the default weight interval in SPRAND. The properties of the random graphs and circuits are given together with the experimental results, e.g., see Tables II and IV.

A close look at our test suite will reveal that the random graphs in our test suite are quite sparse. The reason for this choice is that we wanted to have random graphs which best represent real circuits, which are sparse. In addition, all the applications of these algorithms as mentioned in § 1.2 usually have sparse graphs. In general, we were not interested in evaluating the performance of these algorithms on graphs that do not exist in any application that we know of.

We did more experiments than were reported in this paper. Since the trend for the dependence of the performance on the graph parameters is evident from those that we included in this paper, we did not see any need to include more experimental results. When doing our experiments, we tried to follow the guidelines in [1], e.g., the graph parameters, the number of versions of the same graph, etc. When comparing the algorithms using their operation counts, we compared only the relevant ones because all the algorithms do not have the same kind of operations. For instance, we compared only the KO and YTO algorithms for the number of heap operations.

We are not aware of any other published work that contains a comparison of the algorithms that we compare. As a matter of fact, most of the works that introduced these algorithms do not present any experimental results on these algorithms. Worse, some of the works are not even aware of the others. Hence, our study should be taken as a first step in the direction of obtaining better versions of these algorithms using the insight from experimental analysis. We believe that these algorithms have a great deal of potential for improvement.

4. Experimental results and observations

We now list the experimental results and our observations as they apply to our test suite. We also present the results of some of the experiments on the representative operation counts of the algorithms. We will explain these counts very briefly because of lack of space.

4.1. The minimum cycle mean and the graph parameters.

For the random graphs, the minimum cycle mean is almost independent of the number of nodes, and it changes inversely with the density of the graph, as shown in Figure 1. This is expected because as the density of a graph increases, the graph contains more cycles and the critical cycles get smaller. This simple observation will be used to explain the behavior of some of the algorithms.

4.2. KO versus YTO.

In our implementation, both algorithms use Fibonacci heaps, which is the default heap data structure in LEDA. The YTO algorithm stores the node keys in the heap rather than the arc keys, which is the case in the KO algorithm, in an attempt to decrease the number of increase key operations. In

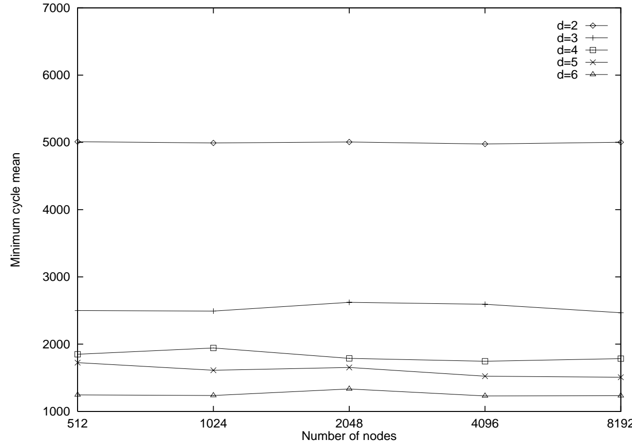

Figure 1. The minimum cycle mean λ versus the number of nodes n on the random graphs. Here d is the density of the graph, $d = 2m/n$, and the x-axis is logarithmic.

our implementation, increase and decrease key operations are implemented using heap insertions and deletions.

These two algorithms can be compared in terms of the number of iterations, the number of heap insertions and deletions in total and per iteration, and the running time. Tables VI and VII compare these algorithms for their parameters other than the running time on the random graphs and the circuits, respectively. We can see that both the algorithms perform almost the same number of iterations on each test case; however, the YTO algorithm provides savings in the number of heap operations, especially in the number of insertions. The savings are more on the random graphs, and they get better as the density increases because the rate of increase in these numbers in the KO algorithm is larger.

As shown in Tables II and IV, on the random graphs, their running times are comparable but the YTO algorithm performs a bit faster when the density increases. This is expected because the YTO algorithm performs fewer heap operations. On the circuits, their running times are comparable for the smaller circuits but the KO algorithm is a bit faster when the size of the circuits gets larger. This is unexpected. Our explanation for this behavior is that the savings achieved by the decrease in the number of heap operations are not enough to compensate for the extra time needed to maintain the node keys in the YTO algorithm.

4.3. Number of iterations.

Burns', KO, YTO, and Howard's algorithms perform a number of iterations before they converge. An upper bound on the number of iterations for the first three algorithms is n^2, and that for Howard's algorithm is the product of the out-degrees of all the nodes. We have also measured the value of k when the HO algorithm terminates. We refer to this value as the number of iterations of the HO algorithm although it is not one in the sense of the other algorithms. It is always less than n.

As shown in Tables VIII and IX, the number of iterations is always less than the number of nodes for each algorithm. One anomaly is the behavior of Howard's algorithm on a random graph of 512 nodes and 1024 arcs. It seems that unless $n = m$, the number of iterations for the first three algorithms is around $n/2$ on the random graphs, each of which is strongly connected. Moreover, Burns' algorithm performs fewer number of iterations than the KO algorithm, and the KO and YTO algorithms perform

the same number of iterations. The number of iterations of the Howard's algorithm is drastically small compared to the other algorithms. In [6], it is conjectured that the number of iterations is $O(\lg n)$ on the average, and it is $O(m)$ in the worst case. Our experiments support the worst case conjecture. They also show that the number of iterations for Howard's algorithm and the HO algorithm gets smaller as the density of the graph increases although some anomalies exist. This can be explained by the first observation.

4.4. Karp's algorithm and its variants.

The improvement achieved by the DG algorithm in the number of arcs visited during the computation of $D_k(v)$ for each v is very small on the random graphs, indicating that it is not effective for dense graphs. Note that when $n = m$, the running time of the DG algorithm is linear whereas that of Karp's algorithm is quadratic. The improvement on the circuits is far better, which explains the better performance of the DG algorithm than Karp's algorithm.

The space efficient version of Karp's algorithm, the Karp2 algorithm, roughly doubles its running time, as expected. The space efficiency of the Karp2 algorithm is directly applicable to the DG and HO algorithms. The most effective improvement on the Karp's algorithm is the HO algorithm. Its running time is even better than those algorithms which are asymptotically faster than it. Extrapolating from the Karp2 algorithm, we can say that the space efficient version of the HO algorithm will double its running time, which still maintains its superiority to most of the other algorithms.

