JOURNAL OF ALGORITHMS 9, 114–128 (1988)

A Mixed-Integer Linear Programming Problem Which Is Efficiently Solvable*

CHARLES E. LEISERSON

Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139

AND

JAMES B. SAXE

Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania 15213

Received September 18, 1986

Efficient algorithms are known for the simple linear programming problem where each inequality is of the form $x_j - x_i \leq a_{ij}$. Furthermore, these techniques extend to the integer linear programming variant of the problem. This paper gives an efficient solution to the mixed-integer linear programming variant where some, but not necessarily all, of the unknowns are required to be integers. The algorithm we develop is based on a graph representation of the constraint system and runs in $O(|V||E| + |V|^2 \lg|V|)$ time. It has several applications including optimal retiming of synchronous circuitry, VLSI layout compaction in the presence of power and ground buses, and PERT scheduling with periodic constraints.

1. INTRODUCTION

Much research has centered on the problem of finding shortest paths in graphs. It is well known that there is a direct correspondence between the *single-source shortest-paths problem* and the following simple linear

*This research was supported in part by the Defense Advanced Projects Agency under Contract N00014-80-C-0622 and by the Office of Naval Research under Contract N00014-76-C-0370.

0196-6774/88 $3.00

programming problem:

> *Let S be a set of linear inequalities of the form $x_j - x_i \leq a_{ij}$, where the x_i are unknowns and the a_{ij} are given real constants. Determine a set of values for the x_i such that the inequalities in S are satisfied, or determine that no such values exist.*

This paper considers the *mixed-integer* linear programming variant of this problem in which some (but not necessarily all) of the x_i are required to be integers. The problem arises in the context of synchronous circuit optimization [9], but it has applications to PERT scheduling and VLSI layout compaction as well.

Before formally defining the mixed-integer programming problem, we restate the linear programming problem above in another form.

PROBLEM L. *Let $G = (V, E, a)$ be an edge-weighted, directed graph, where $V = \{1, 2, \ldots, |V|\}$ is the vertex set, the set E of edges is a subset of $V \times V$, and for each edge $(i, j) \in E$ the edge weight a_{ij} is a real number. Find a vector $x = (x_1, x_2, \ldots, x_{|V|})$ satisfying the constraint that*

$$x_j - x_i \leq a_{ij}$$

for all $(i, j) \in E$, or determine that no feasible vector exists.

The graph G is called a *constraint graph* for the linear programming problem. There are three advantages in adopting a graph representation of the problem. First, an adjacency-list representation [1, p. 200] of the constraint graph G is more economical than, for example, a linear programming tableau or, when the graph has relatively few edges, a matrix of the a_{ij}. Second, Problem L frequently arises in situations that are naturally described by a graph. Finally, the graph-theoretic formulation helps in understanding the algorithms that solve this kind of problem.

A method for solving Problem L was discovered in the late 1950s by Ford and Bellman [8, p. 74]. Yen [13] gave some improvements to the Bellman–Ford algorithm as well as a cogent analysis showing that its running time is $O(|V|^3)$. This bound is easily improved to $O(|V|\,|E|)$ by using an adjacency-list representation for the constraint graph.

The Bellman–Ford algorithm can also be used to solve the *integer* linear programming variant of Problem L, in which all the x_i are required to be integers. If the edge weights a_{ij} all happen to be integers, the Bellman–Ford algorithm will produce integer values for the x_i. If the a_{ij} are not integers, however, but the x_i are required to be integers, each edge weight a_{ij} may be replaced by $\lfloor a_{ij} \rfloor$ without affecting the satisfiability of the inequalities.

The focus of this paper is the *mixed-integer* variant of Problem L.

PROBLEM MI. *Let $G = (V, V_I, E, a)$ be a edge-weighted, directed graph, where $V = \{1, 2, \ldots, |V|\}$ is the vertex set, the set V_I is a subset of V, the set E of edges is a subset of $V \times V$, and for each edge $(i, j) \in E$ the edge weight a_{ij} is a real number. Find a vector $x = (x_1, x_2, \ldots, x_{|V|})$ satisfying the constraints that*

$$x_j - x_i \leq a_{ij}$$

for all $(i, j) \in E$ and that $x_i \in \mathbf{Z}$ for all $i \in V_I$, or determine that no feasible vector exists.

