Faster Maximum and Minimum Mean Cycle Algorithms for System Performance Analysis

Ali Dasdan[†] and *Rajesh K. Gupta*[‡]

Technical Report #97-07

February 1997

[†]Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801
E-mail: dasdan@cs.uiuc.edu

[‡]Department of Information and Computer Science
University of California, Irvine, CA 92697
E-mail: rgupta@ics.uci.edu

Faster Maximum and Minimum Mean Cycle Algorithms for System Performance Analysis

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

Rajesh K. Gupta
Department of Information and Computer Science, 208B IERF
University of California
Irvine, CA 92697-3425
E-mail: rgupta@ics.uci.edu

Abstract

Maximum and minimum mean cycle problems are important problems with many applications in performance analysis of synchronous and asynchronous digital systems including rate analysis of embedded systems, in discrete-event systems, and in graph theory. Karp's algorithm is one of the fastest and commonest algorithms for both of these problems. We present this paper mainly in the context of the maximum mean cycle problem. We show that Karp's algorithm processes more vertices and arcs than needed to find the maximum cycle mean of a digraph. This observation motivated us to propose a new graph unfolding scheme that remedies this deficiency and leads to three faster algorithms with different characteristics. Asymptotic analysis tells us that our algorithms always run faster than Karp's algorithm. Experiments on benchmark graphs confirm this fact for most of the graphs. Like Karp's algorithm, they are also applicable to both the maximum and minimum mean cycle problems. Moreover, one of them is among the fastest to date.

I. Introduction

The average weight of a directed cycle is the total weight of the arcs on the cycle divided by the total number of the arcs on the cycle, and is called the *cycle mean*. The *maximum mean cycle* problem for a directed graph (digraph) with cycles is to find a cycle having the maximum average weight, called the *maximum cycle mean*, over all directed cycles in the graph. Such a cycle is called a critical cycle. The minimum mean cycle problem and the minimum cycle mean are defined analogously. Both of these problems have many important applications in performance analysis of digital systems, in discrete-event systems, and in graph theory. For instance, some of the uses in performance analysis that are more relevant to our study are as follows. The minimum mean cycle problem has applications in finding the cycle period in an asynchronous system [1], [2], and in data-flow partitioning of synchronous systems [3]; the maximum mean cycle problem has applications in finding the iteration bound of a data-flow graph for digital signal processing [4], in performance analysis of synchronous, asynchronous, or mixed systems [5], and in rate analysis for embedded systems [6].

The application in rate analysis for embedded systems is the reason that we have become interested in the maximum mean cycle problem. We have recently proposed a framework for an interactive analysis of rate constraints and debugging of their violations in an embedded system [6]. Rate constraints are often imposed by designers on the execution rate of each process in the system in order to ensure correct timing behavior and achieve performance goals. These constraints are usually placed under an overall view of the system and in an *ad hoc* manner. As the design goes along, it becomes more and more difficult for the system to conform to these constraints. Due to diverse interactions of the processes in the system with one another and the system's environment, synthesizing even one process can create constraint violations for all the other processes. Designers may need to make refinements as violations occur. However, spotting these violations and debugging them are tedious and difficult tasks due to the complexity and size of current designs, the number of refinements needed, and so on. Our framework automates this whole process. The framework includes modules to check the consistency of the imposed rate constraints, compute the execution rate of each process (rate analysis), compare the imposed and computed rates to check satisfiability or for possible constraint violations, output useful information to help the designer in case of violations. This useful information is usually in the form of critical cycles, which are a byproduct of a maximum mean cycle algorithm. The interactive nature of this framework requires it to be very fast. The most time consuming part is the rate analysis part, and the rate analysis part uses a maximum mean cycle algorithm. Even a small improvement in running time of the maximum mean cycle algorithm employed can pay off a lot as the time saved can accumulate to big gains because the number of refinements or iterations to carry out such a constraint-driven design of embedded systems is not known in advance and can potentially be large.

There are many algorithms proposed for the minimum mean cycle problem, e.g., [1], [7], [8], [9], [10], [11]. A more complete list is given in [7]. In this paper, we focus on Karp's algorithm [8] among them because of three reasons: 1) it has one of the best asymptotic running times, 2) it also works for the maximum mean cycle problem (as proved in [12]), and 3) it is usually the algorithm of choice [3], [4], [5]. Originally, Karp [8] gave a theorem (Karp's theorem) to characterize the minimum cycle mean in a digraph and an algorithm to compute it efficiently. Although the running time of Karp's algorithm is given in [8] as $O(nm)$ for a strongly connected digraph with n nodes and m arcs, its actual running time is $\Theta(nm)$, i.e., its best and worst case running times are the same, as also observed in [7], [11]. In this paper, Karp's theorem and

algorithm refer to their maximum mean cycle forms as defined in [12].

In this paper, we propose a graph unfolding scheme for the maximum mean cycle problem. This scheme is also applicable to the minimum mean cycle problem. The proposed scheme leads to three new algorithms. These algorithms have different characteristics in terms of actual running time, asymptotic running time, data structures, and implementation complexity. We evaluate these algorithms using asymptotic analysis and running experiments on benchmark graphs. Asymptotic analysis tells us that our algorithms are always faster than Karp's algorithm. Experimental results validates this fact for most of the test cases in practice. The main contribution of this paper is the application of our unfolding scheme to the maximum and minimum mean cycle problems and the algorithms it yields.

The rest of the paper is organized as follows. § II gives the necessary definitions and our notation. § III discussed Karp's theorem and Karp's algorithm in detail. Then, we motivate our approach on an example graph in § IV. Our unfolding scheme is discussed in detail in § V. We explain the details of our algorithms in § VI. Unfolding generates a graph called unfolded graph, and the running time of our algorithms depends on the size of that graph. § VII develops bounds on the size of the unfolded graph. Using these bounds, we work out the time and space complexity analysis of each algorithm in § VIII. We next discuss our experiments on benchmark graphs and their results in § IX. Finally, we conclude this paper in § X.

II. Definitions and Notation

Let $G = (V, E)$ be a digraph where the vertex set V has n vertices and the arc set E has m arcs. Each arc $e = (u, v)$ from vertex u to vertex v has a weight $w(e) = w(u, v)$. We allow G to have self-loops, i.e., arcs from and to the same vertex; thus, m can be equal to n^2. We use the adjacency-list representation of G with two arrays, $AdjIn$ and $AdjOut$, where each array has n lists, one for each vertex in V. The list $AdjIn[v]$ for vertex v contains all the predecessors of v, i.e., all those vertices u such that there is an arc (u, v) in E. Similarly, the list $AdjOut[v]$ for vertex v contains all the successors of v, i.e., all those vertices u such that there is an arc (v, u) in E.

