A CONTINUOUS-TIME NETWORK FLOW PROBLEM ARISING FROM
MACHINE SCHEDULING: AN ALGORITHM

by

Dev Joneja\textsuperscript{1}

and

Robin Roundy\textsuperscript{2, 3}

\textsuperscript{1} Graduate School of Business, Columbia University, New York NY 10027
\textsuperscript{2} School of Operations Research and Industrial Engineering, Cornell University, Ithaca NY 14850
\textsuperscript{3} This research was supported by NSF Grant DMC-8451984 and also by AT&T Information Systems and the DuPont Corporation.
A CONTINUOUS-TIME NETWORK FLOW PROBLEM ARISING FROM MACHINE SCHEDULING:

AN ALGORITHM

by Dev Joneja and Robin Roudy

Section 1. Introduction and Review of the Literature.

In this paper we develop a linear-time algorithm for solving a continuous-time maximum network flow problem that arises from a scheduling problem. The algorithm relies heavily on the structure of the continuous-time network. The network in question can be viewed as a continuous-time version of a tree with arbitrarily directed edges. Each node in the tree is mapped into a finite interval of real numbers, and non-negative amounts of material are allowed to flow from left to right along these intervals. Each arc \((m,n)\) in the tree is mapped into a continuum of arcs, each of which connects the intervals that correspond to nodes \(m\) and \(n\) at a different point.

Fixed exogenous supplies and demands are distributed along the real intervals. The left end of each interval is a source from which any non-negative amount of material of material can emanate, and the right end of each interval is a sink which can absorb any nonnegative amount of material. The goal is to find a feasible flow in which the total amount of material that emanates from the sources is minimal.

The algorithm is first presented in a manner which makes no specific assumptions regarding the manner in which the supplies and demands are distributed along the real intervals. The dual of the network flow problem is derived, and the the finiteness and accuracy of the algorithm are proven. We then assume that the distribution of the supplies and the demands along the real intervals is piecewise constant, and show that the algorithm can be implemented so that it runs in time which is linear in the amount of information required to specify the optimal flows that are computed by the algorithm. This in turn is shown to be at most \(O(NB)\) where \(N\) is the number of nodes in the tree and \(B\) is the amount of input data.

The algorithm discussed in this paper was developed for use in an interactive system for job shop scheduling. The application is briefly discussed in the appendix.

Review of the Literature.

A variety of continuous generalizations of Ford and Fulkerson's (1962) classical max-flow, minimum-cut theorem for maximum network flows in discrete networks have been proven. Several of the proofs make use of abstract duality theory (see Strang (1983), Newmann (1984) and Ogier (1985)). A recent proof by Philpott (1990) uses flow augmenting paths in a proof that is
analogous to Ford and Fulkerson's, although it is necessarily much more complex. An augmenting
path algorithm has been developed (see Philpott (1982) and Anderson et al (1982)), and it
apparently works quite well when the arc capacities are sufficiently well behaved, but no sufficient
conditions to guarantee that it will compute an optimal solution have appeared. Hajek & Ogier
(1984) developed an efficient algorithm for a continuous-time minimum-cost network flow problem
with special structure that has application to the control of communications networks. However
their algorithm assumes that the arc capacities are constant in time, a serious limitation for a
continuous-time maximum-flow problem.

Overview of the Paper.

In section 2 we formulate the continuous-time network flow problem and summarize the
notation used. In section 3 we discuss the dual problem and its relationship to cuts, prove weak
duality, and give necessary and sufficient (complementary slackness) conditions for solutions to the
primal and dual problems to be optimal. We also discuss related results in the literature. Section 4
presents the algorithm and proves that it is finite and computes optimal primal and dual solutions.
Section 5 specializes the algorithm to the case of piece-wise constant demands and supplies and does
an analysis of running times. In section 6 we draw our conclusions. The appendix discusses the
application of the algorithm to job shop scheduling.

Section 2. Problem Formulation and Preliminaries.

The deterministic scheduling problem that we are interested in can be formulated as a
continuous-time, mixed integer linear programming problem as follows. We are given a network
\( \mathcal{G} = (\mathcal{N}, \mathcal{A}) \). Nodes in \( \mathcal{N} \) are denoted by the letters \( m, n, k \) and an arc in \( \mathcal{A} \) from \( m \) to \( n \) is denoted
by \( (m, n) \). \( \mathcal{G} \) is a tree with directed edges — it has no (directed or undirected) cycles.

We assume the existence of a class \( \mathcal{C} \) of real-to-real functions that have the following
properties.

Property I. The class \( \mathcal{C} \) contains the constant functions \( f(t) = a \) where \( a \in \mathbb{R} \).

Property II. The class \( \mathcal{C} \) is closed under addition and subtraction.

Property III. If \( f(t) \in \mathcal{C} \) and \( I \) is an interval then \( g(t) = \begin{cases} f(t), & t \notin I \\ 0, & t \in I \end{cases} \) is in \( \mathcal{C} \).

Property IV. On any finite (open or closed) interval, each function in \( \mathcal{C} \) is of bounded variation.