4.5. Running times.

The running time comparisons are given in Tables II, III, IV, and V. The results show the following:

1. Howard's algorithm is the fastest. Its running time is impressive. The HO algorithm ranks second, which indicates that the early termination scheme in the HO algorithm is very effective. The slowest algorithm is Lawler's algorithm. This ranking is based on our implementation choices.
2. The good performance of Karp's algorithm, especially on small test cases, is mostly due to its simplicity; it contains three simple nested loops. Its simplicity facilitates its optimization by a compiler, e.g., when compiled without optimization, the DG algorithm almost always beats it. However, it seems that as the number of nodes gets larger, its performance degrades more rapidly.
3. Burns' algorithm is slower than the KO and YTO algorithms although it performs fewer number of iterations and it does not perform expensive operations such as heap operations. We attribute this behavior to the fact that it is not incremental; every iteration builds from the scratch.
4. The OA1 and OA2 algorithms are not as fast as their running time implies. They are in general slower than Karp's algorithm. We attribute much of this to their complexity; they are more difficult to optimize than the other algorithms. It is interesting to note that these algorithms perform very well for the two largest circuits and that they perform very badly when $n = m$. We have also performed experiments on the parameters of the OA1 and OA2 algorithms. We observed that the successive shortest path algorithm, which comes after the assignment algorithm in the OA2 algorithm, is never invoked. Experiments on some other random graphs show that whenever it is invoked on a graph, it is invoked only a few times. This indicates that the assignment algorithm used in these algorithms is very good at assigning all but a few of the nodes, so it may be more

profitable to focus on improving it rather than the successive shortest path algorithm.

5. Conclusions and Future Work

We have presented a comprehensive experimental evaluation of ten leading algorithms for the minimum mean cycle problem. This is the first such study. We systematically compare these algorithms on random graphs as well as benchmark circuits and provide important insights into their individual performance as well as relative performance in practice. One of the most surprising results of this study is that Howard's algorithm, a well known algorithm to the stochastic control community but an unknown algorithm to the graph theory community, is by far the fastest algorithm on the graphs tested in this study. Unfortunately, the only known bound on the running time of this algorithm is exponential. We present two new bounds on its running time that are stronger on most graphs.

We are working on improving these algorithms based on the insight that we have obtained from this study. We have achieved some encouraging results for Howard's algorithm and Lawler's algorithm. We are going to present these results in a future paper.

References

[1] Ahuja, R. K., Kodialam, M., Mishra, A. K., and Orlin, J. B. Computational investigation of maximum flow algorithms. *European J. of Operational Research*, 97 (1997), 509–542.

[2] Ahuja, R. K., Magnanti, T. L., and Orlin, J. B. *Network Flows.* Prentice Hall, Upper Saddle River, NJ, USA, 1993.

[3] Bacelli, F., Cohen, G., Olsder, G. J., and Quadrat, J.-P. *Synchronization and Linearity.* John Wiley & Sons, New York, NY, USA, 1992.

[4] Burns, S. M. Performance analysis and optimization of asynchronous circuits. PhD thesis, California Institute of Technology, 1991.

[5] Cherkassky, B. V., Goldberg, A. V., and Radzik, T. Shortest ptah algorithms: Theory and experimental evaluation. In *Proc. 5th ACM-SIAM Symp. on Discrete Algorithms* (1994), pp. 516–525.

[6] Cochet-Terrasson, J., Cohen, G., Gaubert, S., McGettrick, M., and Quadrat, J.-P. Numerical computation of spectral elements in max-plus algebra. In *Proc. IFAC Conf. on Syst. Structure and Control* (1998).

[7] Cuninghame-Green, R. A., and Yixun, L. Maximum cycle-means of weighted digraphs. *Applied Math.-JCU 11* (1996), 225–34.

[8] Dasdan, A., and Gupta, R. K. Faster maximum and minimum mean cycle algorithms for system performance analysis. To appear *ieee transactions on computer-aided design*, 1997.

[9] Gerez, S. H., de Groot, S. M. H., and Herrmann, O. E. A polynomial-time algorithm for the computation of the iteration-period bound in recursive data-flow graphs. *IEEE Trans. on Circuits and Syst.-1 39*, 1 (Jan. 1992), 49–52.

[10] Gondran, M., and Minoux, M. *Graphs and Algorithms.* John Wiley and Sons, Chichester, NY, USA, 1984.

[11] Hartmann, M., and Orlin, J. B. Finding minimum cost to time ratio cycles with small integral transit times. *Networks 23* (1993), 567–74.

[12] Ito, K., and Parhi, K. K. Determining the minimum iteration period of an algorithm. *J. VLSI Signal Processing 11*, 3 (Dec. 1995), 229–44.

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

[14] Karp, R. M., and Orlin, J. B. Parametric shortest path algorithms with an application to cyclic staffing. *Discrete Applied Mathematics 3* (1981), 37–45.

[15] Lawler, E. L. *Combinatorial Optimization: Networks and Matroids*. Holt, Reinhart, and Winston, New York, NY, USA, 1976.

[16] Mathur, A., Dasdan, A., and Gupta, R. K. Rate analysis of embedded systems. *ACM Trans. on Design Automation of Electronic Systems 4*, 2 (Apr. 1999).

[17] Megiddo, N. Combinatorial optimization with rational objective functions. *Mathematics of Operations Research 4*, 4 (Nov. 1979), 414–424.

[18] Mehlhorn, K., and Naher, S. LEDA: A platform for combinatorial and geometric computing. *Comm. of the ACM 38*, 1 (1995), 96–102.

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

[20] Teich, J., Sriram, S., Thiele, L., and Martin, M. Performane analysis and optimization of mixed asynchronous synchronous systems. *IEEE Trans. Computer-Aided Design 16*, 5 (May 1997), 473–84.

[21] Yang, S. Logic synthesis and optimization benchmarks user guide version 3.0. Tech. rep., Microelectronics Center of North Carolina, Jan. 1991.

[22] Young, N. E., Tarjan, R. E., and Orlin, J. B. Faster parametric shortest path and minimum-balance algorithms. *Networks 21* (1991), 205–21.