A path of length k from vertex u to vertex u' is a sequence $< v_0, v_1, \ldots, v_k >$ of vertices such that $u = v_0$, $u' = v_k$, and $(v_{i-1}, v_i) \in E$ for $i = 1, 2, \ldots, k$. The weight $w(p)$ of path p is the sum of the weights of the arcs in p. A path $< v_0, v_1, \ldots, v_k >$ is a cycle if $v_0 = v_k$. The mean weight $\lambda(C)$ of a cycle C of length k is defined as $w(C)/k$. The mean weight of a cycle gives the average weight of each arc on the cycle. The *maximum cycle mean* λ^* of a digraph G is equal to $max_C\{\lambda(C)\}$ where C ranges over *all* directed cycles in G. The minimum cycle mean is defined analogously. A cycle whose mean weight equals the maximum cycle mean is called a *critical cycle*. We adopt the following conventions as in [7]: 1) the maximum over an empty set is $-\infty$; 2) $-\infty + w = -\infty$ for any w, 3) $-\infty + \infty = 0$. By the first convention, we assume that the input graphs in this paper have at least one arc. In our implementation, we use a very large real number to represent ∞, which explains why the third convention holds.

III. Karp's Theorem and Algorithm

Let G be a strongly connected digraph and s be an *arbitrary* vertex (the *source*). For every $v \in V$ and every nonnegative integer k, define $D_k(v)$ as the maximum weight of a path of length k from s to v; if no such path exists, then $D_k(v) = -\infty$. Then, the maximum cycle mean λ^* of G is given by the following theorem. Its proof can be found in [8], [12].

Theorem 1: (**Karp's Theorem**) The maximum cycle mean λ^* of G is given by

$$\lambda^* = \max_{v \in V} \min_{0 \le k \le n-1} \frac{D_n(v) - D_k(v)}{n-k}. \quad (1)$$

If G is not strongly connected, then we can find the maximum cycle mean by finding the strongly connected components of G (in linear time), determining the maximum cycle mean for each component, and then taking the largest of these as the maximum cycle mean of G. Unless stated otherwise, we consider only strongly connected graphs henceforth.

In [8], Karp's algorithm is given by the following recurrence for $D_k(v)$:

$$D_k(v) = \max_{(u,v) \in E} [D_{k-1}(u) + w(u,v)], \quad k = 1, 2, \ldots, n \quad (2)$$

with the initial conditions $D_0(s) = 0$ and $D_0(v) = -\infty$, $v \neq s$. This is also the form used in other works, e.g., [7], [13], [14]. We give the algorithm in Fig. 1, called Karp's algorithm, to compute λ^* based on this recurrence. The algorithm mainly fills the entries of an array of size $(n+1) \times n$, called the *table* and labeled D, such that $D[k, v] = D_k(v)$ for $k = 0, 1, \ldots, n$ and $v \in V$. We say that row k of D represents the kth *level* or level k.

For simplicity, we present this algorithm and our algorithms in three parts: *head*, *body*, and *tail*. These parts almost always realize the same goals for all the algorithms. Below we explain Karp's algorithm in detail. Many points will also apply to our algorithms.

The head initializes each table entry with $-\infty$ and determines the source. Note that $-\infty$ is both the identity for the max operation and is a flag to indicate the lack of a path. In case of Karp's algorithm, we select the first vertex of the vertex list as the source (denoted s) because the source can be an arbitrary vertex as guaranteed by Karp's theorem. Moreover, Karp's algorithm takes the same time regardless of the identity of the source. The length of the path from s to itself, $D[0, s]$, is set to 0, and the end of the predecessor list, $\pi[0, s]$, is set to NIL. We later use the array π to construct critical cycle(s).

The body finds $D_k(v)$ for each vertex $v \in V$ and each $k = 1, 2, \ldots, n$, and constructs the predecessor lists in π. The body basically uses the recurrence in Eq. 2. That is, for each vertex v, it checks every predecessor u of v (line 7), and computes $D_k(v)$ as $D_k(v) = \max\{D_k(v), D_{k-1}(u) + w(u,v)\}$ (lines 8-9). If $D_k(v)$ happens to be updated due to a predecessor u, u is also designated as the predecessor of v in the predecessor lists (line 10).

The tail finishes up the work by computing Eq. 1. For each vertex v, it computes the fraction in Eq. 1 for each $k = 0, 1, \ldots, n-1$, takes the minimum of all these fractions (corresponding to the min operation in Eq. 1), and stores the result in $M[v]$. The particular k that leads to the minimum in $M[v]$ is also recorded in $K[v]$, which will be useful for critical cycle construction. As for the max operation in Eq. 1, the tail checks $M[v]$ as they are computed and updates λ (lines 18-19). The λ of line 21 is the maximum cycle mean λ^*. The vertex that leads to λ^* is also stored in v^* for later reference in critical cycle construction.

IV. Motivation

We give the example digraph in Fig. 2 to illustrate Karp's algorithm as well as base our motivation on. The digraph in Fig. 2(a) is the input graph with 4 vertices and 5 arcs. The diagram in Fig. 2(b) presents how the algorithm works starting from the source s and gives the table entries. Each row (column) of circles corresponds to a row (column) of the table where each row is identified by an integer and each column by a vertex. The symbol ϵ represents $-\infty$. The numbers just to the right of each circle represent the values stored at the corresponding table

```
Input: A strongly connected digraph G = (V, E).
Output: λ* of G.
  /* Head */
1 for each level k, k = 0, 1, ..., n, do
2   for each vertex v ∈ V do
3     D[k, v] ← −∞
4 D[0, s] ← 0; π[0, s] ← NIL /* s is any vertex in V */

  /* Body */
5 for each level k, k = 1, 2, ..., n, do
6   for each vertex v ∈ V do
7     for each predecessor vertex u ∈ AdjIn[v] do
8       if (D[k, v] < D[k − 1, u] + w(u, v)) then
9         D[k, v] ← D[k − 1, u] + w(u, v) /* max */
10        π[k, v] ← u

  /* Tail */
11 λ ← −∞
12 for each vertex v ∈ V do
13   M[v] ← +∞ /* the identity for min */
14   for each level k, k = 0, 1, ..., n − 1, do
15     if (M[v] > (D[n, v] − D[k, v])/(n − k)) then
16       M[v] ← (D[n, v] − D[k, v])/(n − k) /* min */
17       K[v] ← k
18   if (λ < M[v]) then
19     λ ← M[v] /* max */
20     v* ← v
21 return λ
```

Fig. 1. Karp's maximum mean cycle algorithm.

entries, e.g., $D[2, c] = 9$ and $D[3, a] = -\infty$. There are two cycles in this graph: $< s, a, b, c, s >$ and $< s, b, c, s >$. As we have only two cycles, we can just use the definition of the maximum cycle mean instead of Eq. 1 to find the maximum cycle mean. The maximum cycle mean is then $\max\{(3+4+7+2)/4, (2+7+2)/3)\} = 4$, and $< s, a, b, c, s >$ is the critical cycle.

