A Strongly Polynomial-Time Algorithm for Over-Constraint Resolution

[Efficient Debugging of Timing Constraint Violations]

Ali Dasdan
Advanced Technology Group
Synopsys, Inc.
700 E. Middlefield Rd., Mountain View, CA 94043, U.S.A.
dasdan@synopsys.com

ABSTRACT

A system of binary linear constraints or difference constraints (SDC) contains a set of variables that are constrained by a set of unary or binary linear inequalities. In such diverse applications as scheduling, interface timing verification, real-time systems, multimedia systems, layout compaction, and constraint satisfaction, SDCs have successfully been used to model systems of both temporal and spatial constraints. Formally, SDCs are modeled by weighted, directed (constraint) graphs. The consistency of an SDC means that there is at least one instantiation of its variables that satisfies all its constraints. It is well known that the absence of positive cycles in a graph implies the consistency of the corresponding SDC, so the consistency can be decided in strongly polynomial time. If a SDC is found to be inconsistent, it has to be repaired to make it consistent. This task is equivalent to removing positive cycles from the corresponding graph. All the previous algorithms for this task take time proportional to the number of positive cycles in the graph, which can grow exponentially. In this paper, we propose a strongly polynomial-time algorithm, i.e., an algorithm whose time complexity is polynomial in the size of the graph. Our algorithm takes in a graph and returns a list of edges and the changes in their weights to remove all the positive cycles from the graph. We experimentally quantify the length of the edge list and the running time of the algorithm on large benchmark graphs. We show that both are very small, so our algorithm is practical.

Keywords

Behavioral synthesis, high-level synthesis, scheduling, timing constraints, rate analysis, constraint satisfaction.

1. INTRODUCTION

A special case of the general linear-programming problem is the system of difference constraints (SDC). In such a system, each constraint is a binary linear inequality defined over two variables. As such, this type of a system is mapped to a weighted, directed graph (called a constraint graph) where vertices correspond to the variables and the edges correspond to the constraints. A solution to a given SDC means finding an instantiation of the variables that satisfies all the constraints. If at least one solution exists, the SDC is consistent; otherwise, it is inconsistent. If the SDC is inconsistent, it is also said to be over-constrained.

Once the constraint graph for an SDC is constructed in a certain way, it can be shown that the inconsistency of an SDC is tied to the presence of positive cycles in the graph. Thus, a simple single-source longest-paths algorithm is all that it takes to verify the inconsistency of the SDC. However, if a solution is needed, the SDC has to be repaired, or the inconsistency has to be resolved. Simply, converting every positive cycle in the graph to a zero cycle will make the SDC consistent. One way or another, most of the approaches in the literature are based on this intuition. The problem here is, of course, that the graph can have an exponential number of positive cycles, resulting in exponential time algorithms to gain consistency. The main contribution of this paper is to propose an intelligent algorithm that not only restores consistency correctly (by outputting a list of edges whose weights have to be changed) but also performs this task in time that is a polynomial function of the number of vertices and the number of edges in the input graph, resulting in a strongly polynomial-time algorithm.

The reason why we should care about such an algorithm is the wide applicability of SDCs. Below we list only a few applications without being exhaustive:

- scheduling, e.g., operation scheduling in behavioral synthesis [20, 23, 28, 32], software scheduling [8, 22], job / task scheduling [9], or data-flow scheduling [31];
- interface timing verification [3, 4];
- rate analysis [10, 25];
- real-time systems, e.g., real-time scheduling [27], run-time monitoring timing constraints [29], or safety analysis of real-time systems [18];
- modeling most of the temporal, spatial, and quality-of-service requirements for collaborative multimedia systems [5];
- layout compaction [14, 17, 19, 21, 26, 30]; and
- constraint satisfaction problems [13].

Note that the constraints in an SDC can be used to model temporal, spatial, and possibly other kinds of constraints as long as

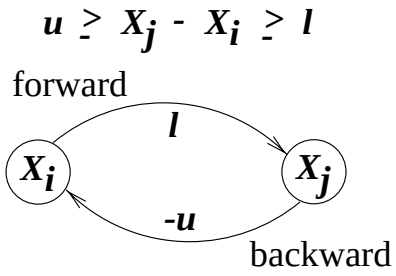

Figure 1: Modeling a minimum constraints $X_j - X_i \geq l$ with a forward edge and a maximum constraint $u \geq X_j - X_i$ with a backward edges.

they result in unary or binary linear inequalities. For example, for scheduling, all the constraints are timing constraints (derived from data / control flow ordering and user constraints) whereas for layout compaction, all the constraints are spatial constraints (derived from technology design rules and user constraints). Our interest in this paper is on timing constraints.