The vector $x = (x_1, x_2, \ldots, x_{|V|})$ is called a *solution* to graph G, and if graph G has a solution, we say that G is *satisfiable*. When it is clear from context, we use the same terminology for Problem L.

In addition, we shall refer to the vertices in V_I as the *integer* vertices of G and the vertices in $V_R = V - V_I$ as the *real* vertices of G. We also partition the set of edges into two sets depending on whether the vertex at the head of the edge is integer or real:

$$E_I = \{(i, j) \in E | j \in V_I\},$$
$$E_R = \{(i, j) \in E | j \in V_R\}.$$

This paper presents two algorithms to solve Problem MI. The first, which runs in $O(|V||V_I||E|)$ time, is a straightforward extension of the Bellman–Ford algorithm. The second is more sophisticated and has a running time of $O(|V||E| + |V||V_I|\lg|V|)$. We conjecture that the $O(|V||E|)$ running time achieved by the Bellman–Ford algorithm for the pure linear programming and pure integer programming versions of the problem is not achievable in general for sparse instances of Problem MI.

The remainder of this paper is organized as follows. Section 2 reviews the Bellman–Ford algorithm. Section 3 presents a simple relaxation algorithm for solving Problem MI. Section 4 discusses three techniques—Dijkstra's algorithm, reweighting, and Fibonacci heaps—which are used in Section 5 to construct an asymptotically efficient algorithm for Problem MI. We discuss applications and present some concluding remarks in Section 6.

2. SHORTEST PATHS AND THE BELLMAN–FORD ALGORITHM

This section reviews how the *Bellman–Ford algorithm* solves Problem L. Although the results of this section are well known and can be found in most textbooks on combinatorial optimization (see, for example, [8, p. 74]), we repeat the material here for the reader's convenience.

There is a natural correspondence between Problem L and the graph-theoretic *single-source shortest-paths* problem. Let $G = (E, V, a)$ be an in-

stance of Problem L. Suppose that for each vertex $i \in V$, there is a path to i from vertex 1, and let d_i be the weight of shortest (least-weight) path from vertex 1 to vertex i. (At the end of the section, we shall discuss the case in which some vertices are not reachable from vertex 1.) Then for any edge $(i, j) \in E$, we have $d_j - d_i \leq a_{ij}$ since the edge (i, j) can be appended to a shortest path from vertex 1 to vertex i to produce a path from vertex 1 to vertex j of weight $d_i + a_{ij}$. Thus the shortest-path weights d are a solution to G.

Whenever G is satisfiable, there are infinite number of solutions. For example, if x is a solution to G, then uniformly adding any constant k to each x_i yields another solution y, where $y_i = x_i + k$ for each $i \in V$. The assignment $x_i \leftarrow d_i$ gives each x_i its largest possible value subject to the constraint that $x_1 = 0$. To see this, consider any path p of weight d_i from vertex 1 to vertex i. If the inequalities associated with the edges of p are summed, the unknowns associated with the intermediate vertices cancel and the result is the inequality $x_i - x_1 \leq d_i$.

Whenever the graph G contains some cycle c whose weight is negative, the shortest path weight from vertex 1 to any vertex i on cycle c is undefined because the weight of any path to vertex i can be diminished by appending a traversal of c. In this case the graph G is not satisfiable. If the inequalities associated with the edges of c are summed, all the unknowns x_i cancel, and the resulting inequality asserts that 0 is less than or equal to the weight of c, which is false.

The Bellman–Ford algorithm, which is given below, solves Problem L by finding the weight of the shortest path to each vertex 1. Should the graph contain a negative-weight cycle, the algorithm reports that the graph is unsatisfiable by calling the procedure *Fail*, whose semantics we leave unspecified.

ALGORITHM BF *(Bellman–Ford algorithm).*

```
BF1. x_1 ← 0;
BF2. for i ← 2 to |V| do x_i ← ∞;
BF3. for ind ← 1 to |V| − 1 do
BF4.     foreach (i, j) ∈ E do
BF5.         x_j ← min(x_j, x_i + a_ij);
BF6. foreach (i, j) ∈ E do
BF7.     if x_j > x_i + a_j then Fail;
```

For each vertex $j \in V$, the Bellman–Ford algorithm iteratively updates the weight x_j of a tentative shortest path from vertex 1 to vertex j. After initialization, the algorithm makes $|V| - 1$ passes through the edges in E and *relaxes* each edge (i, j) by computing $x_j \leftarrow \min(x_j, x_i + a_{ij})$.