One important point is that the running times of Karp's algorithm and our algorithms are proportional to the number of vertices and the arcs visited during the process. As a result, we want to reduce their numbers. Karp's algorithm visits every arc (solid or dotted) in the diagram to fill the entries of the table. This is because at each level, Karp's algorithm has to check every predecessor of each vertex at that level (except for the 0th level). Thus, Karp's algorithm visits 20 arcs for this example, the total number of solid and dotted arcs. However, if we exclude all the dotted arcs, corresponding to the diagram in Fig. 2(c), the remaining (solid) arcs are enough to find the same maximum cycle mean. If we do so, we will visit only 9 arcs, which means more than 50% less work. Why is this so? Consider the recurrence given in Eq. 2. If $D_{k-1}(u) = -\infty$ for a predecessor u of v, then u cannot contribute to the computation of $D_k(v)$. In other words, if there is no path from s to u with $(k-1)$ arcs, then the path from s to v with k arcs, if exists, cannot pass through u. In particular, if $D_{k-1}(u) = -\infty$ for every predecessor u of v, there is no need even to touch any of these predecessors at all because $D_k(v) = -\infty$ by definition. These arguments prove that it does not necessary to check every predecessor of each vertex at every level. How can we ensure that? If we implement the recurrence above as it is, which makes v dependent on *all* of its predecessors and is the scheme that Karp's algorithm uses, we

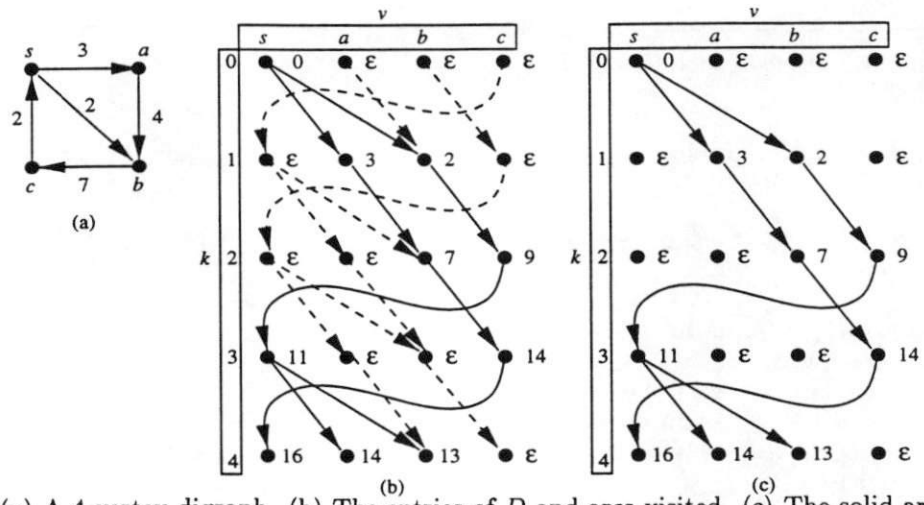

Fig. 2. (a) A 4-vertex digraph. (b) The entries of D and arcs visited. (c) The solid arcs of (b).

cannot eliminate unnecessary work. Instead, we propose the following *unfolding* scheme, which we formally define in the next section. Intuitively, it works as follows: starting from the source, visit in the next level the successors of all the vertices in the current level until the nth level is reached. In the case of the example above, visit the successors a and b of s at the 1st level, visit only the successors of a and b in the 2nd level, which are b and c, and so on.

V. Unfolding

Unfolding is the scheme that we propose to reduce the work required to find the maximum cycle mean. It is also applicable to the minimum mean cycle problem. This scheme is the basis of all our algorithms. The concept of unfolding is not new, e.g., see [13], [15], but our formulation of unfolding is. Moreover, to our knowledge, we are the first to use it for the maximum and minimum mean cycle problems.

Going from the kth level of a graph G to the $(k+1)$th level is called the $(k+1)$th *iteration* of G. This iteration visits all the arcs that originate from the kth level. We also say that the 0th level corresponds to the 0th iteration. Given a graph $G = (V, E)$, unfolding generates all the iterations from the 0th to the nth and also creates a new graph, called the *unfolded graph* and denoted $G_U = (V_U, E_U)$, in the process. The unfolded graph is a weighted graph with weighted vertices and arcs. Arc weights directly come from those in G but the vertex weights are computed during unfolding. Vertices in the unfolded graph are denoted as v^k, corresponding to vertex v in G, where k indicates that v^k is inserted into G_U in the kth iteration. Also note that v^k is at level k. The weight of vertex v^k in G_U is $D_k(v)$, and $D_k(v)$ is as defined in § III.

Unfolding proceeds by the following two rules for $k = 0, 1, \ldots, n-1$. Let $s \in V$ be the vertex designated as the source.

1. Initially, s^0 is in V_U, and $D_0(s) = 0$.
2. If $v^k \in V_U$ and $(v, u) \in E$, then $(v^k, u^{k+1}) \in E_U$ with $w(v^k, u^{k+1}) = w(v, u)$ and do either of the following:
   (a) if $u^{k+1} \notin V_U$, then $u^{k+1} \in V_U$ with $D_{k+1}(u) = D_k(v) + w(v, u)$, or
   (b) if $u^{k+1} \in V_U$, then $u^{k+1} \in V_U$ with $D_{k+1}(u) = \max\{D_{k+1}(u), D_k(v) + w(v, u)\}$.

The first rule above initializes the weight of u^{k+1} when it is inserted into V_U for the first time, and the second rule updates its weight if necessary. In our algorithms, we combine the two parts