2. PROBLEM FORMULATION

We now present the formal definition of an SDC (§ 2.1), the modeling it using a constraint graph (§ 2.2), the correspondence between the inconsistency of the system and the presence of positive cycles in the constraint graph (§ 2.3), the problem of resolving inconsistency (§ 2.4), and a list of previous approaches to this problem (§ 2.4). The section is largely based on the references mentioned in the introduction section.

2.1 Systems of Difference Constraints

Definition 1. A system of difference constraints (SDC) consists of n unknowns (or variables) $X_1, X_2, \ldots, X_n$ and m difference constraints, in which each difference constraint is a *binary linear inequality* of the (canonical) form $X_j - X_i \geq b_k$, where $1 \leq i, j \leq n$, $1 \leq k \leq m$, and b_k are constants.

Definition 2. Given an SDC with n variables and m constraints, an n-tuple $x = (x_1, x_2, \ldots, x_n)$ is a *(feasible) solution* if the assignment (or instantiation) $\{X_1 = x_1, X_2 = x_2, \ldots, X_n = x_n\}$ *satisfies* all the constraints. Each constraint that is not satisfied is said to be *violated*. The system is *consistent* if at least one solution exists, and is *inconsistent* otherwise.

2.2 Modeling Using Constraint Graphs

Definition 3. Given an SDC with n variables and m constraints, the corresponding *constraint graph* is a weighted, directed graph $G = (V, E, w)$ with $|V| = (n+1)$ vertices, $|E| = (m+n)$ edges, and an edge weight function w, in which $V = \{X_0, X_1, \ldots, X_n\}$ and $E = \{(X_i, X_j) : w(X_i, X_j) = b_k\} \cup \{(X_0, X_j) : w(X_0, X_j) = 0\}$ for $1 \leq i, j \leq n$. Every constraint of the form $X_j - X_i \geq b_k$ is converted to an edge (X_i, X_j) with weight $w(X_i, X_j) = b_k$.

Note that G contains a vertex for each variable and an extra vertex X_0, which is the reference or source vertex and is connected to every other vertex via zero(-weighted) edges. (An edge or a cycle is positive, negative, or zero based on the sign of its weight). For simplicity, we assign zero to X_0 in any solution.

We will use the notations $SDC[G]$ and $G[SDC]$ to denote the SDC corresponding to G and vice versa, respectively. An SDC as modeled by a constraint graph is general enough to model any unary and binary linear constraints, which can contain either $\leq$ or $\geq$. That is, any such constraint can be converted to the canonical form, as shown in the following examples for two variables X_i and X_j:

- $X_j \geq 5 \Leftrightarrow X_j - X_0 \geq 5$;
- $X_j \leq 5 \Leftrightarrow X_0 - X_j \geq -5$;
- $X_j - X_i \leq 5 \Leftrightarrow X_i - X_j \geq -5$; and
- $X_j - X_i = 5 \Leftrightarrow X_j - X_i \geq 5$ and $X_i - X_j \geq -5$.

Therefore, without loss of generality, we will consider binary constraints in the canonical form.

Definition 4. Consider an SDC and $G[SDC]$. A constraint of the form $X_j - X_i \geq b_k$ is called a *minimum constraint* as it provides a *lower bound* b_k for the difference $X_j - X_i$. The corresponding edge (X_i, X_j) with weight b_k is called a *forward edge*. A constraint of the form $X_j - X_i \leq b_k$ is called a *maximum constraint* as it provides a *upper bound* b_k for the difference $X_j - X_i$. This constraint is equivalent to $X_i - X_j \geq -b_k$ in the canonical form. The corresponding edge (X_j, X_i) with weight $-b_k$ is called a *backward edge*.

Fig. 1 depicts modeling binary constraints with forward and backward edges.

2.3 Characterizing Consistency Using Positive Cycles

THEOREM 1. *(e.g., [9]) Consider an SDC and $G[SDC] = (V, E, w)$. Then, G is consistent if and only if G contains no positive cycles. Furthermore, if G is consistent, then*

$$x = (0, d(X_0, X_1), \ldots, d(X_0, X_n)) \quad (1)$$

is a solution where $d(X_0, X_i)$ is the length of the longest path from X_0 to X_i for $1 \leq i \leq n$. This solution is also the smallest one in that it minimizes $X_i - X_0$ for each X_i.