A simple analysis due to Yen [13] indicates why the Bellman–Ford algorithm works. The weight x_j converges to the weight d_j of a shortest path from vertex 1 to vertex j if the edges on the path are relaxed in order along the path. The sequence of edges relaxed by the Bellman–Ford algorithm consists of $|V| - 1$ copies of some ordering of E, and therefore contains every vertex-disjoint path as a subsequence. If there are no negative-weight cycles in G, then every shortest path is vertex disjoint, so each x_i converges to the shortest-path weight d_i. On the other hand, if there is a negative-weight cycle in the graph, the algorithm detects this condition by iterating once more through all edges to see whether any of the inequalities remain unsatisfied.

The Bellman–Ford algorithm as given above determines the weight of the shortest path from vertex 1 to each vertex, and therefore solves Problem L whenever all vertices of G are reachable from vertex 1. The code can be adapted to solve Problem L on arbitrary graphs by simply changing the initialization step (lines BF1–BF2). In particular, if each x_i is assigned a finite initial value u_i, the relaxation in lines BF3–BF5 sets each x_i to its maximum value subject to the constraints that $x_j - x_i \leq a_{ij}$ for each edge $(i, j) \in E$ and that $x_i \leq u_i$ for each vertex $i \in V$. Notice that whenever the constraint graph G is satisfiable, it is satisfiable subject to the additional constraints $x_i \leq u_i$. Should the inequalities be inconsistent because there is a negative-weight cycles in the graph, the relaxation will not converge to a solution, and the inconsistency will be detected by the test in lines BF6–BF7.

3. Simple Relaxation Algorithms for Problem MI

As was mentioned in the introduction, Problem MI can be solved directly by the Bellman–Ford algorithm when all unknowns are real (Problem L) and when all unknowns are integer. The combination of integer and real unknowns, however, seems to make the problem harder. In this section, we gain some intuition about the structure of Problem MI by introducing two algorithms that solve it in $O(|V||V_I||E|)$ time much the same way as the Bellman–Ford algorithm solves Problem L. The asymptotically efficient algorithm in Section 5 is derived from the second of these algorithms.

A natural approach to solving Problem MI is to see whether the Bellman–Ford relaxation approach can be made to work. Since we have both integer and real vertices in the graph, however, we must modify the relaxation step BF5 in the Bellman–Ford algorithm to produce an integer value whenever j is an integer vertex (line R6, Algorithm R). This approach does in fact work, but it requires more iterations than the simple Bellman–Ford algorithm. The next algorithm solves Problem MI. The

number of iterations n in line R2 will be determined in the analysis following the algorithm.

ALGORITHM R *(Relaxation).*

R1. **foreach** $i \in V$ **do** $x_i \leftarrow 0$;
R2. **for** *ind* $\leftarrow 1$ **to** n **do**
R3. **foreach** $(i, j) \in E$ **do**
R4. **begin**
R5. $x_j \leftarrow \min(x_j, x_i + a_{ij})$;
R6. **if** $j \in V_I$ **then** $x_j \leftarrow \lfloor x_j \rfloor$;
R7. **end**;
R8. **foreach** $(i, j) \in E$ **do**
R9. **if** $x_j > x_i + a_{ij}$ **then** *Fail*;

In order to determine a value of n such that Algorithm R works, we introduce the notion of a *reducing path*. Let p be a path starting at some vertex k, and suppose that x_k is initially set to 0 and that all the remaining x_i are initialized to ∞. Suppose the edges in path p are traversed in order starting from k, and each edge (i, j) along the path is relaxed as in statements R5–R6. If each relaxation of an edge (i, j) reduces the value x_j, the path p is called a *reducing path*.

Whenever a sequence of edges contains all reducing paths as subsequences, the relaxation of each edge in the sequence in order yields a solution. (The proof is analogous to Yen's analysis [13] of the Bellman–Ford algorithm.) The Bellman–Ford algorithm solves Problem L because in a satisfiable graph with only real vertices, each vertex occurs at most once on any single reducing path. (And in fact, every shortest path is a reducing path.)