Appendix

A. The Proof of the time complexity of Howard's Algorithm

Proof of Theorem 1. We refer in this proof to the outline of the algorithm in Figure 10. Consider G_π before an after the for-loop of line 21. We will denote the resulting graph after the for-loop of line 21 as G'_π. Note that G_π has exactly one cycle after the while-loop of line 13. We will call an arc *new* if it is in G'_π but not in G_π. If there are no new arcs in G'_π, then the algorithm stops. If there is a cycle in G'_π with any new arcs, then λ must decrease. This is because we know that for each $u \in V$, $d(u) + \lambda - w(u, \pi(u)) - d(\pi(u)) = 0$. The policy $\pi(u)$ will only get reassigned to, say v, if $d(u) + \lambda - w(u, v) - d(v) > \epsilon$. Thus, for any new arc in G'_π, we know that $d(u) + \lambda - w(u, v) - d(v) > \epsilon$. Thus, if there is a cycle C with a new arc in G'_π, we know that

$$\sum_{(u,v)\in C} d(u) + \lambda - w(u, v) - d(v) = \sum_{(u,v)\in C} \lambda - w(u, v) = |C|\lambda - w(C) > \epsilon. \tag{6}$$

Let the mean of C be $\lambda(C)$, where $\lambda(C) = w(C)/|C|$. Using Equation 6, we have

$$|C|\lambda - w(C) = |C|\lambda - |C|\lambda(C) = |C|(\lambda - \lambda(C)) > \epsilon, \tag{7}$$

which implies that

$$\lambda - \lambda(C) > \frac{\epsilon}{|C|} \geq \frac{\epsilon}{n}. \tag{8}$$

Thus, the mean $\lambda(C)$ of cycle C is at least ϵ/n less than λ.

We only have to worry now about the case where there are new arcs in G_π but no cycles with new arcs. We will prove that this can happen at most $n-1$ times in a row. Let $G^1_\pi, G^2_\pi, \ldots, G^x_\pi$ be the sequence of policy graphs where in each iteration, there are no cycles with new arcs. This means that all these graphs have exactly one cycle and it is the same cycle C in every graph. This also implies that the chosen vertex u from which we do our reverse BFS is always the same. Since each graph G^j_π has exactly one cycle, every vertex v has a unique simple path to the chosen vertex u. Given a graph G^j_π, we will assign each vertex v a number $n(v)$ which is the minimum number of arcs in this path from v to u. We will prove that for $k > j$, there will be no new paths in G^k_π to u with j or fewer arcs. This will imply that after G^n_π, either the graph will not change (in which case the algorithm terminates) or there will be a path with at most n arcs, which means there is a cycle with a new arc in the graph.

We will prove our claim by contradiction. Consider the first time in making G^j_π for some j, that $\pi(y)$ is reassigned from x to z where $n(z) < j - 1$. Let $d'(y)$ and $d'(z)$ be the $d(\cdot)$ values for y and z at the time of this reassignment. We know that $d'(y) > d'(z) + w(y, z) - \lambda$. Let $d''(y)$ and $d''(z)$ be the values for $d(\cdot)$ at the time the arc (y, z) is considered in making G^{j-1}_π. Since this is the first time that the claim has been violated, we know that the path from z to u has remained unchanged since G^{j-2}_π which means that $d''(z) = d'(z)$. We also know that $d''(y) \geq d'(y)$ since the $d(\cdot)$ values can not increase in time. Putting these inequalities together, we get that $d''(y) > d''(z) + w(y, z) - \lambda$ which implies that at the time G^{j-1}_π was being made, $\pi(y)$ would have been reassigned to z and $d(y)$ would have been assigned a value of $d''(z) + w(y, z) - \lambda$ which is strictly less than $d'(y)$, the value of $d(y)$ at the time G^j_π is made. Since the $d(\cdot)$ values can not increase, this is a contradiction. ∎

B. Algorithms in Pseudocode

```
Input: A strongly connected digraph G = (V, E).
Output: λ* of G.
     /* Initialize */
1    for each node u ∈ V do
2      d(u) ← 0

     /* Set λ to the minimum arc weight. */
3    λ ← +∞
4    for each arc (u, v) ∈ E do
5      λ ← MIN{λ, w(u, v)}

     /* Iterate */
6    while (true) do
       /* Find the set E' of critical arcs */
7      E' ← φ
8      for each arc e = (u, v) ∈ E do
9        if (d(v) − d(u)) − (w(u, v) − λ) = 0 then
10         E' ← E' + {e} /* e is critical */

11     Topologically sort the graph G' = (V, E') of critical arcs.

12     if (G' is cyclic) then
13       return λ

       /* Compute the (negative) node heights h in G' */
14     for each node v ∈ V in the topological order of G' do
15       if (v has no predecessors) then
16         h(v) ← 0
17       else
18         for each predecessor node u of node v in E' do
19           h(v) ← MIN{h(v), h(u) − 1}

       /* Compute θ */
20     θ = −∞
21     for each arc (u, v) ∈ E do
22       if (h(v) − h(u) + 1 > 0) then
23         θ = MAX{θ, ((d(v) − d(u)) − (w(u, v) − λ))/(h(v) − h(u) + 1)}

       /* Update using θ */
24     λ ← λ − θ
25     for each node u ∈ V do
26       d(u) ← d(u) − θ * h(u)
```

Figure 2. Burns' algorithm.

```
Input: A strongly connected digraph G = (V, E).
Output: λ* of G.
     /* Initialize */
1    for each k = 0 to n do
2      for each node u ∈ V do
3        d_k(u) ← +∞
4    d_0(s) ← 0 /* s is any node in V */

     /* Compute the distances */
5    for each k = 1 to n do
6      for each node v ∈ V do
7        for each arc (u, v) ∈ E then
8          d_k(v) ← MIN{d_k(v), d_{k-1}(u) + w(u, v)}

     /* Compute λ using Karp's theorem */
9    for each node u ∈ V do
10     λ' ← +∞
11     for each k = 0 to (n − 1) do
12       λ' ← MAX{λ', (d_n(u) − d_k(u))/(n − k)}
13     λ ← MIN{λ, λ'}

14   return λ
```

Figure 3. Karp's algorithm.

```
Input: A strongly connected digraph G = (V, E).
Output: λ* of G.
     /* Initialize */
1    for each node u ∈ V do
2      level(u) ← −1
3    Find a "good" or arbitrary source node s.
4    d_0(s) ← 0; π_0(s) ← −1
5    level(s) ← 0; ENQUEUE(Q, < 0, s >)

     /* Compute the distances */
7    < k, u >← DEQUEUE(Q)
8    do
9      for each arc (u, v) ∈ E do
10       if (level(v) < k + 1) then
11         ENQUEUE(Q, < k + 1, v >)
12         π_{k+1}(v) ← level(v)
13         level(v) ← k + 1
14         d_{k+1}(v) ← +∞
16       d_{k+1}(v) ← MIN{d_{k+1}(v), d_k(u) + w(u, v)}
18     < k, u >← DEQUEUE(Q)
19   while (k < n)

     /* Compute λ using Karp's theorem */
20   λ ← +∞
21   for each node u ∈ V do
22     if (level(u) = n) then
23       λ' ← −∞
24       k ← π_n(u)
25       while (k > −1) do
27         λ' ← MAX{λ', (d_n(u) − d_k(u))/(n − k)}
29         k ← π_k(u)
31       λ ← MIN{λ, λ'}
33   return λ
```

Figure 4. Dasdan-Gupta's algorithm (the DG algorithm).