If we reverse every edge in G and negate their weights, then the consistency of $SDC[G]$ is guaranteed in the absence of any negative cycles [9]. Thus, without loss of generality, we will deal with positive cycles.

To detect the presence of positive cycles and find a solution in their absence, we can use a single-source longest-path algorithm. In theory, the fastest time bound for such algorithms is $O(nm)$, which is achieved by the well-known Bellman-Ford algorithm. In practice, faster algorithms exist [7]. Especially, if it is more likely that G will have a positive cycle, even faster algorithms, e.g., Tarjan's algorithm, exist [6]. One algorithm that is not reported in [6, 7] is the Liao-Wong algorithm [21]. Its time complexity is $O((b+1)m)$ where $b = O(n)$ is the number of the backward edges in G. As such, this algorithm can be more suitable if b is small.

Now that we know how to find a solution when $SDC[G]$ is consistent, we next focus on resolving inconsistency when $SDC[G]$ is inconsistent.

2.4 Dealing With Inconsistency and Previous Work

To simplify our discussion, we assume the following scenario. Suppose a designer has a problem in one of the applications mentioned in the introduction. S/he maps the problem to an SDC, and uses a software tool to check its consistency. The tool internally converts the input SDC to a constraint graph and checks for positive cycles to decide the consistency (by Theorem 1). If no positive

cycles are found, the tool reports "success". If the graph has positive cycles, the tool reports "failure" but also tries to remove them by either (1) changing the topology of the graph, or (2) changing the edge weights, or (3) both. This paper focuses on (2) because it seems "easier". Thus, we assume that the tool will report a list of edges together with the needed weight changes for each edge.

By changing the edge weights, the positive cycles can be converted to either (1) zero cycle, or (2) negative cycles. Without loss of generality, we have chosen (1). Since the designer will be more satisfied if the edge list is minimized, we want to solve the following problem.

PROBLEM 1. *(POSITIVE CYCLES PROBLEM) Given G as in Def. 3, find the minimum number edges whose weights must be changed to make G free of positive cycles.*

The solution of this problem depends on the solution of a more specialized problem called the feedback arc set (FAS) problem.

PROBLEM 2. *(FEEDBACK ARC SET PROBLEM) Given an unweighted, directed graph G, find the minimum number of edges (or arcs) whose removal makes G acyclic.*

Since the FAS problem is NP-hard [16], Prob. 1, being its generalization, is also NP-hard. Due to this unfortunate difficulty, we relax Prob. 1 and opt for a best-effort approach.

PROBLEM 3. *(BEST-EFFORT POSITIVE CYCLES PROBLEM) Given G as in Def. 3, find a "small" set of edges whose weights must be changed to make G free of positive cycles.*

Prob. 3 is our main problem in this paper. The chief difficulty of this problem stems from the number p of positive cycles in G, which can be exponential in n. Not surprisingly, the previous approaches and our approach propose ways of dealing with this difficulty. All the previous approaches can be classified into two main groups: (1) "resolve-all" approaches and (2) "resolve-some" approaches. For simplicity, assume that each approach is implemented in the tool mentioned in the first paragraph.

Resolve-all approaches [10, 21, 31] offer the designer an edge list that removes *all* of the positive cycles from G. These approaches first find all the simple cycles (those that do not contain other cycles) using one of the algorithms in [24], and then visit and resolve each positive cycle. The total worst-case time complexity is exponential ($O(p(n+m))$).

Resolve-some approaches [4, 8, 14, 17, 20] offer the designer an edge list that removes *some* of the positive cycles from G. The edge list is found in polynomial time. The designer is expected to resolve the output positive cycles and rerun the tool until the consistency is achieved. Therefore, the total worst-case time complexity is proportional to p, i.e., exponential. These approaches have the variations as listed below.

1. "Output" variation [4, 8, 20] finds one positive cycle in $O(nm)$ time and just outputs it.
2. "Backward edge" variation [17] first removes all the backward edges from G and repeats the following two steps until all the backward edges are eliminated: (1) temporarily add one backward edge back to G and check for a positive cycle; (2) if a positive cycle is found, output the cycle and eliminate the backward edge from further consideration. This variation runs in $O(bnm)$ time for b backward edges.
3. "Freeze" variation [14] repeats the following two steps: (1) find one positive cycle in $O(nm)$ time; (2) "freezes" it, i.e., collapses it into one vertex. This variation runs in $(O((m-n+1)nm))$ time.