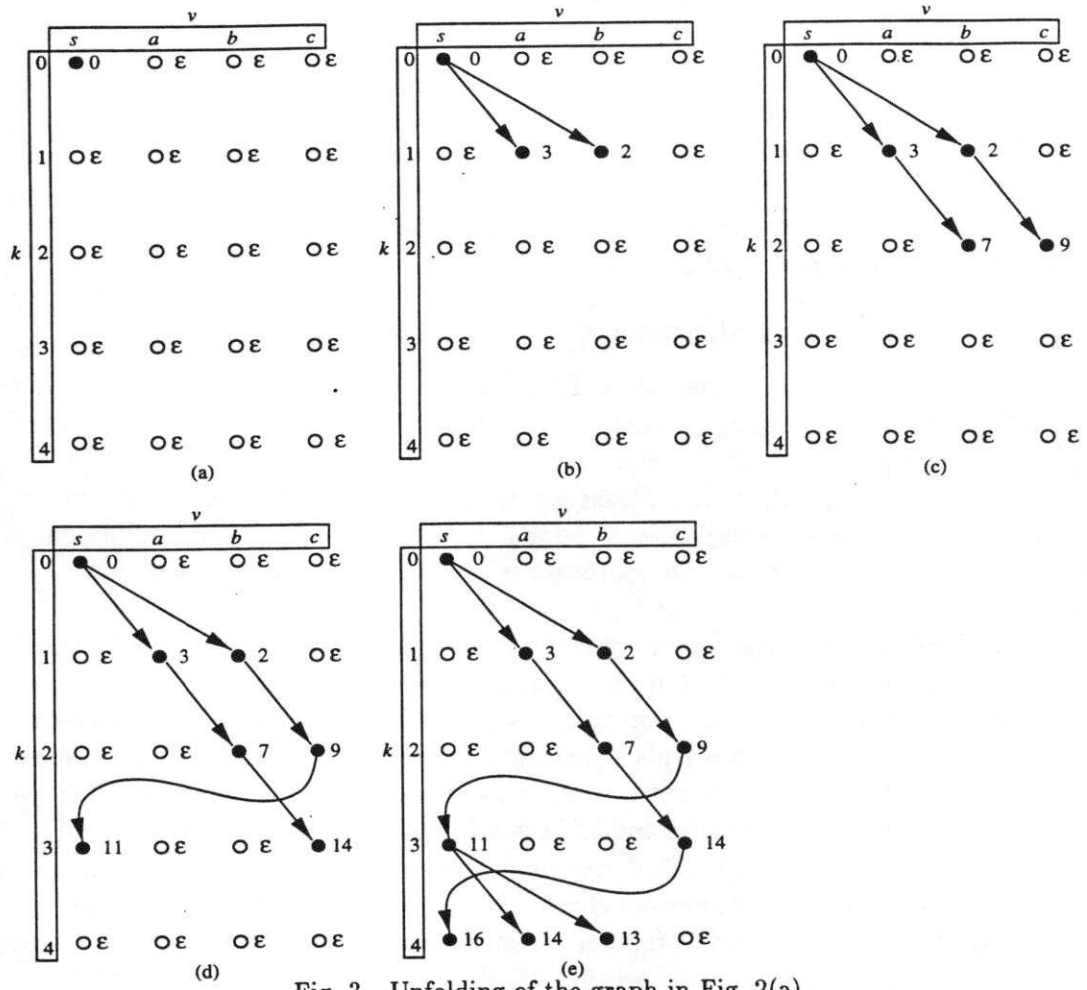

Fig. 3. Unfolding of the graph in Fig. 2(a).

of the second rule above into one by appropriate initialization, i.e., by initializing the vertex weights to $-\infty$ before processing them.

Fig. 3 illustrates the above rules on the example graph of Fig. 2(a). The iterations from 0 to 4 correspond to diagrams (a) to (c), respectively. Since we use the table D to implement unfolding, we present the iterations and the unfolded graph on the table. In this figure, each circle (empty or filled) represents an entry of D such that each entry has the value shown to the right of the corresponding circle. The filled circles and the arcs are the vertices and the arcs of the unfolded graph. We do not show the arc weights because they are readily available from Fig. 2(a); moreover, the vertex weights of the unfolded graph can easily be determined: the number right to each vertex is its weight. As Fig. 3(e) shows, the unfolded graph has 10 vertices and 9 arcs: a filled circle at the kth row and vth column corresponds to vertex v^k, and an arc between vertices v^k and u^{k+1} to arc (v^k, u^{k+1}) of the unfolded graph, i.e., $V_U = \{s^0, a^1, b^1, b^2, c^2, s^3, c^3, s^4, a^4, b^4\}$ and $E_U = \{(s^0, a^1), (s^0, b^1), (a^1, b^2), (b^1, c^2), (b^2, c^3), (c^2, s^3), (s^3, a^4), (s^3, b^4), (c^3, s^4)\}$.

Note that the unfolded graph is a finite, acyclic graph. It is finite because the total number of iterations required is n, and it is acyclic because it is impossible to create a back edge (one from a higher level to a lower one) hence a cycle by construction. This property alone implies that the vertex weights can be computed in polynomial time. Moreover, using this property, the argument in § IV, and induction on k, we can prove the correctness of unfolding. As such a proof

is quite easy, we just mention the following lemma without proof.

Lemma 1: Unfolding correctly computes $D_k(v)$ for each $v \in V$ and $k = 0, 1, \ldots, n$.

We have to mention one more property of unfolding as it will later have some implications for memory space allocated in two of our algorithms by constraining the total number of vertices in one iteration. The $(k+1)$st iteration of unfolding is between the levels k and $(k+1)$ only. It involves the vertices at both of these levels and the arcs from the kth to the $(k+1)$st but no more. Actually, Karp's algorithm has a similar property. Thus,

Lemma 2: Unfolding works on exactly two levels in one iteration.

VI. Faster Maximum Mean Cycle Algorithms

We now give three algorithms, named DG1, DG2, and DG3, to compute the maximum cycle mean in a strongly connected digraph. These algorithms are all based on unfolding. They are presented in decreasing order in terms of their asymptotic running times but in increasing order in terms of sophistication and their actual running times. Moreover, they are presented in the context of the maximum mean cycle problem. To apply these algorithms to the minimum mean cycle problem, we have to exchange min operation with max operation and $-\infty$ with $+\infty$, or vice versa.

In Karp's algorithm and our algorithms, the most time consuming part is usually the body. Also in our case, it is mainly the body that implements unfolding. The body of Karp's algorithm has very tight loops, and the innermost loop contains only one conditional and two assignments. We tried to make DG1 as similar to Karp's algorithm as possible to make use of that advantage of Karp's algorithm. DG2 is a more sophisticated implementation of unfolding to obtain better asymptotic time complexity in the body, and DG3 is a further improvement on DG2.

All of our algorithms are asymptotically faster than Karp's algorithm. One more advantage is that it does matter for our algorithms how we choose our source vertex. With a "suitable" source, the work done by the algorithms can be further reduced. We use a function $FindSource(G)$ to find such a source. This function is explained in § VI-D.

A. Our First Algorithm (DG1)

Our first algorithm, called DG1, is given in Fig. 4. DG1 has almost the same structure as Karp's algorithm. We tried to make the innermost loop in DG1 to contain as few statements as possible while still using unfolding. The use of unfolding is evident from the change of k from 0 to $n-1$ (line 7), and the use of the successor lists $AdjOut$ (line 11). Main differences from Karp's algorithm are the call to $FindSource(G)$ in the head to determine a suitable source, the use of $Visit$ and $turn$, and the elimination of some unnecessary work in the tail. $FindSource(G)$ is explained later in detail.

The need for $Visit$ and $turn$ can be explained as follows. DG1 still goes over every vertex at every level but eliminates unnecessary arc visits using $Visit$ and $turn$. Initially, $Visit[s, 0]$ is set to *true* to indicate that unfolding begins by visiting the source s. The flag $turn$ is used to differentiate between the two levels needed by Lemma 2 such that whatever the value of $turn$ is for the current level, its value for the next level is $1 - turn$. Suppose we are at level k with $turn$ set to 0. We could also have $turn$ set to 1 because it is more important that $turn$ toggles between levels rather than that $turn$ is 0 or 1. Then, if v is to be visited at this level, v must pass the test at line 9. Line 10 shows that v is being visited at level k. Since the successors of v are to be visited in the next level (level $(k+1)$) by the definition of unfolding, their $Visit$ fields are set to *true* at line 15.

The tail is almost the same as that of Karp's algorithm except for line 19. Line 19 guarantees that there will be no processing for vertex v if $D_n(v) = -\infty$, i.e., if there is no path of length n from s to v. The proof of the correctness of this argument can be stated using the definition of the D values and Karp's theorem. First, we know that there will be at least one path of length n from s to some vertex in G as we assume that G is strongly connected with at least one arc. This implies that the array M will have at least one real value. Now assume $D_n(v) = -\infty$ for some vertex v. Again since G is strongly connected, there is at least one real number in the column for v in D, and so the fraction in Karp's theorem takes on the value of $-\infty$ for v at least once. By noting that Karp's theorem first involves a minimization operation in the column for v in D, and also using the third convention of § II, we find that $M[v] = -\infty$. The fact that this value is the identity for the max operation and so does not affect the output of the max operation at lines 25-26 completes the proof.