Input: A strongly connected digraph $G = (V, E)$.
Output: λ^* of G.

```
     /* Initialize */
1    for each k = 0 to n do
2      for each node u ∈ V do
3        d_k(u) ← +∞
4        π_k(u) ← −1
5    d_0(s) ← 0 /* s is any node in V */

     /* Compute the distances and the means of the cycles found */
6    for each k = 1 to n do
7      for each node v ∈ V do
8        for each arc (u, v) ∈ E do
9          if (d_{k−1}(u) + w(u, v) < d_k(v)) then
10           d_k(v) ← d_{k−1}(u) + w(u, v) /* MIN */
11           π_k(v) ← u

12     if ((k is a power of 2) OR (k = n)) then
         /* Compute λ using the mean of all the cycles found so far */
13       λ ← +∞
14       found ← false /* Set to true when a cycle is found */
15       for each node u ∈ V do
16         for each node v ∈ V do
17           level(v) ← 0
18         if (d_k(u) ≠ +∞) then
19           v ← u
20           for each j = k downto 0 then
21             if (level(v) > 0) then /* A cycle is found */
22               λ ← MIN{λ, (d_{level(v)}(v) − d_j(v))/(level(v) − j)}
23               found ← true /* A cycle is found */
24             level(v) ← j
25             v ← π_j(v)

         /* If at least one cycle is found, check for the criticality */
26       if (found = true) then
           /* Find the node potentials */
27         for each node u ∈ V do
28           d(u) = +∞
29           for each j = 0 to k do
30             d(u) ← MIN{d(u), d_j(u) − j ∗ λ}

           /* Check for criticality of each arc */
31         done ← true
32         for each arc e = (u, v) ∈ E do
33           if ((d(v) − d(u)) − (w(u, v) − λ) = 0) then
34             done ← false

35         if (done = true) then
36           return λ
```

Figure 5. Hartmann-Orlin's algorithm (the HO algorithm).

Input: A strongly connected digraph $G = (V, E)$.
Output: λ^* of G.

/* Initialize */
1 Let $G' = (V', E)$ such that $V' = V + \{s\}$.

/* Find the shortest path tree $T(s)$ rooted at s on G' */
2 $V[T(s)] \leftarrow V'$
3 **for** $u \in V$ **do**
4 $\quad E[T(s)] \leftarrow E[T(s)] + \{(s, u)\}$ with $w(s, u) = 0$

/* Initialize the distance d and length l of each node in $T(s)$ */
5 **for** each node $u \in V[T(s)]$ **do**
6 $\quad d(u) \leftarrow 0;\ l(u) \leftarrow 1$
7 $l(s) \leftarrow 0$

/* Insert every arc into the heap H based on their keys */
/* For an arc e, H holds $< key(e), e >$ where $key(e) = w(e)$ */
8 $HEAP_INSERT(H, < +\infty, _ >)$
9 **for** each arc $(u, v) \in E$ **do**
10 $\quad HEAP_INSERT(H, < w(u, v), (u, v) >)$

/* Iterate */
11 **while** (true) **do**
12 $\quad < \lambda, (u, v) > \leftarrow HEAP_FIND_MIN(H)$
13 $\quad$ **if** $(\lambda = +\infty)$ **then**
14 $\quad\quad$ **return** λ /* G is acyclic */

15 $\quad$ Let $T(v)$ be the tree rooted at v in $T(s)$.
16 $\quad$ To find $T(v)$, run BFS starting from v on $T(s)$.
17 $\quad$ **if** $(u \in V[T(v)])$ **then**
18 $\quad\quad$ **return** λ /* A critical cycle is found */

$\quad$ /* Update the distance and length of each node in $T(v)$ */
19 $\quad \Delta_1 \leftarrow d(u) + w(u, v) - d(v);\ \Delta_2 \leftarrow l(u) + 1 - l(v)$
20 $\quad$ **for** each node $x \in V[T(s)]$ **do**
21 $\quad\quad$ **if** $(x \in V[T(v)])$ **then**
22 $\quad\quad\quad d(x) \leftarrow d(x) + \Delta_1;\ l(x) \leftarrow l(x) + \Delta_1$

$\quad$ /* Find the new shortest path tree */
23 $\quad$ Delete the unique predecessor arc of v from $T(s)$.
24 $\quad$ Insert (u, v) into $T(s)$.

$\quad$ /* Update the keys of the other arcs in E */
25 $\quad$ **for** each arc $(y, z) \in E$ **do**
$\quad\quad$ /* No update if both y and x are in $T(s)$ */
26 $\quad\quad$ **if** $((y \in V[T(v)])\ XOR\ (z \in V[T(s)]))$ **then**
27 $\quad\quad\quad HEAP_DELETE(H, < _, (y, z) >)$
28 $\quad\quad\quad \Delta_2 \leftarrow l(y) + 1 - l(z)$
29 $\quad\quad\quad$ **if** $(\Delta_2 > 0)$ **then**
30 $\quad\quad\quad\quad \Delta_1 \leftarrow d(y) + w(y, z) - d(z)$
31 $\quad\quad\quad\quad HEAP_INSERT(H, < (\Delta_1/\Delta_2), (y, z))$
32 $\quad\quad\quad$ **else**
33 $\quad\quad\quad\quad HEAP_INSERT(H, < +\infty, (y, z))$

Figure 6. Karp-Orlin's algorithm (the KO algorithm).