Our approach can also be considered as a resolve-all approach; however, we do not enumerate every positive cycle in G. In the previous resolve-all approaches, the positive cycles to resolve are selected arbitrarily. In the next section, we show that making the selection in a certain way implicitly resolves many other positive cycles such that the total worst-case time complexity becomes strongly polynomial in the size of G.

3. OUR ALGORITHM: FIX

We now present the cycle means (§ 3.1), the graph property that our algorithm is based on, the characterization of the presence of positive cycles using cycle means (§ 3.2), and our algorithm (§ 3.2) together with its correctness (§ 3.3) and running time proofs (§ 3.4). This section contains the main contribution of this paper. Without loss of generality, we assume that G is cyclic.

3.1 Cycle Means

Definition 5. The (cycle) mean $\lambda(C)$ of a cycle C in G is defined as

$$\lambda(C) = \frac{w(C)}{|C|} = \frac{\sum_{e \in C} w(e)}{|C|}, \tag{2}$$

where $|C|$ is the length of C, i.e., the number of edges on it.

Note that (1) the mean of a cycle gives its average edge weight, and (2) the maximum cycle mean in G is well defined since G has a finite number of cycles.

Definition 6. The *maximum cycle mean* $\lambda^*(G)$ of G is defined as

$$\lambda^*(G) = \max_{\forall C \in G} \{\lambda(C)\} \tag{3}$$

A cycle whose mean is equal to the maximum cycle mean is called a *critical* cycle.

The maximum cycle mean of G can found in $O(nm)$ time, e.g., see a list of the possible algorithms in [12]. In practice, we have found out and reported in [11] that the Young-Tarjan-Orlin algorithm (YTO) [33] is one of the fastest maximum cycle mean algorithms. Its time complexity is $O(nm + n^2 \lg n)$ using Fibonacci heaps, but these heaps are not efficient in practice. For our experiments, we used an efficient implementation using binary heaps although it resulted in a higher worst-case time complexity of $O(nm \lg n)$.

3.2 Finding Positive Cycles Using Cycle Means in FIX

The following result gives another characterization of inconsistency and easily follows from Def. 5. It enables us to use a maximum cycle mean algorithm to resolve positive cycles.

THEOREM 2. *A cyclic graph G has at least one positive cycle if and only if* $\lambda^*(G) > 0$.

Given G as an input, YTO computes $\lambda^*(G)$ as well as returns one of the positive cycles that is critical. Using YTO as a subroutine, we get the algorithm FIX in Fig. 2 to restore consistency to $SDC[G]$. FIX iterates until $\lambda^*(G) \leq 0$, which, by Theorem 2, implies that G no longer has any positive cycles. At each iteration, we go over the positive edges of C and change one or more edge weights to make C a zero cycle. We ensure that non-negative edge weights stay non-negative during these weight changes.

Note that lines 5-7 of FIX can easily be replaced to use any other heuristics (e.g., based on "priority" or "criticality" of the edges),

FIX(G)
1. Empty the edge list L.
2. **repeat**
3. $(\lambda^*(G), C)$ = YTO(G).
4. **if** $(\lambda^*(G) > 0)$ **then**
5. Select one or more positive edges on C.
6. Decrease the sum of their weights by $w(C)$, i.e., zero $w(C)$.
7. Add the edges to L.
8. **until** $(\lambda^*(G) \leq 0)$.
9. **return** L.

Figure 2: Our algorithm FIX to resolve the inconsistency in $SDC[G]$. FIX uses the Young-Tarjan-Orlin algorithm (YTO) as a subroutine to find critical cycles.

or to simulate any resolve-some approaches. However, to guarantee the polynomial-time complexity of FIX, any of these heuristics must not increase any edge weight, i.e., they must not create new positive cycles.

The output of FIX is an edge list L. At each iteration of FIX, the edges whose weights have to be changed are added to L. The goal is to make the length $|L|$ of L as small as possible. This goal may be realized in many ways, e.g., by associating counters with edges.

3.3 Correctness of FIX

We first prove the following simple lemma, which states that FIX does not create any new positive cycles. The proofs in the sequel assume the correctness of YTO, e.g., see [33] for a proof.

LEMMA 1. *At line 6 of FIX, zeroing $w(C)$ decreases the weight of every other cycle that shares an edge with C.*

PROOF. We have to show that any non-positive cycle stays as non-positive after making $w(C)$ zero. The proof is by contradiction. Let C' be a non-positive cycle, i.e., $w(C') \leq 0$. Suppose C and C' share a number of edges. By line 5, $w(C')$ can change only if one or more of these edges have positive weights. Assume that the total change in their weights is $\delta w > 0$. Then, if $w(C') - \delta w > 0$ after line 6, we must have $w(C') > \delta w > 0$, which is a contradiction. □

We now prove that FIX is correct.