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

```
   /* Head */
 1 for each level k, k = 0, 1, ..., n, do
 2   for each vertex v ∈ V do
 3     D[k, v] ← −∞
 4 Set Visit[v, 0] ← Visit[v, 1] ← false for ∀v ∈ V
 5 s ← FindSource(G); D[0, s] ← 0
 6 π[0, s] ← NIL; turn = 0; Visit[s, turn] ← true

   /* Body */
 7 for each level k, k = 0, 1, ..., n − 1 do
 8   for each vertex v ∈ V do
 9     if (Visit[v, turn] = true) then
10       Visit[v, turn] ← false
11       for each successor vertex u ∈ AdjOut[v] do
12         if (D[k + 1, u] < D[k, v] + w(v, u)) then
13           D[k + 1, u] ← D[k, v] + w(v, u) /* max */
14           π[k + 1, u] ← v
15           Visit[u, 1 − turn] ← true
16   turn ← 1 − turn

   /* Tail */
17 λ ← −∞
18 for each vertex v ∈ V do
19   if (Visit[v, turn] = true) then
20     M[v] ← +∞ /* the identity for min */
21     for each level k, k = 0, 1, ..., n − 1, do
22       if (M[v] > (D[n, v] − D[k, v])/(n − k)) then
23         M[v] ← (D[n, v] − D[k, v])/(n − k) /* min */
24         K[v] ← k
25     if (λ < M[v]) then
26       λ ← M[v] /* max */
27       v* ← v
28 return λ
```

Fig. 4. Our first maximum mean cycle algorithm DG1.

B. Our Second Algorithm (DG2)

Our second algorithm, called DG2, is given in Fig. 5. DG2 reveals the flavor of unfolding better because it touches on as many vertices and arcs as required by unfolding. We use a circular queue, denoted Q, to process the vertices of the unfolded graph. By Lemma 2, Q should have a size of $2n$ elements as there cannot be more than $2n$ elements in two successive levels. This renders unnecessary the need for a check to see if Q is full. There is also no need for a check to see if Q is empty because G has at least one arc by assumption.

The head is like that of DG1 but the tail is exactly the same as that of Karp's algorithm. The head and the body contain queue functions as appropriate. Each queue element contains a vertex v and the length of the path from s to v, which is also the level number at which v is visited. Line 9 eliminates duplicates from the queue. We used an array and standard queue functions to implement the queue.

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

```
   /* Head */
 1 for each level k, k = 0, 1, ..., n, do
 2   for each vertex v ∈ V do
 3     D[k, v] ← −∞
 4 s ← FindSource(G); D[0, s] ← 0
 5 π[0, s] ← NIL; Enqueue(Q, < 0, s >)

   /* Body */
 6 < k, v >← Dequeue(Q)
 7 do
 8   for each successor vertex u ∈ AdjOut[v] do
 9     if (D[k + 1, u] = −∞) then
10       Enqueue(Q, < k + 1, u >)
11     if (D[k + 1, u] < D[k, v] + w(v, u)) then
12       D[k + 1, u] ← D[k, v] + w(v, u) /* max */
13       π[k + 1, u] ← v
14   < k, v >← Dequeue(Q)
15 while (k < n)

   /* Tail */
16 λ ← −∞
17 for each vertex v ∈ V do
18   M[v] ← +∞ /* the identity for min */
19   for each level k, k = 0, 1, ..., n − 1, do
20     if (M[v] > (D[n, v] − D[k, v])/(n − k)) then
21       M[v] ← (D[n, v] − D[k, v])/(n − k) /* min */
22       K[v] ← k
23   if (λ < M[v]) then
24     λ ← M[v] /* max */
25     v* ← v
26 return λ
```

Fig. 5. Our second maximum mean cycle algorithm DG2.

C. Our Third Algorithm (DG3)

Our third algorithm, called DG3, is given in Fig. 6. DG3 is based on DG2, the main difference being that DG2 eliminates the initialization of D in the head. To do that, we noted that a vertex v may not be included at each iteration of unfolding, i.e., the entry $D[k, v]$ will stay at $-\infty$ for some k even after the last iteration. But, the tail must know which entries of the table D contain a valid entry, i.e., an entry that is not $-\infty$. The array $Valid$ points to valid entries: $Valid[k, v]$ gives the last level (obtained from array $LastLevel$) that is less than k and such that $D[Valid[k, v], v] \neq -\infty$. Only the valid entries are initialized (line 14). The entries of $Valid$ are like a linked list and the end of the list is signified by -1. The tail is also altered accordingly. The rest is self-explanatory. Note that we can also use linked lists instead of the table D because for sparse graphs, many of the table entries are probably not needed. We also implemented this extension but saw that the resulting algorithm was very slow although it had the lowest asymptotic space complexity.

D. Finding a Suitable Source

The source vertex mentioned in Karp's theorem is arbitrary. However, a different source may pay off a lot. For example, consider the same example given in Fig. 2. In (c), the unfolded graph contains 9 arcs when unfolding starts from s. Suppose instead we selected b as the source. Then, the unfolded graph would contain 6 arcs, yielding an improvement in the running time. Basically, by a suitable source, we mean a source vertex that leads to the "optimum" unfolding, by which we mean the one with the smallest unfolded graph in terms of the number of arcs. How can we find the optimum unfolding then? We can unfold the graph starting from each vertex once and select as the source the vertex leading to the least arcs. The problem with this approach is that it will take time probably larger than the running time of the whole process of finding the maximum cycle mean. Then, we should resort to a fast heuristic. The function $FindSource(G)$ given in Fig. 7 implements our heuristic. Our heuristic is to unfold the graph for a limited number, denoted N, of iterations, starting from each vertex, and returning as the source the vertex leading to the least number of arcs in these N iterations. We did experiments on our test suite by changing N from 1 to 30, and saw that $N = 10$ worked reasonably well, compromising between the running time and the solution quality. Note that the implementation of $FindSource(G)$ assumes that the degree of each vertex is available. Using these degrees and the arrays F and L (for First and Last), it counts the number of arcs that will result from unfolding the graph N times. The running time of $FindSource(G)$ is $\Theta(N(n + m))$, which is equal to $\Theta(m)$ as N is a constant and G is strongly connected.

E. Finding A Critical Cycle

We sometimes need to find a critical cycle as well as its mean, which is the case in the framework mentioned in § I. We noted that only [7], [8], and [14] give some information as to how to find a critical cycle. However, neither of them give it in an algorithmic form. Due to its importance, we present an algorithm in Fig. 8 to find one. Recall that in the tails of the algorithms above, we save the vertex in v^* that leads to the maximum cycle mean. The algorithm in Fig. 8 first constructs a path of length n from s to v^* using the information stored in array π and puts it into array P. Then, it finds the weight of each prefix of this path to facilitate finding the weight of any subpath of the path in P. Using array K of the k values recorded in the tails, we find the length of the critical cycle on P, which is $n - K[v^*]$. Finally, we go through each subpath of that length on P and check its weight to see if it is a critical cycle. We output the cycle as soon as we

```
Input: A strongly connected digraph G = (V, E).
Output: λ* of G.
  /* Head */
1 for each vertex v ∈ V do
2   LastLevel[v] ← −1
3 s ← FindSource(G)
4 D[0, s] ← 0; π[0, s] ← NIL
5 Valid[0, s] ← −1; LastLevel[s] ← 0
6 Enqueue(Q, < 0, s >)

  /* Body */
7 < k, v >← Dequeue(Q)
8 do
9   for each successor vertex u ∈ AdjOut[v] do
10    if (LastLevel[u] < k + 1) then
11      Enqueue(Q, < k + 1, u >)
12      Valid[k + 1, u] ← LastLevel[u]
13      LastLevel[u] ← k + 1
14      D[k + 1, u] ← −∞
15    if (D[k + 1, u] < D[k, v] + w(v, u)) then
16      D[k + 1, u] ← D[k, v] + w(v, u) /* max */
17      π[k + 1, u] ← v
18  < k, v >← Dequeue(Q)
19 while (k < n)

  /* Tail */
20 λ ← −∞
21 for each vertex v ∈ V do
22   if (LastLevel[v] = n) then
23     M[v] ← +∞ /* the identity for min */
24     k ← Valid[n, v]
25     while (k > −1) do
26       if (M[v] > (D[n, v] − D[k, v])/(n − k)) then
27         M[v] ← (D[n, v] − D[k, v])/(n − k) /* min */
28         K[v] ← k
29       k ← Valid[k, v]
30     if (λ < M[v]) then
31       λ ← M[v] /* max */
32       v* ← v
33 return λ
```