Input: A strongly connected digraph $G = (V, E)$.
Output: λ^* of G.

```
1-7    Put lines 1-7 of the KO algorithm here.

       /* Find the node and arc keys */
8      for each node u ∈ V do
9        key(u) ← NULL
       for each arc e = (u, v) ∈ E do
         key(e) ← w(e)
         if ((key(v) = NULL) OR (key(e) < key(key(v)))) then
           key(v) ← e

       /* Insert every arc into the heap H based on their keys */
10     HEAP_INSERT(H, < +∞, _ >)
11     for each arc e = (u, v) ∈ E do
12       HEAP_INSERT(H, < key(e), e >)

13     for each node u ∈ V do
14       changed(u) ← false

       /* Iterate */
15     while (true) do
16-27    Put lines 12-24 of the KO algorithm here.

         /* Update the keys of the other nodes and arcs in G */
28       for each arc e = (y, z) ∈ E do
           /* No update if both y and x are in T(s) */
29         if ((y ∈ V[T(v)]) OR (z ∈ V[T(s)])) then
30           Δ_2 ← l(y) + 1 − l(z)
31           if (Δ_2 > 0) then
32             Δ_1 ← d(y) + w(y, z) − d(z)
33             key(e) ← Δ_1/Δ_2
34           else
35             key(e) ← +∞

36           if (e = key(z)) then
37             changed(z) ← true
38           else if (key(e) < key(key(z))) then
39             key(z) ← e
40             changed(z) ← true

         /* Update the heap */
41       for each node u ∈ V do
42         if (changed(u) = true) then
43           changed(u) ← false
44           HEAP_DELETE(H, < _, key(u) >)
45           HEAP_INSERT(H, < key(key(u)), key(u))
```

Figure 7. Young-Tarjan-Orlin's algorithm (the YTO algorithm).

```
Input: A strongly connected digraph G = (V, E).
Output: λ* of G.
       /* Find the min and max arc weights L and H */
1      L ← +∞; H ← −∞
2      for each arc (u, v) ∈ E do
3        L ← MIN{L, w(u, v)}; H ← MAX{H, w(u, v)}

       /* Iterate */
4      while (H − L > ε) do
         /* Find new λ */
5        λ ← (H + L)/2

6        Find G_λ by subtracting λ from each arc weight.
7        Check to see if G_λ has a negative cycle.

         /* Update the interval for λ */
8        if (G_λ has a negative cycle) then
           /* λ is too big so decrease it */
9          if (H − λ < (ε/2)) then
10           return λ
11         H ← λ
12       else
           /* λ is too small so increase it */
13         if (λ − L < (ε/2)) then
14           return λ
15         L ← λ
```

Figure 8. Lawler's algorithm.

```
Input: A strongly connected digraph G = (V, E).
Output: λ* of G.
      /* Initialize */
1     for each node u ∈ V do
2       d(u) ← +∞
3     for each arc (u, v) ∈ E do
4       if (w(u, v) < d(u)) then
5         d(u) ← w(u, v)
6         π(u) ← v /* policy */

      /* Iterate */
7     while (true) do
        /* Find the set E_π of policy arcs */
8       E_π ← {(u, π(u)) ∈ E}

        /* Compute λ from an arbitrary cycle C */
9       Find an arbitrary cycle C in G_π = (V, E_π).
10      λ ← w(C)/|C|
11      Select an arbitrary node u ∈ C.

        /* Compute the node distances using the reverse BFS */
12      if (there is a path from v to u in G_π) then
13        d(v) ← d(π(v)) + w(v, π(v)) − λ

        /* Improve the node distances */
14      improved ← false
15      for each arc (u, v) ∈ E do
16        δ(u) ← d(u) − (d(v) + w(u, v) − λ)
17        if (δ(u) > 0) then
18          if (δ(u) > ε) then
19            improved ← true
20          d(u) ← d(v) + w(u, v) − λ
21          π(u) ← v

        /* If not much improvement in the node distances, exit */
22      if (NOT improved) then
23        return λ
```

Figure 9. Howard's algorithm.

Input: A strongly connected digraph $G = (V, E)$.
Output: λ^* of G.

```
     /* Initialize */
1    for each node u ∈ V do
2      d(u) ← +∞
3    for each arc (u, v) ∈ E do
4      if (w(u, v) < d(u)) then
5        d(u) ← w(u, v)
6        π(u) ← v /* policy */

     /* Iterate */
7    while (true) do
       /* Find the set E_π of policy arcs */
8      E_π ← {(u, π(u)) ∈ E}
9      Examine every cycle in G_π = (V, E_π).
10     Let C be the cycle with the smallest mean w(C)/|C|.
11     λ ← w(C)/|C|

       /* Break all the cycles in G_π other than C */
12     Let V_C be the set of nodes which can reach C in G_π.
13     while (V_C ≠ V) do
14       Find u ∈ V − V_C such that there is a v ∈ V_C and (u, v) ∈ E.
15       π(u) ← v
16       V_C ← V_C + {u}

       /* Compute the node distances using the reverse BFS */
17     Select the lowest numbered node u ∈ C.
18     if (there is a path from v to u in G_π) then
19       d(v) ← d(π(v)) + w(v, π(v)) − λ

       /* Improve the node distances */
20     improved ← false
21     for each arc (u, v) ∈ E do
22       δ(u) ← d(u) − (d(v) + w(u, v) − λ)
23       if (δ(u) > ε) then
24         improved ← true
25         d(u) ← d(v) + w(u, v) − λ
26         π(u) ← v

       /* If not much improvement in the node distances, exit */
27     if (NOT improved) then
28       return λ
```

Figure 10. Howard's algorithm (for the proof of Theorem 1).