THEOREM 3. *FIX makes G free of positive cycles in finite time.*

PROOF. Consider an iteration of FIX. By the correctness of YTO, C of line 5 is a positive cycle. By Lemma 1, FIX does not create new positive cycles. Also, FIX decreases the number of positive cycles at least by one. Since G has a finite number of positive cycles, FIX resolves all the positive cycles in finite time. □

3.4 Time Complexity Analysis of FIX

By Theorem 3, FIX takes finite time. We now show that FIX with a slight modification at line 5 is actually a strongly polynomial-time algorithm, i.e., its time complexity is a polynomial function of n and m. The modification is that FIX selects only one edge at line 5. We first prove the following simple but important lemma.

LEMMA 2. *Let e be the edge selected at line 5. At line 6 of FIX, zeroing $w(C)$ makes every other positive cycle C' on e non-positive if $|C'| \leq |C|$.*

PROOF. First, by the correctness of YTO, C of line 5 is a positive cycle that is critical. Thus, by Eq. 2, we must have

$$\frac{w(C)}{|C|} \geq \frac{w(C')}{|C'|}. \tag{4}$$

Second, since $w(e)$ is decreased by $w(C)$ at line 6, the weight of C' after line 6 becomes $w(C') - w(C)$.

Now, assume that C' was a positive cycle before line 6 and $|C'| \leq |C|$. Then, Eq. 4 implies that $w(C) \geq w(C')$, or $0 \geq w(C') - w(C)$. Thus, C' becomes non-positive after line 6. □

We are now ready to prove the main theorem of this paper.

THEOREM 4. *The loop of lines 2-8 in FIX iterates at most nm times.*

PROOF. During its execution, FIX resolves positive cycles of different lengths; however, since these cycles are simple cycles without loss of generality, these lengths range from 1 to n. Thus, the worst-case time complexity of FIX occurs when the length of C of line 5 monotonically increase from 1 to n during the iterations of FIX. Now, we have to find out how many iterations are required to resolve all the positive cycles of a given length, say k, $1 \leq k \leq n$.

Let N_k denote all the positive cycles of length k in G. For each edge e in G, let $N_k(e)$ denote those cycles in N_k that contain e as one of their edges. On the one hand, by Lemma 2, resolving a critical cycle in $N_k(e)$ for some edge e (at line 5-6) resolves every other cycle in $N_k(e)$. On the other hand, $N_k \leq \sum_{e \in G} N_k(e)$. Therefore, to resolve all the cycles in N_k, each edge e needs to be selected *at most once*, or FIX needs to iterate at most m times. Since k ranges from 1 to n, the total number of iterations of FIX is at most nm. □

COROLLARY 1. *The time complexity of FIX on G is $O(nmT(n,m))$ where $T(n,m)$ is the time complexity of computing the maximum cycle mean of G, which is currently $O(nm)$. In particular, FIX with YTO runs in $O(nm(nm + n^2 \lg n))$ time with Fibonacci heaps and in $O(nm(nm \lg n))$ time with binary heaps.*

In [33], it is shown that YTO runs in $O(m + n \lg n)$ time on the average (for random graphs). In [11], we validated this claim for both random graphs and circuits. Therefore, the average time complexity of FIX is at least a factor of n less than its worst-case time complexity.

4. EXPERIMENTAL RESULTS

We now present our experimental setup and test suite (§ 4.1), and the experimental results (§ 4.2).

4.1 Experimental Setup and Test Suite

We wrote all our programs in C++ and built them using the GNU g++ compiler (version 2.95.2). We performed all the experiments on one CPU of an 8-CPU SUN workstation computer running the UNIX operating system (SunOS version 5.7). Each CPU was a 336 MHz UltraSPARC-II processor. This processor has a 16 KB instruction cache and a 16 KB data cache, both of which seem small. The computer had 7.2 gigabytes of main memory, which was more than enough for our experiments to fit in main memory.

Instead of reporting the total running time, we report the running time of the part of the program that takes in a graph that is already in the main memory and produces its output. To give an idea about the total running time, we note that the time to read the largest graph in our test suite and prepare it as an input took less than 10 s.

Table 1: Our test suite (the first three columns) and the experimental results for the algorithms DFS, FAS, POS-CYCLE, and FIX. Here, n denotes the number of vertices, m the number of edges, $|L|$ the length of the edge list returned (i.e., the solution quality), B the bound on the solution quality of FAS, T the running time in seconds, and m_+ the number of positive edges in the input graphs of POS-CYCLE and FIX.