Fig. 6. Our third maximum mean cycle algorithm DG3.

find it. The running time of this algorithm is $\Theta(n)$ because it is dominated by loops and the other steps take constant time, including line 10 as $w(P[k-1], P[k]) = D[k, P[k]] - D[k-1, P[k-1]]$.

Although this algorithm finds only one critical cycle, we can modify it to find more. Recall that in the tails, we first compute the min of the columns of D into array M and then find the max of these minimum values. It is possible that there are more than one value among these minimum values that is equal to the maximum found. We can easily go through the M array entries and can determine all these minimum values. Then, we can run this algorithm for each vertex that has one of these minimum values instead of just for v^*. Moreover, the path stored into P may contain more than one critical cycle, and so we can continue checking after outputting each critical cycle found.

```
Input: A strongly connected digraph G = (V, E).
Output: A source vertex s.
1 for each vertex v ∈ V do
2   F(v) ← 0
3 for each iteration i = 1, ..., N do
4   for each vertex v ∈ V do
5     L(v) ← d(v)
6     for each successor vertex u ∈ AdjOut(v) do
7       L(v) ← L(v) + F(u)
8   for each vertex v ∈ V do
9     F(v) ← L(v)
10 Find s such that L(s) = min_{v∈V}{L(v)}
11 return s
```

Fig. 7. Our algorithm to find a suitable source vertex.

```
Input: λ*, v*, and array π.
Output: A critical cycle of G.
1 k ← n
2 P[k] ← v*
3 while (k > 0) do
4   P[k − 1] ← π[k, P[k]]
5   k ← k − 1
  /* P now has a path from s to v*.*/
6 k ← 0
7 W[k] ← 0
8 while (k < n) do
9   k ← k + 1
10  W[k] ← W[k − 1] + w(P[k − 1], P[k])
  /* W now has all prefix path costs on P.*/
11 len ← n − K[v*]
12 k ← 0
13 while (k ≤ K[v*]) do
14   if ((P[k] = P[k + len]) and (W[k + len] − W[k] = λ*)) then
15     return P[k] ··· P[k + len]
16   k ← k + 1
```

Fig. 8. Algorithm to find a critical cycle.

VII. Bounding the Size of Unfolded Graph

In this section, we derive bounds on the size of the unfolded graph. These bounds are also instrumental in understanding the improvements we have obtained in the running time. Let $t(v)$, $c(v)$ and $d(v)$ be the number of times an arc originates from vertex v (except at the 0th level), the number of levels (except the 0th level) at which v is included, and the out-degree of v, respectively. Consider a strongly connected graph $G = (V, E)$ with n vertices and m arcs that may also include self-loops. Then, we have the following equations for the unfolded graph $G_U = (V_U, E_U)$:

$$|E_U| = d(s) + \sum_{v \in V} d(v)t(v), \text{ and } |V_U| = 1 + \sum_{v \in V} c(v) \tag{3}$$

where the first term in each equation is for the source vertex at the 0th level. Note the special relationship between $t(v)$ and $c(v)$: if v is included at the nth level, then $t(v) = c(v) - 1$, or else

$t(v) = c(v)$.

Since we allow self-loops, we have $1 \leq d(v) \leq n$. Also, we have n levels in total when we exclude the 0th level. Thus, an arc can originate from vertex v if v is included at any or all of the levels from 1 to $n-1$, i.e., $1 \leq t(v) \leq (n-1)$, and v can be included at any or all of the levels from 1 to n, i.e., $1 \leq c(v) \leq n$. Inclusion at successive levels probably stems from self-loops. Plugging these values into Eq. 3, we can bound the size of the unfolded graph in terms of the parameters of G as

$$m \leq |E_U| \leq (n-1)m + n, \text{ and } (1+n) \leq |V_U| \leq (1+n^2), \tag{4}$$

where $(n-1)m+n \leq nm$ if $n \leq m$, which is the case when G is strongly connected. The smallest unfolded graph occurs when G is a ring with n vertices and m arcs, and the largest one occurs when G is a complete graph with self-loops on every vertex.

VIII. Complexity Analyses

For space complexity analysis, we note that the arrays used in the algorithms consume much of the space. Every algorithm uses D, an array of size $(n+1) \times n$. The other arrays are at most as large as D. Hence, we can say that the space usage is dominated by D, and all the algorithms have the same space complexity of $\Theta(n^2)$.

For time complexity analysis, we will examine each algorithm separately. For Karp's algorithm, our analysis is fairly straightforward and proceeds as follows. The running time is dominated by the nested **for** loops in each part: The head runs in $\Theta(n^2)$; the body goes through each predecessor of each vertex at every level, so runs in $\Theta(nm)$; finally, the tail visits each table entry, and so runs in $\Theta(n^2)$. The total running time comes to $\Theta(n^2 + nm)$, which is $\Theta(nm)$ as we assume that the input is a strongly connected graph.

As for DG1, we have the following. The head initializes the table and array $Visit$, which take $\Theta(n^2)$ time and $\Theta(n)$ time respectively. $FindSource$ runs in $\Theta(m)$, so the head runs in $\Theta(n^2)$ time in total. The body checks every vertex at every level but visits the arcs of the unfolded graph only. So, it runs in $\Theta(n^2 + |E_U|)$. The tail runs in $\Theta(n^2)$ in the worst case but can run in less time due to line 19, e.g., if the graph is a ring, the tail takes linear time. Hence, the tail runs in $O(n^2)$ time, and the total running time for DG1 is $\Theta(n^2 + |E_U|)$.

As for DG2, the head (even with $FindSource$) and the tail have the same running time as those in Karp's algorithm. Thus, they both run in $\Theta(n^2)$ time. The body visits only the vertices and the arcs in the unfolded graph and runs in $\Theta(|E_U|)$. Hence, the total running time for DG2 is $\Theta(n^2 + |E_U|)$. Here also note that the queue functions run in constant time.