C. Experimental Results in Tables

TABLE II

BURNS', KO, YTO, HOWARD'S, AND HO ALGORITHMS FOR THE RUNNING TIME ON THE RANDOM GRAPHS.

n	m	Burns	KO (KO/Burns)	YTO (YTO/KO)	Howard (Howard/KO)	HO
512	512	3.48	1.51 (0.43)	1.67 (1.11)	0.01 (0.01)	1.00
512	768	2.34	1.04 (0.44)	1.12 (1.08)	0.16 (0.15)	0.32
512	1024	2.72	1.21 (0.44)	1.21 (1.00)	6.75 (5.58)	0.29
512	1280	4.11	1.82 (0.44)	1.73 (0.95)	0.17 (0.09)	0.31
512	1536	3.52	1.59 (0.45)	1.52 (0.96)	0.13 (0.08)	0.27
1024	1024	13.98	5.87 (0.42)	6.50 (1.11)	0.02 (0.00)	4.03
1024	1536	10.17	4.41 (0.43)	4.61 (1.05)	0.34 (0.08)	1.07
1024	2048	11.32	4.98 (0.44)	4.99 (1.00)	0.21 (0.04)	0.84
1024	2560	15.16	6.74 (0.44)	6.62 (0.98)	0.23 (0.03)	0.94
1024	3072	13.91	6.25 (0.45)	5.90 (0.94)	0.22 (0.04)	0.87
2048	2048	55.88	23.13 (0.41)	25.46 (1.10)	0.04 (0.00)	16.45
2048	3072	44.55	20.37 (0.46)	22.19 (1.09)	0.64 (0.03)	4.26
2048	4096	42.88	20.59 (0.48)	20.31 (0.99)	0.88 (0.04)	3.14
2048	5120	63.22	30.95 (0.49)	29.95 (0.97)	0.76 (0.02)	3.56
2048	6144	73.92	36.56 (0.49)	34.61 (0.95)	0.80 (0.02)	3.53
4096	4096	218.31	91.50 (0.42)	100.40 (1.10)	0.07 (0.00)	N/A
4096	6144	161.07	79.09 (0.49)	81.05 (1.02)	7.00 (0.09)	N/A
4096	8192	167.63	88.86 (0.53)	88.01 (0.99)	1.47 (0.02)	N/A
4096	10240	242.75	132.26 (0.54)	130.01 (0.98)	1.62 (0.01)	N/A
4096	12288	236.71	139.22 (0.59)	137.87 (0.99)	13.84 (0.10)	N/A
8192	8192	826.08	363.45 (0.44)	398.11 (1.10)	0.14 (0.00)	N/A
8192	12288	559.88	306.52 (0.55)	329.73 (1.08)	4.09 (0.01)	N/A
8192	16384	626.65	382.82 (0.61)	380.58 (0.99)	4.53 (0.01)	N/A
8192	20480	840.50	536.50 (0.64)	524.82 (0.98)	4.73 (0.01)	N/A
8192	24576	874.21	609.60 (0.70)	587.51 (0.96)	5.57 (0.01)	N/A

TABLE III

Karp's, DG, Lawler's, Karp2, OA1 algorithms for the running time on the random graphs.

n	m	Karp	DG	Lawler	Karp2 (Karp2/Karp)	OA1
512	512	0.79	0.06	11.09	1.41 (1.78)	328.88
512	768	0.98	1.03	6.51	1.83 (1.87)	5.80
512	1024	1.17	1.26	9.26	2.25 (1.92)	5.66
512	1280	1.37	1.47	10.62	2.71 (1.98)	6.98
512	1536	1.57	1.69	10.98	2.87 (1.83)	6.51
1024	1024	3.36	0.25	44.82	6.72 (2.00)	2790.12
1024	1536	4.17	4.66	34.67	7.87 (1.89)	12.34
1024	2048	5.05	5.64	30.33	9.04 (1.79)	13.78
1024	2560	5.91	6.63	54.77	10.82 (1.83)	23.67
1024	3072	6.77	7.56	51.91	14.60 (2.16)	17.13
2048	2048	13.48	1.02	186.35	21.80 (1.62)	20110.28
2048	3072	17.14	19.45	178.86	29.65 (1.73)	62.81
2048	4096	21.87	24.96	165.61	42.25 (1.93)	37.04
2048	5120	27.10	30.83	221.90	53.30 (1.97)	80.97
2048	6144	32.86	37.05	244.05	64.89 (1.97)	85.87
4096	4096	55.76	4.56	659.74	89.59 (1.61)	N/A
4096	6144	76.82	86.64	736.16	135.46 (1.76)	N/A
4096	8192	103.13	115.81	781.84	195.35 (1.89)	N/A
4096	10240	129.03	144.75	1305.47	259.19 (2.01)	N/A
4096	12288	156.70	173.94	1132.57	313.06 (2.00)	N/A
8192	8192	N/A	N/A	2819.30	355.00	N/A
8192	12288	N/A	N/A	2949.25	595.02	N/A
8192	16384	N/A	N/A	3708.98	858.01	N/A
8192	20480	N/A	N/A	5112.67	1110.84	N/A
8192	24576	N/A	N/A	5417.58	1905.20	N/A

TABLE IV

Burns', KO, YTO, Howard's, and HO algorithms for the running time on the circuits.

Name	n	m	Burns	KO	YTO	Howard	HO
s382	273	438	0.06	0.04	0.04	0.01	0.01
s344	274	388	0.24	0.13	0.18	0.01	0.07
s349	278	395	0.22	0.10	0.12	0.04	0.05
s526n	292	560	0.05	0.04	0.05	0.01	0.02
mult16a	293	582	0.24	0.12	0.10	0.02	0.10
s400	287	462	0.07	0.05	0.05	0.01	0.01
s444	315	503	0.10	0.07	0.07	0.04	0.03
s526	318	576	0.06	0.05	0.06	0.01	0.01
mult16b	333	545	0.01	0.02	0.01	0.00	0.00
s641	477	612	0.58	0.26	0.28	0.53	0.10
s713	515	688	0.73	0.36	0.35	0.77	0.25
mult32a	565	1142	0.87	0.35	0.41	0.04	0.25
mm9a	631	1182	0.51	0.28	0.29	0.04	0.11
s838	665	941	0.02	0.04	0.03	0.00	0.01
s953	730	1090	0.71	0.32	0.33	0.04	0.08
mm9b	777	1452	3.20	1.40	1.78	0.14	0.28
s1423	916	1448	2.00	0.96	1.00	0.14	0.92
sbc	1147	1791	0.72	0.36	0.39	0.03	0.12
mm30a	2059	3912	6.43	2.91	3.14	0.36	0.78
s5378	3076	4590	43.60	18.16	18.52	0.19	3.57
s9234	3083	4298	34.15	15.47	15.91	1.09	3.96
bigkey	3661	12206	0.45	0.52	0.50	0.08	0.11
dsip	4079	6602	18.57	8.98	9.43	3.30	6.53
s38584	20349	34563	982.98	908.48	924.02	10.87	N/A
s38417	24255	34876	500.97	236.41	255.74	59.45	N/A

TABLE V

KARP'S, DG, LAWLER'S, KARP2, OA1, OA2 ALGORITHMS FOR THE RUNNING TIME ON THE CIRCUITS.