When some unknowns are integer and some are real, however, it is possible for a reducing path to visit the same vertex more than once, even if the graph is satisfiable. For example, in the graph shown in Fig. 1, the reducing path $p = 3 \rightarrow 2 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 3 \rightarrow 2$ visits vertices 2 and 3 three times each. If all the x_i are initially set to 0, the edges of p must be relaxed in their order along the path to achieve convergence. Moreover, relaxing the entire edge set in some arbitrary order only $3 = |V| - 1$ times might not achieve convergence. Since the value of n in line R2 must be at least the maximum number of edges in any reducing path, the value $|V| - 1$, which was used in Algorithm BF, will not suffice.

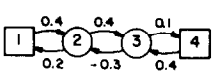

FIG. 1. An instance of Problem MI. Integer vertices ($V_I = \{1, 4\}$) are shown as squares and real vertices as circles.

Fortunately, reducing paths are never very long in satisfiable graphs because of the following lemma.

LEMMA 1. *Suppose* $G = (V, V_I, E, a)$ *is satisfiable. If* p *is a reducing path in* G, *then*

1. p *visits no integer vertex more than once, and*
2. p *never visits the same real vertex twice without visiting some integer vertex in between.*

Proof. If either condition is violated, then the reducing path p can be extended indefinitely by repeating the cycle that causes violation. □

Lemma 1 allows us to determine a value for n in line R2 of Algorithm R such that the x converges to a solution whenever G is satisfiable. Any reducing path contains each integer vertex at most once and each real vertex at most $|V_I| + 1$ times. Since the number of edges in a path is one less than the number of vertices, any reducing path for a satisfiable graph can have no more than $|V_I| + (|V_I| + 1)|V_R| - 1 = |V_I||V_R| + |V| - 1$ edges. Thus the limit n of the outer loop in Algorithm R should be set to $|V_I||V_R| + |V| - 1$. The overall running time of Algorithm R is thus $O(|V||V_I||E|)$.

This analysis suggests the following algorithm, which is slightly more efficient than Algorithm R, and which forms the basis of the asymptotically efficient algorithm presented in the next section.

ALGORITHM M *(Modified relaxation).*

```
M1.  foreach i ∈ V do x_i ← 0;
M2.  for ind ← 1 to |V_R| do
M3.      foreach (i, j) ∈ E_R do
M1.          x_j ← min(x_j, x_i + a_ij);
M5.  for ind2 ← 1 to |V_I| do
M6.      begin
M7.      foreach (i, j) ∈ E_I do
M8.          x_j ← min(x_j, ⌊x_i + a_ij⌋);
M9.      for ind ← 1 to |V_R| do
M10.         foreach (i, j) ∈ E_R do
M11.             x_j ← min(x_j, x_i + a_ij);
M12.     end;
M13. foreach (i, j) ∈ E do
M14.     if x_j > x_i + a_ij then Fail;
```

The only difference between this algorithm and Algorithm R is that it treats the edges in E_I separately from the edges in E_R.[1] In lines M7–M8 of

[1] As is done in [7], for example.

Algorithm M, each edge in E_I is relaxed once. There are $|V_I|$ such passes over E_I which are preceded, followed, and separated by *exhaustive relaxations* of the edges in E_R (lines M2–M4 and M9–M11). In each exhaustive relaxation of E_R, edges are relaxed until no further changes in the values of x_j are possible for $j \in V_R$. (Actually, the relaxations in lines M2–M4 and M9–M11 are only guaranteed to be exhaustive if there are no negative-weight cycles in E_R. If there are cycles of negative weight, however, this condition is detected at the end by the convergence test in lines M13–M14.)

4. Dijkstra's Algorithm and Reweighting

Section 5 gives a more efficient algorithm to solve Problem MI than either Algorithm R or Algorithm M. Three important techniques are used in the algorithm. The first is Dijkstra's algorithm which finds shortest paths in a graph from a single source in the case when all the edge weights are nonnegative. The second is *reweighting*, which is a technique used by Johnson [7] in his efficient algorithm for solving the *all-pairs shortest-paths problem*. The third is the *Fibonacci heap* data structure due to Fredman and Tarjan [4], which is an improved priority queue that makes Dijkstra's algorithm run in time $O(|E| + |V|\lg|V|)$.