As for DG3, the head mainly has a **for** loop to initialize $LastLevel$ and $FindSource$, so runs in $\Theta(m)$ due to the latter. The body, similar to that of DG2, visits only the vertices and the arcs in the unfolded graph and runs in $\Theta(|E_U|)$. The tail has an interesting behavior. Due to the use of array $Valid$, the tail checks only the vertices in the unfolded graph. Also, due to line 22, it runs in $O(|V_U|)$ time. Thus, the total running time for DG3 is $O(|V_U| + |E_U|)$, which seems to be the best we can get given the size of the unfolded graph.

A summary of the running times is given in Tab. I and in the following theorem. We conclude from these analyses that our algorithms have the same running time as Karp's algorithm *only* in the worst case; for all the other (and common) cases, our algorithms outperform Karp's algorithm asymptotically. The running time difference can be significant, e.g., DG3 requires $\Theta(n)$ time for a ring graph but Karp's algorithm requires $\Theta(n^2)$.

TABLE I

TIME COMPLEXITY OF EACH ALGORITHM IN Θ-NOTATION UNLESS NOTED OTHERWISE.

Alg.	Head	Body	Tail	Total
Karp	n^2	nm	n^2	$n^2 + nm$
DG1	n^2	$n^2 + \|E_U\|$	$O(n^2)$	$n^2 + \|E_U\|$
DG2	n^2	$\|E_U\|$	n^2	$n^2 + \|E_U\|$
DG3	m	$\|E_U\|$	$O(\|V_U\|)$	$O(\|V_U\| + \|E_U\|)$

Theorem 2: For a strongly connected graph with n vertices and m arcs, Karp's algorithm runs in $\Theta(nm)$ time, and our algorithms run in $O(nm)$ time. For a non-strongly connected graph with the same parameters, all of the algorithms run in $O(nm)$ time.

IX. EXPERIMENTAL RESULTS AND DISCUSSION

In this section, we report the results of our experiments that we did to see the performance of our algorithms in practice. We coded all the algorithms in `C`, and did experiments on a `Sun SPARC 20` with 64 MB real memory, running `Sun OS R.5.4`. We used two groups of benchmark graphs as our test suite[1]. The first group of graphs were obtained from the high-level synthesis benchmarks. The second group consists of signal transition graphs used in [16], [17].

Tab. II presents the size of each test case and the experimental results in terms of the number of vertices and arcs visited and the running time. In the table, the first group consists of the first 8 graphs (whose names are capitalized), and the second group consists of the rest, 42 graphs, of the graphs (whose names are all in lower case). We included an extra large ring graph at row 9, R2000, to show the performance in the best case for our algorithms. Rings graphs also occur naturally, e.g., mod4-counter in the second group. The numbers reported are for the whole graphs, i.e., they are for all the strongly connected components. The running time is the sum of the user and system CPU times (in seconds) from the beginning of execution of the algorithm to its completion.

From Tab. II, we can observe the following. Each observation is also discussed below.

1. DG1 and Karp's algorithm (column 4) visit the same number of vertices but DG2 and DG3 (column 5) visit a lot less than both of them. The improvement achieved ranges from 99.95% on R2000 to 13.32% on MC6502-GROUP0. The improvement is more than 50% on 43 out of 52 graphs in the table.
2. All our algorithms (column 7) visit a lot less arcs than Karp's algorithm (column 6) does. The improvement achieved ranges from 99.95% on R2000 to 13.47% on MC6502-GROUP0. The improvement is more than 50% on 43 out of 52 graphs in the table.
3. Our algorithms visit as many arcs as should be visited to get the result. For example, since there is only one cycle in a ring graph (R2000), the maximum cycle mean can be found by computing the mean weight of that cycle in time linear in the size of the graph (excluding the complexity of the head and tail). Karp's algorithm instead takes time quadratic in the size of the graphs. The behaviors of the algorithms are evident from row 9 of Tab. II. DG3 is the fastest because it does as much work as required. DG1 and DG2 visit as many arcs as there are in R2000 but the times spent during head and tail take most of the running time. Note the number of vertices and arcs Karp's algorithm visits, which is why it is the slowest on this simple graph. This weakness of Karp's algorithm is usually expected to be seen in sparse graphs.

[1]The program and the test suite are available from the authors.

4. DG1 is almost always the fastest algorithm among the three (columns 8-11). For only three of the graphs (in the second group), Karp's algorithm runs faster than DG1. We think that this might have resulted from the resulotion of the timer we used.[2]
5. Karp's algorithm sometimes outperforms DG2 and DG3 especially on graphs in the first group. As DG2 and DG3 do much less work, this observation seems a bit strange. However, the reason is simple. As we mentioned earlier, the body is the most time consuming part of these algorithms, and Karp's algorithm has a body with very tight, small loops. This is actually the biggest advantage of Karp's algorithm. The innermost loop in DG2 and DG3 has to include more statements to realize unfolding. This together with possible cache misses due to the circular queue seems to be the reason for this behavior.

We should mention that the number of vertices and arcs visited is independent of implementation, so is a better indicator of the improvements that our algorithms achieve. This fact should be observed especially for the benchmarks in the second group because the running time figures for them are very small. In summary, DG3 is asymptotically the fastest algorithm to the extent that it is one of the fastest in the literature; however, we think that DG1 is the best choice, considering the behavior in practice. The implementation complexity of DG1 is also no worse than that of Karp's algorithm.

X. Conclusions

In this paper, we discussed the maximum mean cycle problem and an algorithm, Karp's algorithm, that has been commonly used to solve it. We showed the shortcomings of Karp's algorithm and proposed a graph unfolding scheme for the maximum mean cycle problem to remedy them. The proposed scheme leads to three faster maximum mean cycle algorithms with different characteristics. We evaluated these algorithms both in theory using asymptotic analysis and in practice using experiments on benchmark graphs in comparison with Karp's algorithm. Asymptotic analysis shows that our algorithms are always faster than Karp's algorithm. Experimental results give a somewhat different picture. One of our algorithms is always faster than Karp's algorithm, but the other two are sometimes slower. We concluded that this is due to the very tight loops in Karp's algorithm, which probably makes the constants in its asymptotic running time smaller. Experiments also show that our algorithms process a lot less vertices and arcs than Karp's algorithm does, as we expected from the asymptotic analysis. This observation, unlike that from actual running times, is more promising because it is independent of any implementation of these algorithms. We also have to mention that our improvements are also applicable to the minimum mean cycle problem and that one of our algorithms is one of the fastest to date.

Acknowledgment. We would like to thank J. Cortadella and L. Lavagno for sending us the signal transition graphs. This work was supported by the National Science Foundation under grants NSF Career Award MIP 95-01615 and ASC-96-34947.

References

[1] S. M. Burns and A. J. Martin, "Performance analysis and optimization of asynchronous circuits", in *Proc. Advanced Research in VLSI Signal Processing Conf.*, 1991, pp. 71–86.

[2] J. Magott, "Performance evaluation of concurrent systems using petri nets", *Infor. Proc. Letters*, vol. 18, pp. 7–13, 1984.

[3] L.-T. Liu, M. Shih, and C.-K. Cheng, "Data flow partitioning for clock period and latency minimization", in *Proc. 31st Design Automation Conf.* ACM/IEEE, 1994, pp. 244–249.

[2] We used the `times` system call.