Name	n	m	Karp	DG	Lawler	Karp2	OA1	OA2
s382	273	438	0.01	0.01	0.13	0.02	0.80	0.87
s344	274	388	0.06	0.06	0.87	0.10	1.48	1.39
s349	278	395	0.07	0.06	0.18	0.10	1.11	1.18
s526n	292	560	0.01	0.01	0.10	0.01	0.99	1.24
mult16a	293	582	0.13	0.14	2.03	0.23	1.39	1.31
s400	287	462	0.02	0.01	0.22	0.03	0.88	0.96
s444	315	503	0.02	0.02	0.22	0.05	1.29	1.20
s526	318	576	0.01	0.02	0.16	0.02	1.08	1.19
mult16b	333	545	0.00	0.01	0.02	0.00	0.34	0.37
s641	477	612	0.18	0.17	0.72	0.33	1.87	2.43
s713	515	688	0.30	0.24	0.81	0.48	1.85	2.01
mult32a	565	1142	0.64	0.64	4.21	1.03	2.60	2.90
mm9a	631	1182	0.18	0.18	0.74	0.31	2.33	2.55
s838	665	941	0.01	0.00	0.06	0.00	1.11	1.51
s953	730	1090	0.35	0.33	1.73	0.50	1.83	2.08
mm9b	777	1452	0.83	0.91	3.32	1.45	4.57	5.29
s1423	916	1448	2.31	2.39	7.59	3.28	4.40	4.78
sbc	1147	1791	0.26	0.22	2.20	0.39	3.23	3.62
mm30a	2059	3912	2.43	2.53	10.00	4.25	12.31	13.23
s5378	3076	4590	18.72	16.09	122.60	30.24	79.14	94.00
s9234	3083	4298	21.26	18.07	76.06	32.54	20.06	22.21
bigkey	3661	12206	0.11	0.11	1.79	0.18	13.46	15.13
dsip	4079	6602	11.71	11.19	74.18	21.81	15.43	16.82
s38584	20349	34563	N/A	N/A	14614.98	4776.98	165.53	167.20
s38417	24255	34876	N/A	N/A	1854.91	759.02	165.34	181.45

TABLE VI

THE KO VS YTO ALGORITHMS FOR THE NUMBER OF ITERATIONS AND HEAP OPERATIONS ON THE RANDOM GRAPHS.

		Total			Per iteration			
		KO/YTO			KO		YTO	
n	m	#iterations	#inserts	#deletes	#inserts	#deletes	#inserts	#deletes
512	512	1.00	0.58	0.48	3.00	2.00	5.19	4.19
512	768	0.96	1.31	1.20	6.81	3.95	5.01	3.17
512	1024	1.00	1.79	1.66	9.43	5.45	5.30	3.29
512	1280	1.01	2.09	1.92	11.10	7.17	5.36	3.77
512	1536	0.98	2.43	2.13	14.41	8.14	5.83	3.77
1024	1024	1.00	0.54	0.44	3.00	2.00	5.60	4.60
1024	1536	0.99	1.35	1.26	6.60	3.94	4.85	3.09
1024	2048	0.99	1.76	1.62	9.27	5.36	5.24	3.29
1024	2560	0.99	2.07	1.88	11.73	7.40	5.60	3.89
1024	3072	1.01	2.48	2.19	14.92	8.42	6.06	3.87
2048	2048	1.00	0.49	0.39	3.00	2.00	6.09	5.09
2048	3072	0.95	1.27	1.17	6.49	4.06	4.83	3.29
2048	4096	1.01	1.79	1.65	9.63	5.46	5.42	3.32
2048	5120	1.01	2.09	1.91	11.50	7.26	5.55	3.83
2048	6144	1.01	2.41	2.16	13.96	8.83	5.84	4.11
4096	4096	1.00	0.45	0.35	3.00	2.00	6.67	5.67
4096	6144	1.01	1.36	1.27	6.63	3.96	4.92	3.13
4096	8192	1.01	1.81	1.68	9.73	5.45	5.45	3.28
4096	10240	1.00	2.07	1.87	11.97	7.46	5.78	3.98
4096	12288	0.98	2.40	2.10	14.86	8.63	6.07	4.04
8192	8192	1.00	0.44	0.34	3.00	2.00	6.82	5.82
8192	12288	0.96	1.32	1.21	6.77	3.89	4.94	3.10
8192	16384	1.01	1.81	1.68	9.70	5.44	5.41	3.27
8192	20480	1.00	2.12	1.91	11.91	7.05	5.65	3.70
8192	24576	1.01	2.46	2.18	14.85	8.60	6.07	3.97

TABLE VII

The KO vs YTO algorithms for the number of iterations and heap operations on the circuits.

Name	n	m	Total KO/YTO #iterations	Total KO/YTO #inserts	Total KO/YTO #deletes	Per iteration KO #inserts	Per iteration KO #deletes	Per iteration YTO #inserts	Per iteration YTO #deletes
s382	273	438	1.02	1.22	1.15	4.82	2.82	4.02	2.49
s344	274	388	1.00	1.13	1.06	4.45	2.73	3.95	2.58
s349	278	395	1.01	1.18	1.12	4.70	2.67	4.03	2.41
s526n	292	560	1.00	1.14	1.07	4.45	2.94	3.91	2.74
mult16a	293	582	1.00	1.70	1.59	9.95	4.10	5.85	2.58
s400	287	462	1.02	1.23	1.17	4.72	2.88	3.89	2.51
s444	315	503	1.00	1.18	1.09	4.90	2.98	4.14	2.72
s526	318	576	1.00	1.12	1.06	4.28	2.67	3.81	2.52
mult16b	333	545	1.00	1.09	1.03	3.70	1.98	3.41	1.93
s641	477	612	1.00	1.12	1.06	4.53	2.73	4.06	2.57
s713	515	688	1.00	1.20	1.13	5.23	3.04	4.35	2.68
mult32a	565	1142	1.00	1.73	1.62	10.95	4.18	6.32	2.58
mm9a	631	1182	1.00	1.13	1.07	4.65	2.57	4.11	2.41
s838	665	941	1.01	1.04	0.99	3.30	1.94	3.20	1.96
s953	730	1090	1.01	1.29	1.21	5.82	3.06	4.56	2.54
mm9b	777	1452	1.00	1.10	1.02	4.64	3.03	4.21	2.96
s1423	916	1448	1.00	1.37	1.27	8.56	2.82	6.25	2.21
sbc	1147	1791	1.04	1.25	1.19	5.40	3.36	4.50	2.94
mm30a	2059	3912	1.00	1.17	1.11	4.98	2.72	4.25	2.45
s5378	3076	4590	1.01	1.26	1.19	5.27	3.44	4.20	2.92
s9234	3083	4298	1.01	1.26	1.18	5.39	3.13	4.31	2.67
bigkey	3661	12206	1.01	1.97	1.91	8.31	4.87	4.25	2.56
dsip	4079	6602	1.01	1.26	1.22	6.07	2.95	4.86	2.44
s38584	20349	34563	1.00	1.48	1.41	9.88	3.42	6.67	2.42
s38417	24255	34876	1.00	1.09	1.01	4.58	2.52	4.21	2.50