Given a graph $G = (V, E, a)$ such that all edge weights a_{ij} are nonnegative, Dijkstra's algorithm computes for each vertex i, the weight d_i of the shortest path from vertex 1. Because each edge is relaxed exactly once, this algorithm is faster than the Bellman–Ford algorithm which solves the same problem for arbitrary edge weights. Dijkstra's algorithm derives its efficiency from the observation that along any shortest path from vertex 1, the shortest-path weights d_i form a nondecreasing sequence if all the edge weights are nonnegative. Thus, a sequence consisting of all edges $(i, j) \in E$ in nondecreasing order of the distances d_i contains as subsequences shortest paths from vertex 1 to all vertices in V. Furthermore, such a sequence of edges can be computed on the fly using a priority queue. (The textbook [1] gives a proof of correctness for this algorithm.)

ALGORITHM D *(Dijkstra's algorithm).*

D1. $x_1 \leftarrow 0$;
D2. **for** $i \leftarrow 2$ **to** $|V|$ **do** $x_i \leftarrow \infty$;
D3. $Q \leftarrow V$;
D4. **while** $Q \neq \emptyset$ **do**
D5. **begin**
D6. *Choose* $i \in Q$ *such that* $x_i = \min_{j \in Q} x_j$;

D7. $Q \leftarrow Q - \{i\}$;
D8. **foreach** $j \in V$ *such that* $(i, j) \in E$ **do**
D9. $\quad x_j \leftarrow \min(x_j, x_i + a_{ij})$;
D10. **end**;

If the set Q in the algorithm is implemented as a standard priority queue, each extraction (lines D6–D7) and update (line D9) costs $O(\lg|Q|) = O(\lg|V|)$ time. Thus the total running time of Dijkstra's algorithm is $O(|E|\lg|V|)$. Fredman and Tarjan [4] describe a data structure called Fibonacci heaps that supports arbitrary deletion in $O(\lg n)$ amortized time and all other standard priority queue operations (including update) in constant amortized time. By using a Fibonacci heap in Dijkstra's algorithm, they show that the performance can be improved to $O(|E| + |V|\lg|V|)$.

Since Dijkstra's algorithm is equivalent to the Bellman–Ford algorithm on graphs with nonnegative edge weights, it can be used to solve Problem L on such graphs. This is not very interesting in itself, since any graph $G = (V, E, a)$ in which all edge weights are nonnegative can be trivially satisfied by setting x_i to 0 for each $i \in V$. Our interest in Dijkstra's algorithm comes from a stronger property of the solutions it finds. Suppose the initialization step (lines D1–D2) is changed so that each variable x_i is initialized to a finite value u_i. Then the relaxation procedure in lines D3–D10 will set each x_i to its largest possible value consistent with the constraints that $x_j - x_i \le a_{ij}$ for each edge $(i, j) \in E$ and that $x_i \le u_i$ for each vertex $i \in V$. In other words, lines D3–D10 of Dijkstra's algorithm are functionally equivalent to lines BF3–BF5 of the Bellman–Ford algorithm provided that all the edge weights a_{ij} are nonnegative. Since a graph with only nonnegative edge weights can never contain a negative-weight cycle, no test for convergence is necessary in this case.

The efficient algorithm we shall present to solve Problem MI is a modification of Algorithm M. Notice that lines M9–M11 of Algorithm M exhaustively relax the edges in E_R in a manner similar to lines BF3–BF5 of the Bellman–Ford algorithm. In Algorithm M, however, this code is executed many times. The efficient algorithm to solve Problem MI uses a trick to replace this code with code based on the more efficient relaxation procedure in lines D3–D10 of Dijkstra's algorithm. This trick is the technique of *reweighting* used extensively in the literature [3], [7].

LEMMA 2. *Let $G = (V, E, a)$ be an edge-weighted graph, for each $i \in V$ let r_i be a real number, and let $H = (V, E, b)$, where $b_{ij} = a_{ij} + r_i - r_j$ for each edge $(i, j) \in E$. For each vertex $i \in V$ let x_i be a real number and let $y_i = x_i - r_i$. Then $x_j - x_i \le a_{ij}$ for all $(i, j) \in E$ if and only if $y_j - y_i \le b_{ij}$ for all $(i, j) \in E$ (that is, x is a solution to G if and only if y is a solution to H.)*

Proof. Trivial. □

We call the vector $r = (r_1, r_2, \ldots, r_{|V|})$ a *reweighting* of the graph G.