Test			DFS		FAS			POS-CYCLE		FIX		
Name	n	m	$\|L\|$	T	$\|L\|$	B	T	$\|L\|$	T	$\|L\|$	m_+	T
i01	12,752	36,681	6,227	0.01	3,309	14,656	0.03	35	0.31	35	3,694	0.93
i02	19,601	61,829	9,287	0.03	6,140	25,101	0.07	7	0.11	7	1,663	0.25
i03	23,136	66,429	8,334	0.04	7,263	29,168	0.09	8	0.22	8	2,147	0.36
i04	27,507	74,138	10,305	0.04	6,600	32,266	0.11	8	0.21	8	1,284	0.33
i05	29,347	98,793	4,871	0.02	5,350	13,730	0.05	2	0.05	2	196	0.08
i06	32,498	93,493	13,905	0.05	7,566	41,235	0.15	11	0.45	11	3,745	0.95
i07	45,926	127,774	20,586	0.08	11,946	54,211	0.22	49	2.47	48	9,944	7.54
i08	51,309	154,644	29,271	0.10	13,433	67,220	0.26	7	0.58	7	1,553	0.66
i09	53,395	161,430	20,572	0.09	16,104	69,834	0.25	70	4.24	69	10,569	11.77
i10	69,429	223,090	37,240	0.14	23,342	97,826	0.40	33	3.92	33	8,002	6.54
i11	70,558	199,694	25,693	0.13	20,455	85,668	0.38	35	3.31	35	6,256	5.95
i12	71,076	241,135	23,338	0.15	23,524	107,054	0.44	6	1.10	6	3,530	1.01
i13	84,199	257,788	25,022	0.16	21,885	114,642	0.51	8	1.03	8	4,707	1.77
i14	154,605	394,497	57,519	0.33	34,102	170,542	0.98	3	1.52	3	5,331	1.72
i15	161,570	529,562	52,033	0.37	40,915	237,486	1.18	115	33.66	115	30,538	77.51
i16	183,484	589,253	82,799	0.42	54,069	260,718	1.41	41	12.84	41	17,525	18.17
i17	185,495	671,174	63,103	0.46	55,961	302,893	1.55	3	1.93	3	2,387	2.13
i18	210,613	618,020	64,988	0.48	42,340	257,727	1.43	12	3.44	12	6,060	4.66

Our test suite is presented in Table 1 (the first three columns). We generated the graphs i01-i18 in our test suite from the ISPD98 circuit benchmark suite [2]. This test suite contains newer and far larger circuits than those in the ACM/SIGDA (or MCNC) benchmarks. These circuits are not constraint graphs but large enough to give an idea about the running time of our algorithm in practice. The circuits in the ISPD98 had the direction information for their nets. We used this information to generate directed edges and convert the circuits to the directed graphs in our test suite. This way, the total number of edges in our graphs equaled approximately 2.8 times the total number of nets in the original circuits.

We chose the edge weights uniform-randomly out of the interval [-1, -3000]. To generate positive cycles, we computed the maximum cycle mean of the graph, multiplied it by 4, and added it to every edge weight. The column m_+ in Table 1 shows the total number of positive edges resulted.

4.2 Experimental Results and Discussion

We implemented four algorithms to compare: (1) FIX, (2) the depth-first search algorithm (DFS) [9], (3) an approximation algorithm for the feedback arc set problem (FAS) [15], and (4) POS-CYCLE, which is explained below.

POS-CYCLE is an algorithm similar to FIX in that both repeat the following two steps: (1) find a cycle and (2) make it a zero cycle. For both algorithms, the cycle at step 1 is not arbitrary: FIX finds a critical cycle, whereas POS-CYCLE finds a positive cycle, hence, its name. To find a cycle, FIX uses YTO, one of the fastest maximum cycle mean algorithms [11], whereas POS-CYCLE uses Tarjan's algorithm, one of the fastest positive (or negative) cycle detection algorithms [6].

We compared all the implemented algorithms in terms of the running time and the solution quality. The latter refers to the length of the edge list returned. For both criteria, the smaller, the better.

The length of the edge list returned by any algorithm is trivially bounded by m. For DFS, $|L|$ gives the number of the back edges (those edges that create cycles, as defined in [9]) in the input graph. Among the algorithms, only FAS has a provably worst-case solution quality: the length B of the edge list returned by FAS is at most $B(m,n) = m/2 - n/6$ [15].