[4] K. Ito and K. K. Parhi, "Determining the minimum iteration period of an algorithm", *J. VLSI Signal Processing*, vol. 11, no. 3, pp. 229–244, Dec. 1995.

[5] J. Teich, S. Sriram, L. Thiele, and M. Martin, "Performane analysis of mixed asynchronous synchronous systems", in *Proc. Workshop on VLSI Signal Processing*. IEEE, 1994.

[6] A. Dasdan, A. Mathur, and R. K. Gupta, "Ratan: A tool for rate analysis and rate constraint debugging for embedded systems", in *Proc. Euro. Design and Test Conf.* IEEE, 1997.

[7] M. Hartmann and J. B. Orlin, "Finding minimum cost to time ratio cycles with small integral transit times", *Networks*, vol. 23, pp. 567–574, 1993.

[8] R. M. Karp, "A characterization of the minimum cycle mean in a digraph", *Discrete Mathematics*, vol. 23, pp. 309–311, 1978.

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

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

[11] N. E. Young, R. E. Tarjan, and J. B. Orlin, "Faster parametric shortest path and minimum-balance algorithms", *Networks*, vol. 21, pp. 205–221, 1991.

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

[13] L. E. Lucke, A. P. Brown, and K. K. Parhi, "Unfolding and retiming for high-level DSP synthesis", in *Proc. Int. Symp. Circuits and Syst.* IEEE, 1991.

[14] G. Cohen, D. Dubois, J. P. Quadrat, and M. Viot, "A linear-system-theoretic view of discrete-event processes and its use for performance evaluation in manufacturing", *IEEE Tran. Automatic Control*, vol. 30, no. 3, pp. 210–220, Mar. 1985.

[15] H. Hulgaard, S. M. Burns, T. Amon, and G. Borriello, "An algorithm for exact bounds on the time seperation of events in concurrent systems", *IEEE Trans. Comput.*, vol. 44, no. 11, pp. 1306–1317, Nov. 1995.

[16] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev, "Methodology and tools for state encoding in asynchronous circuit synthesis", in *Proc. 33rd Design Automation Conf.* ACM/IEEE, 1996, pp. 63–66.

[17] L. Lavagno, K. Keutzer, and A. Sangiovanni-Vincentelli, "Synthesis of hazard-free asynchronous circuits with bounded wire delays", *IEEE Trans. Computer-Aided Design*, vol. 14, no. 1, pp. 61–86, Jan. 1995.

TABLE II

BENCHMARKS AND EXPERIMENTAL RESULTS.

Name	Size		# vertices visited		# arcs visited		Running times (s)			
			Karp DG1	DG2 DG3						
	n	m			Karp	DG[123]	Karp	DG1	DG2	DG3
DIFFEQ	172	204	4740	2439	11780	6035	0.01	0.01	0.02	0.01
ELLIPTIC	307	344	9049	4719	26602	13324	0.02	0.01	0.02	0.02
GCD	49	55	449	59	465	37	0.00	0.00	0.01	0.00
MC6502-GROUP0	2219	2410	106825	92599	583823	505207	0.37	0.33	0.56	0.66
MC6502-GROUP1	620	671	17912	13684	68508	52375	0.04	0.04	0.07	0.07
MC6502-GROUP2	2928	3198	233360	178447	960221	754319	0.65	0.56	0.97	1.21
PARKER1986	172	177	8004	5830	14863	11005	0.02	0.02	0.02	0.02
TSENG	136	161	406	136	360	34	0.00	0.00	0.00	0.00
R2000	2000	2000	4000000	2000	4000000	2000	8.77	2.25	5.69	0.62
adfast	12	15	144	18	180	23	0.00	0.00	0.00	0.00
alloc-outbound	24	25	576	32	600	33	0.00	0.00	0.01	0.00
chu133	14	17	196	23	238	26	0.00	0.00	0.00	0.00
chu150	14	16	196	31	224	36	0.00	0.00	0.00	0.00
chu172	15	16	225	16	240	17	0.00	0.00	0.00	0.00
combuf2	16	19	172	27	195	29	0.00	0.00	0.00	0.00
converta	14	16	196	26	224	29	0.00	0.00	0.00	0.00
ebergen	14	16	196	26	224	29	0.00	0.00	0.00	0.00
full	8	12	64	14	96	21	0.00	0.00	0.00	0.00
hazard	10	12	100	12	120	14	0.00	0.00	0.00	0.00
hybridf	16	26	256	86	416	142	0.00	0.00	0.00	0.00
ircv-bm	51	64	2601	230	3264	289	0.01	0.01	0.01	0.01
master-read	28	40	784	425	1120	609	0.00	0.00	0.00	0.00
mmu	16	20	256	55	320	68	0.00	0.00	0.00	0.00
mmu0	16	20	256	55	320	68	0.00	0.00	0.00	0.00
mmu1	16	24	256	127	384	187	0.00	0.00	0.00	0.00
mod4-counter	16	16	256	16	256	16	0.00	0.00	0.01	0.00
mp-forward-pkt	16	26	256	25	416	40	0.00	0.00	0.00	0.00
mr0	22	31	484	168	682	229	0.00	0.00	0.00	0.00
mr1	18	25	324	114	450	155	0.00	0.00	0.00	0.00
mux2	53	74	2809	1947	3922	2744	0.00	0.01	0.01	0.01
nak-pa	20	24	400	26	480	30	0.00	0.00	0.01	0.00
nowick	16	21	256	20	336	25	0.00	0.00	0.00	0.00
par-4	20	23	400	44	460	50	0.00	0.00	0.01	0.00
pe-rcv-ifc	63	81	3969	991	5103	1247	0.01	0.00	0.00	0.01
pe-send-ifc	62	83	3844	1066	5146	1464	0.01	0.01	0.00	0.00
postoffice	45	93	2025	1550	4185	3169	0.01	0.01	0.00	0.01
qr42	14	16	196	26	224	29	0.01	0.00	0.00	0.00
ram-read-sbuf	22	28	484	35	616	43	0.00	0.00	0.00	0.00
rcv-setup	17	20	289	57	340	70	0.01	0.00	0.00	0.00
rpdft	22	22	484	22	484	22	0.00	0.00	0.01	0.00
sbuf-ram-write	24	29	576	42	696	51	0.00	0.01	0.00	0.00
sbuf-read-ctl	16	19	256	19	304	22	0.00	0.00	0.00	0.01
sbuf-send-ctl	26	31	676	42	806	49	0.00	0.00	0.00	0.01
sbuf-send-pkt2	30	35	900	201	1050	237	0.00	0.00	0.00	0.00
sendr-done	8	9	64	9	72	10	0.00	0.00	0.00	0.00
seq8	36	36	1296	36	1296	36	0.01	0.00	0.00	0.00
seq-mix	21	21	441	21	441	21	0.00	0.01	0.00	0.00
spec-seq4	20	20	400	20	400	20	0.00	0.00	0.00	0.01
trcv-bm	49	62	2401	224	3038	284	0.01	0.00	0.01	0.00
tsend-bm	44	54	1936	181	2376	222	0.00	0.00	0.00	0.00
vme2int	22	28	484	106	616	135	0.00	0.00	0.00	0.00
wrdatab	24	33	576	269	792	375	0.00	0.00	0.00	0.00