5. An Asymptotically Efficient Algorithm for Solving Problem MI

This section shows how Dijkstra's algorithm and reweighting can be incorporated into Algorithm M to yield a faster algorithm for solving Problem MI. Given a graph $G = (V, V_I, E, a)$, the idea is to find a reweighting r such that the reweighted graph $H = (V, V_I, E, b)$ has edge weights $b_{ij} = a_{ij} + r_i - r_j \geq 0$ for all edges $(i, j) \in E_R$. Lemma 2 guarantees that G is satisfiable if and only if H is satisfiable and also that a solution y to H can be converted into a solution x to G by setting $x_i = y_i + r_i$ for each $i \in V$. The advantage gained by transforming the problem on G to a problem on H is that the relaxation portion of Dijkstra's algorithm (lines D3–D10) can replace the Bellman–Ford relaxation (lines M9–M11), which is the most expensive part of Algorithm M.

The first stage of the algorithm is to determine the reweighting values r_i for all $i \in V$ and the new edge weights $b_{ij} = a_i + r_i - r_j$ for all $(i, j) \in E$. We must choose the values r_i such that $b_{ij} \geq 0$ for all $(i, j) \in E_R$. Since this is equivalent to requiring that $r_j - r_i \leq a_{ij}$ for all $(i, j) \in E_R$, values for the r_i can be found by applying the Bellman–Ford algorithm to the graph (V, E_R, a). The first few lines of the algorithm are:

ALGORITHM T *(Efficient algorithm).*

```
T1. for i ∈ V do r_i ← 0;
T2. for ind ← 1 to |V_R| do
T3.     for (i, j) ∈ E_R do
T4.         r_j ← min(r_j, r_i + a_ij);
T5. for (i, j) ∈ E_R do
T6.     if r_j > r_i + a_ij then Fail
T7. for (i, j) ∈ E do
T8.     b_ij ← a_ij + r_i − r_j;
```

If the algorithm fails in line T6, then there is a cycle of negative weight among the edges in E_R, and hence graph G is unsatisfiable even in the absence of integer constraints. Otherwise, the values b_{ij} computed in line T8 are nonnegative for all $(i, j) \in E_R$.

The next stage of Algorithm T is to solve the mixed-integer problem on the graph $H = (V, V_I, E, b)$. The algorithm alternately performs single relaxation passes on the edges in E_I and exhaustive relaxations of the edges

in E_R, as in Algorithm M. We begin by initializing the values y_i, which will converge to a solution to H if H is satisfiable.

T9. **for** $i \in V$ **do** $y_i \leftarrow 0$;

This initialization has the added fortune of subsuming the first exhaustive relaxation of E_R (lines M2–M4 in Algorithm M). After the execution of line T9 we have $y_j - y_i = 0 - 0 \leq b_{ij}$ for all $(i, j) \in E_R$, which means that the edges in E_R are already exhaustively relaxed.

The next portion of Algorithm T parallels lines M5–M12 of Algorithm M and is where most of the computing gets done.

```
T10. for ind ← 1 to |V_I| do
T11.     begin
T12.     for (i, j) ∈ E_I do
T13.         y_j ← min(y_j, ⌊y_i + b_ij⌋);
T14.     Q ← V;
T15.     while Q ≠ ∅ do
T16.         begin
T17.         Choose i ∈ Q such that y_i = min_{j∈Q} y_j;
T18.         Q ← Q − {i};
T19.         for j ∈ V_R such that (i, j) ∈ E_R do
T20.             y_j ← min(y_j, y_i + b_ij);
T21.         end;
T22.     end;
```

This code solves the problem on graph H in almost exactly the same way that Algorithm M would. The only difference is the method by which the edges of E_R are exhaustively relaxed. Whereas lines M9–M11 of Algorithm M perform the exhaustive relaxation using the Bellman–Ford algorithm, lines T14–T21 of Algorithm T take advantage of the nonnegativity of the b_{ij} for $(i, j) \in E_R$ and use Dijkstra's algorithm.

The final part of Algorithm T is to check the convergence of the y and to apply Lemma 2 to produce a satisfying assignment x for the original graph G.

```
T23. for (i, j) ∈ E_I do
T24.     if y_j > y_i + b_ij then Fail;
T25. for (i, j) ∈ E do
T26.     x_i ← y_i + r_i;
```

Lines T23–T24 check the convergence of y by testing the inequalities associated with the edges in E_I. The inequalities resulting from edges in E_R need not be checked because the relaxation in lines T14–T21 is guaranteed to be exhaustive. (If there were negative-weight cycles in E_R, we would have detected this in lines T5–T6.)