The experimental results are presented in Table 1. For each algorithm, we give the length L (in the number of edges) of the returned edge list and the running time T (in seconds). In addition, for FAS, we also give the worst-case bound B on L, and for POS-CYCLE and FIX, we also give (in FIX's columns only) the number m_+ of positive edges in the input graph.

The results in Table 1 indicate that (1) $|L[FAS]|$ is almost half $|L[DFS]|$; (2) the bound on $|L[FAS]|$ is not tight; (3) $|L[FIX]|$ is slightly shorter than $|L[POS-CYCLE]|$; (4) $|L[POS-CYCLE]|$ and $|L[FIX]|$ are three orders of magnitude (1300x) shorter than $|L[FAS]|$; and (5) FIX is the slowest algorithm although its running time is in a few seconds.

From these observations, we conclude that DFS and FAS should not be used at all for over-constraint resolution because their solution quality is not satisfactory. After eliminating them, the choice is between POS-CYCLE and FIX. In terms of the solution quality, FIX is the best algorithm though the difference is insignificant: except for i07 and i09, both algorithms return the same $|L|$; for i07 and i09, $|L[FIX]|$ is one less than $|L[POS-CYCLE]|$.

In terms of the running time, POS-CYCLE is almost 2x faster than FIX although the running times of all four algorithms are almost negligible compared to the time (around 10s) to read the input graph. For only i15 and i16, POS-CYCLE and FIX take more time than the time it takes to read the input.

In addition to the running time, we should also compare POS-CYCLE and FIX in terms of their worst-case time complexity. As proved in this paper, FIX runs in strongly polynomial time, whereas POS-CYCLE can run in exponential time [1]. Note that we have not proved the worst-case time complexity of POS-CYCLE in this paper; instead, we have extrapolated it from those of similar algorithms for the minimum cost flow problem [1]. Interestingly, although we developed FIX independently, we later realized that similar algorithms had been developed to obtain the first strongly polynomial time algorithm for the minimum cost flow problem [1].

5. CONCLUSIONS

We have defined the problem of ensuring the consistency of a system of difference constraints in terms of the problem of removing the positive cycles from the corresponding constraint graph. We have reviewed the previous approaches to these problems and noted that they result in exponential time algorithms. We have then presented our approach and algorithm and showed that the algorithm is

correct and runs in strongly polynomial time. Finally, we have discussed our experiments done to compare our algorithm with three other algorithms on very large circuit benchmarks. The experimental results show that our algorithm is very efficient in practice, and its solution quality is the best.

6. REFERENCES

[1] AHUJA, R. K., MAGNANTI, T. L., AND ORLIN, J. B. *Network Flows*. Prentice Hall, Upper Saddle River, NJ, USA, 1993.

[2] ALPERT, C. J. The ISPD98 circuit benchmark suite. In *Proc. Int. Symp. on Physical Design* (1998), ACM/IEEE, pp. 588–93.

[3] BORRIELLO, G. Specification and synthesis of interface logic. In *High-Level VLSI Synthesis*, R. Camposano and W. Wolf, Eds. Kluwer Academic Publ., 1991, ch. 7, pp. 153–176.

[4] BRZOZOWSKI, J. A., GAHLINGER, T., AND MAVADDAT, F. Consistency and satisfiability of waveform timing specifications. *Networks 21* (1991), 91–107.

[5] CANDAN, K. S., PRAHJAKARAN, B., AND SUBRAHMANIAN, V. S. Collaborative multimedia documents: Authoring and presentation. *Int. J. of Intelligent Syst. on Multimedia Comput. Syst. 13*, 12 (1998).

[6] CHERKASSKY, B. V., AND GOLDBERG, A. V. Negative-cycle detection algorithms. In *Proc. 4th European Symp. on Algorithms* (1996), pp. 349–63.

[7] CHERKASSKY, B. V., GOLDBERG, A. V., AND RADZIK, T. Shortest path algorithms: Theory and experimental evaluation. In *Proc. 5th ACM-SIAM Symp. on Discrete Algorithms* (1994), pp. 516–25.

[8] CHOU, P., AND BORRIELLO, G. Software scheduling in the co-synthesis of reactive real-time systems. In *Proc. 31st Design Automation Conf.* (June 1994), pp. 1–4.

[9] CORMEN, T. H., LEISERSON, C. E., AND RIVEST, R. L. *Introduction to Algorithms*. MIT Press, Cambridge, MA, USA, 1991.