TABLE VIII

Burns', KO, YTO, Howard's, and HO algorithms for the number of iterations on the random graphs.

n	m	Burns	KO (KO/Burns)	YTO (YTO/KO)	Howard (Howard/KO)	HO
512	512	506.70	512.00 (1.01)	512.00 (1.00)	1.00 (0.00)	512.00
512	768	263.30	269.20 (1.02)	279.60 (1.04)	34.30 (0.13)	60.80
512	1024	248.60	258.00 (1.04)	256.80 (1.00)	1239.40 (4.80)	44.80
512	1280	317.90	326.40 (1.03)	323.30 (0.99)	26.90 (0.08)	48.00
512	1536	234.30	245.70 (1.05)	249.60 (1.02)	17.80 (0.07)	36.80
1024	1024	1001.20	1024.00 (1.02)	1024.00 (1.00)	1.00 (0.00)	1024.00
1024	1536	553.30	578.20 (1.05)	584.10 (1.01)	36.50 (0.06)	73.60
1024	2048	493.30	524.30 (1.06)	527.30 (1.01)	18.00 (0.03)	44.80
1024	2560	552.60	592.60 (1.07)	600.90 (1.01)	16.70 (0.03)	51.20
1024	3072	432.20	473.30 (1.10)	469.90 (0.99)	12.70 (0.03)	41.60
2048	2048	1955.50	2048.00 (1.05)	2048.00 (1.00)	1.00 (0.00)	2048.00
2048	3072	1164.30	1261.20 (1.08)	1331.30 (1.06)	28.10 (0.02)	121.60
2048	4096	879.50	982.20 (1.12)	976.40 (0.99)	34.10 (0.03)	60.80
2048	5120	1060.10	1206.90 (1.14)	1198.00 (0.99)	24.60 (0.02)	67.20
2048	6144	1039.50	1197.20 (1.15)	1188.70 (0.99)	22.40 (0.02)	60.80
4096	4096	3750.00	4096.00 (1.09)	4096.00 (1.00)	1.00 (0.00)	N/A
4096	6144	1959.50	2303.90 (1.18)	2287.70 (0.99)	202.90 (0.09)	N/A
4096	8192	1541.40	1913.70 (1.24)	1894.50 (0.99)	24.20 (0.01)	N/A
4096	10240	1792.60	2273.00 (1.27)	2275.40 (1.00)	21.00 (0.01)	N/A
4096	12288	1445.40	1974.90 (1.37)	2012.20 (1.02)	225.40 (0.11)	N/A
8192	8192	6995.90	8192.00 (1.17)	8192.00 (1.00)	1.00 (0.00)	N/A
8192	12288	3122.80	4274.90 (1.37)	4450.20 (1.04)	37.00 (0.01)	N/A
8192	16384	2576.10	3846.70 (1.49)	3817.50 (0.99)	32.00 (0.01)	N/A
8192	20480	2728.70	4209.30 (1.54)	4193.00 (1.00)	29.00 (0.01)	N/A
8192	24576	2342.90	3933.50 (1.68)	3905.00 (0.99)	29.20 (0.01)	N/A

TABLE IX

Burns', KO, YTO, Howard's, and HO algorithms for the number of iterations on the circuits.

Name	n	m	Burns	KO (KO/Burns)	YTO (YTO/KO)	Howard (Howard/KO)	HO
s382	273	438	110	111 (1.01)	109 (0.98)	15 (0.14)	91
s344	274	388	151	151 (1.00)	151 (1.00)	14 (0.09)	116
s349	278	395	130	131 (1.01)	130 (0.99)	32 (0.24)	116
s526n	292	560	171	171 (1.00)	171 (1.00)	22 (0.13)	147
mult16a	293	582	60	60 (1.00)	60 (1.00)	7 (0.12)	64
s400	287	462	131	131 (1.00)	129 (0.98)	17 (0.13)	90
s444	315	503	147	148 (1.01)	148 (1.00)	85 (0.57)	118
s526	318	576	178	178 (1.00)	178 (1.00)	33 (0.19)	152
mult16b	333	545	62	62 (1.00)	62 (1.00)	20 (0.32)	61
s641	477	612	160	160 (1.00)	160 (1.00)	292 (1.82)	64
s713	515	688	163	164 (1.01)	164 (1.00)	469 (2.86)	128
mult32a	565	1142	103	104 (1.01)	104 (1.00)	10 (0.10)	64
mm9a	631	1182	238	238 (1.00)	238 (1.00)	33 (0.14)	141
s838	665	941	189	189 (1.00)	188 (0.99)	39 (0.21)	198
s953	730	1090	134	138 (1.03)	137 (0.99)	16 (0.12)	32
mm9b	777	1452	434	438 (1.01)	438 (1.00)	36 (0.08)	109
s1423	916	1448	179	188 (1.05)	188 (1.00)	41 (0.22)	175
sbc	1147	1791	271	274 (1.01)	263 (0.96)	24 (0.09)	118
mm30a	2059	3912	624	641 (1.03)	639 (1.00)	61 (0.10)	128
s5378	3076	4590	1225	1333 (1.09)	1325 (0.99)	8 (0.01)	128
s9234	3083	4298	1328	1405 (1.06)	1394 (0.99)	100 (0.07)	497
bigkey	3661	12206	1400	1402 (1.00)	1394 (0.99)	316 (0.23)	896
dsip	4079	6602	883	934 (1.06)	929 (0.99)	453 (0.49)	512
s38584	20349	34563	1787	4302 (2.41)	4297 (1.00)	50 (0.01)	0
s38417	24255	34876	10532	11484 (1.09)	11464 (1.00)	1733 (0.15)	0