Lines T25–T26 convert the solution y to graph H into a solution x to graph G. Lemma 2 ensures that the inequalities $x_j - x_i \leq a_{ij}$ are satisfied, but we must also show that the x_i are integers for all $i \in V_I$. For each $i \in V_I$ the value y_i is an integer, however, and furthermore, the values of the r_i produced in lines T1–T4 are zero for all $i \in V_I$. Thus for all the integer vertices, the x_i are integers.

In summary, we have proved the following theorem.

THEOREM 3. *Algorithm T solves Problem MI.*

The running time of Algorithm T is $O(|V||E| + |V||V_I|\lg|V|)$, if the priority queue is implemented using a Fibonacci heap.

6. Applications, Extensions, and Conclusions

The solution to Problem MI was demanded by a problem concerning optimization of sychronous circuitry by retiming [9]. This section briefly describes two other problems that can be reduced to Problem MI: compaction of VLSI circuits in the presence of power and ground buses and PERT scheduling with periodic constraints. We also consider an extension of Problem MI where multiple classes of periodic constraints must be satisfied. (For example, some of the x_i are required to be integers, and others to be exact multiples of an integer constant c.)

Circuit Compaction

Optimal (one-dimensional) compaction of VLSI circuit layouts [5] is another application of the Bellman–Ford algorithm. Each layout feature is given a variable representing an x-coordinate, and the design rules are enforced using constraints of the form $x_j - x_i \leq a_{ij}$. It may be desirable, however, to allow feature i to be to the left of feature j or vice versa, but not to allow them to occupy the same position. Unfortunately, if one wishes to allow this kind of transposition of layout features, either optimality or performance must be sacrificed because the problem becomes NP-complete [10]. But for certain compaction problems arising in practice, transposition of layout features can be allowed.

Some design methodologies enforce the placement of power, ground, and clock to be at regular intervals. For example, one signal processing system [11] requires that these wires be repeated every 200λ, and that the width of all cells in the system be integer multiples of this distance. The designer is then constrained to build a new cell so that the layout features are tightly packed among the global wires. In this context, where some layout features

may go on one side or the other of some global wire but may not overlap, the compaction problem can be formulated as Problem MI.

PERT Scheduling

Suppose we have a constraint graph with vertices representing milestones in a project, and edge-weights indicating the timing constraints between milestones. Generally, the Bellman–Ford algorithm can be used to provide an optimal scheduling of the milestones. If a work day is from 9:00 AM to 5:00 PM, however, we may not wish to schedule a one-hour job to start at 4:30 PM. Advancing the job to the next day may cause an earlier job to be advanced as well if the two jobs are constrained to fall near each other. The problem of PERT scheduling with periodic constraints can be cast as Problem MI.

Intuitively, the mixed-integer formulation allows one to include for each job (1) a real variable representing the starting time of the job, and (2) an integer variable representing, say, noon on the day the job occurs. Thus one can include constraints which say, for example, "This job must start before 4:00 PM on the day it occurs."

Multiple Periodic Constraints

Suppose that in the PERT scheduling application mentioned above, we also wish to take into consideration constraints involving weekends. To do this, we would associate with each job a third variable representing, say, Sunday noon of the week during which the job occurs. We are then required to solve a variant of Problem MI in which there are two classes of periodic constraints, where some variables are required to be exact integers and others to be exact multiples of 7 while the remainder may have arbitrary real values.

The solution to this problem is based on the following algorithm for solving Problem MI. (We assume without loss of generality that $G = (V, V_I, E, a)$ is strongly connected.)

ALGORITHM U.