[10] DASDAN, A. Timing analysis of embedded real-time systems. PhD thesis, UIUC technical reports UIUCDCS-R-99-2079 and UILU-ENG-99-1702., Univ. of Illinois at Urbana-Champaign, May 1999.

[11] DASDAN, A. Experimental analysis of the fastest optimum cycle ratio and mean algorithms. Tech. Rep. 2001-10-22-01, Synopsys, Inc., Oct. 2001.

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

[13] DECHTER, R., MEIRI, I., AND PEARL, J. Temporal constraint networks. *Artificial Intelligence 49* (1991), 61–95.

[14] DO, J., AND DAWSON, W. SPACER II: A well-behaved IC layout compactor. In *Proc. of VLSI 85 Int. Conf.* (Aug. 1985), pp. 283–91.

[15] EADES, P., LIN, X., AND SMYTH, W. F. A fast and effective heuristic for the feedback arc set problem. *Infor. Proc. Letters 47*, 6 (1993), 319–23.

[16] GAREY, M. R., AND JOHNSON, D. S. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York, NY, USA, 1979.

[17] ISHIMA, K., TSUKIYAMA, S., AND SHINODA, S. An algorithm to detect positive cycles in a constraint graph for layout compaction. In *Proc. IEEE Int. Symp. on Circuits and Syst.* (1990), pp. 2853–6.

[18] JAHANIAN, F., AND MOK, A. K.-L. Safety analysis of timing properties in real-time systems. *IEEE Trans. Software Eng. 12*, 9 (Sept. 1986), 890–904.

[19] KINGSLEY, C. A hiererarchical, error-tolerant compactor. In *Proc. 21st Design Automation Conf.* (June 1984), pp. 126–32.

[20] KU, D. C., AND MICHELI, D. D. Relative scheduling under timing constraints: Algorithms for high-level synthesis of digital circuits. *IEEE Trans. Computer-Aided Design 11*, 6 (June 1992), 696–718.

[21] LIAO, Y.-Z., AND WONG, C. K. An algorithm to compact a VLSI symbolic layout with mixed constraints. *IEEE Trans. Computer-Aided Design 2*, 2 (Apr. 1983), 62–9.

[22] LIU, J., CHOU, P., BAGHERZADEH, N., AND KURDAHI, F. A constraint-based application model and scheduling techniques for power-aware systems. In *Proc. Int. Symp. on HW/SW Codesign* (Apr. 2001), pp. 153–8.

[23] LY, T., KNAPP, D., MILLER, R., AND MACMILLEN, D. Scheduling using behavioral templates. In *Proc. 32nd Design Automation Conf.* (June 1995), pp. 101–6.

[24] MATETI, P., AND DEO, N. On algorithms for enumerating all circuits of a graph. *SIAM J. Comput. 5*, 1 (Mar. 1976), 90–8.

[25] MATHUR, A., DASDAN, A., AND GUPTA, R. K. Rate analysis of embedded systems. *ACM Trans. on Design Automation of Electronic Syst. 3*, 3 (July 1998), 408–36.

[26] MLYNSKI, D. A., AND SUNG, C.-H. Layout compaction. In *Layout Design and Verification*, T. Ohtsuki, Ed. Elsevier Science, 1986, ch. 6, pp. 199–235.

[27] MOK, A. K., TSOU, D.-C., AND DE ROOIJ, R. C. M. The MSP.RTL real-time scheduler synthesis tool. In *Proc. 17th IEEE Real-Time Systems Symp.* (Dec. 1996), pp. 118–28.

[28] NESTOR, J. A., AND KRISHNAMOORTHY, G. SALSA: A new approach to scheduling with timing constraints. *IEEE Trans. Computer-Aided Design 12*, 8 (Aug. 1993), 1107–22.

[29] RAJU, S. C. V., RAJKUMAR, R., AND JAHANIAN, F. Monitoring timing constraints in distributed real-time systems. In *Proc. 13th IEEE Real-Time Systems Symp.* (Dec. 1992), pp. 57–67.

[30] SCHIELE, W. L. Compaction with incremental over-constraint resolution. In *Proc. 25th Design Automation Conf.* (June 1988), pp. 390–5.

[31] SHI, J.-F., AND CHAO, L.-F. Resource constrained algebraic transformation for loop pipelining. In *Proc. 6th Great Lakes Symp. on VLSI* (1996), pp. 14–7.

[32] YEN, T.-Y., AND WOLF, W. An efficient graph algorithm for FSM scheduling. *IEEE Trans. VLSI Syst. 4*, 1 (Mar. 1996), 98–112.

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