U1. **if** (V, E, a) contains a negative-weight cycle **then** *Fail*
else foreach $(i, j) \in V_I \times V_I$ **do**
$b_{ji} \leftarrow$ [the least path weight from i to j in (V, E, a)];

U2. **if** $(V_I, V_I \times V_I, b)$ contains a negative-weight cycle **then** *Fail*
else find an integer assignment x on V_I such that $x_j - x_i \leq b_{ij}$ for all $i, j \in V_I$;

U3. Apply the Bellman–Ford algorithm to (V, E_R, a) using the x_i found in step U2 as initial values for the integer vertices and infinite initial values for the real vertices;

Step U1 produces a graph $H = (V_I, V_I \times V_I, b)$ which is feasible if and only if G is feasible. Step U2 solves H if H is feasible, and step U3 extends the solution from the set V_I of integer vertices to the entire vertex set V. Step U1 can be performed in $O(|V|^3)$ time by the Floyd–Warshall algorithm [8] or in $O(|V||E| + |V_I||V|\lg|V|)$ time by Fredman and Tarjan's improved version [4] of Johnson's algorithm [7]. Step U2 can be performed by the Bellman–Ford algorithm and takes time $O(|V_I|^3)$ because H is a complete graph. The cost of step U1 dominates the cost of step U3, which takes only $O(|V||E_R|)$ time.

Algorithm U extends naturally to the case in which there are multiple classes of periodic constraints, provided that each period (*e.g.*, 1 week) is an exact multiple of the next smaller period (*e.g.*, 1 day). First, step U1 is applied (with an appropriate scaling of the edge weights) to produce an equivalent problem in which the most loosely constrained class of vertices in the original problem is eliminated from consideration. This new problem is then solved recursively (or by direct application of Algorithm T if only two classes of vertices remain). Finally, the solution is extended to the entire set of vertices, as in step U3.

ACKNOWLEDGMENTS

We acknowledge the contributions by Flavio Rose of MIT when we first studied this problem. The three of us originally produced Algorithm U, which is more thoroughly described in Rose's master's thesis [12]. Thanks to Alex Ishii and Ron Rivest of MIT for reading drafts of the paper. Thanks also to Don Johnson of Penn State, Dick Karp of Berkeley, Gene Lawler of Berkeley, and Nimrod Megiddo of CMU for helpful discussions.

REFERENCES

1. A. V. AHO, J. E. HOPCROFT, AND J. D. ULLMAN, "*Data Structures and Algorithms*," Addison–Wesley, Reading, MA, 1983.
2. E. W. DIJKSTRA, A note on two problems in connexion with graphs, *Numer. Math.* **1** (1959), 269–271.
3. J. EDMONDS AND R. M. KARP, Theoretical improvements in algorithmic efficiency for network flow problems, *J. Assoc. Comput. Mach.* **19**, No. 2 (1972), 248–264.
4. M. L. FREDMAN AND R. E. TARJAN, Fibonacci heaps and their uses in improved network optimization algorithms, *in* "*Proceedings, 25th Annual IEEE Symposium on Foundations of Computer Science*," October, 1984, pp. 338–346.
5. M.- Y. HSUEH, "Symbolic Layout and Compaction of Integrated Circuits," Memorandum No. UCB/ERL M79/80, University of California, Berkeley, December 1979.
6. D. B. JOHNSON, Priority queues with update and finding minimum spanning trees, *Inform. Process. Lett.* **4**, No. 3 (1975), 53–57.
7. D. B. JOHNSON, Efficient algorithms for shortest paths in sparse networks, *J. Assoc. Comput. Mach.* **24**, No. 1 (1977), 1–13.

8. E. L. LAWLER, "*Combinatorial Optimization: Networks and Matroids*," Holt, Rinehart & Winston, New York, 1976.
9. C. E. LEISERSON, F. M. ROSE, AND J. B. SAXE, Optimizing synchronous circuitry by retiming, *in* "*Third Caltech Conference on Very Large Scale Integration*," (Randal Bryant, Ed.), pp. 87–116, Computer Sci. Press, Rockville, MD, 1983.
10. T. LENGAUER, On the solution of inequality systems relevant to IC-layout, *in* "*Proceedings, 8th Conference on Graphtheoretic Concepts in Computer Science*," Hanser Verlag, Munich, 1982.
11. R. F. LYON, A bit-serial VLSI architectural methodology for signal processing, *in* "*VLSI '81*" (J. P. Gray, Ed.), pp. 131–140, Academic Press, New York, 1981.
12. FLAVIO M. ROSE, "*Models for VLSI Circuits*," Masters thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, March 1982; MIT VLSI Memo 82-114.
13. J. Y. YEN, An algorithm for finding shortest routes from all source nodes to a given destination in general networks, *Quart. Appl. Math.* **27**, No. 4 (1970), 526–530.