Property V. If \( f(t) \in \mathcal{C} \), if the sequence \( \{t_1, t_2, \ldots\} \) is either increasing or decreasing, and if
\( f(t_k) > \max(f(t_{k-1}), f(t_{k+1})) \) for all even \( k \), then \( |t_k| \to \infty \) as \( k \to \infty \).
Functions in \( C \) are called *regular functions*. The piecewise polynomial functions that have a finite number of pieces in any finite real interval are regular. Note that if \( f \) and \( g \) are regular functions, \( f \) is right-continuous, and \( g \) is left-continuous then the Stieltjes integrals \( \int_a^b f(x) \, dg(x) \) and \( \int_a^b g(x) \, df(x) \) exist for all finite intervals \([a, b]\). Also note that Properties II and V imply that any two functions in \( C \) cross each other at most a finite number of times on any finite (open or closed) interval.

We assume that for each \( n \in \mathbb{N} \) we are given a real-valued function \( s(n,t) \) defined for \( t \in [0, T] \). We assume that the functions \( s(n,t) \) are continuous, regular, and satisfy \( s(n,0) = 0 \). The continuous-time network flow problem that we wish to solve is as follows.

\[
\begin{align*}
(P) \quad \text{min:} \quad & \sum_n [x(n,0) + x(n,T)] \\
\text{st:} \quad & s(n,t) + \sum_{(m,n) \in A} y(m,n,t) - \sum_{(n,k) \in A} y(n,k,t) = x(n,t) - x(n,0) \\
& \forall \ n \in \mathbb{N}, \ t \in [0,T]. \quad (1) \\
& y(m,n,0) = 0, \text{ and } y(m,n,t) \in C \text{ is a non-decreasing right-continuous function of} \\
& \quad t \in [0,T], \forall \ (m,n) \in A. \quad (2) \\
& x(n,t) \in C \text{ is a non-negative, right-continuous function of } t \in [0,T], \forall \ n \in \mathbb{N}. \quad (3)
\end{align*}
\]

Summing (1) over all \( n \in \mathbb{N} \) we have \( \sum_n [s(n,t) - x(n,t)] = -\sum_n x(n,0) \). Noting that \( s(n,t) \) is continuous and that \( s(n,0) = 0 \) we see that \( \sum_n x(n,t) \) must be continuous at \( t = 0 \). Therefore the restrictions that \( x(n,t) \) and \( y(m,n,t) \) are right-continuous do not alter the optimal value of \( (P) \). They are made for technical reasons only.

Problem \( (P) \) can be interpreted as a network flow problem in The Continuous-Time Network, defined as follows (see Figure 1). For each node \( n \) of \( G \) we create a real time interval from time \( t = 0 \) to time \( t = T \), called *bar* \( n \). We refer to the direction in which time decreases as the *left* and to the direction in which time increases as the *right*. Fixed supplies and demands are distributed along the bars. For bar \( n \) we assume that the total net supply between time \( q \) and time \( r \) is \( s(n,r) - s(n,q) \). This number is interpreted as a supply if it is positive, and as a demand if it is negative.

For each arc \( (m,n) \) of \( G \) we create a continuum of arcs, one of which leads from bar \( m \) to bar \( n \) at each point in time \( t, 0 < t < T \). Any non-negative amount of material is allowed to flow from left to right along the bars and along any of the arcs that connect the different bars. The flows along the bars are called *horizontal flows*, and they are denoted by the functions \( x(n,t) \). The flows between bars are called *vertical flows*, and the cumulative values of these flows are denoted by the variable \( y(m,n,t) \).
Conservation of flow must hold along all of the bars at all times $t$, $0 < t < T$. However at time $t = 0$ any nonnegative amount of material can enter bar $n$, and at time $t = T$ any nonnegative amount of material can leave bar $n$, for all $n \in \mathcal{N}$. The goal is to meet all of the demand, while minimizing the amount of material that enters the network from the left and exits from the right.

Figure 1: The Continuous Time Network.

The Network $\mathcal{G}$.

The Continuous Time Network.

We conclude this section by summarizing our notation.

Notation

$\mathcal{G}$ : The Network

$\mathcal{N}$ : The nodes of $\mathcal{G}$. Nodes are denoted by the letters $m, n, k$.

$\mathcal{A}$ : The arcs of $\mathcal{G}$. Arcs are denoted by ordered pairs, i.e., $(m, n)$. 

4
\( N = |N| \) : The number of nodes in \( N \).

\( \pi(n) \) : The direct predecessor of node \( n \), i.e., the node adjacent to \( n \) on the path from \( n \) to node \( N \).

\( \sigma(n) \) : The set \( \{ m : n = \pi(m) \} \) of direct successors of \( n \). \( \sigma(n) = \emptyset \) if \( n \) is a leaf node and \( n \neq N \).

\( L = \{ n \in N : (n, \pi(n)) \in \mathcal{A} \} \) : The set of lower nodes.

\( U = \{ n \in N : (\pi(n), n) \in \mathcal{A} \} \) : The set of upper nodes.

\( t \) : Time.

\( T \) : The end of the time horizon. The horizon goes from time \( t = 0 \) to time \( t = T \).

\( s(n,t) \) : The net cumulative supply function for node \( n \in N \), defined on \( 0 \leq t \leq T \). An increase in \( s(n,t) \) indicates an exogenous supply and a decrease in \( s(n,t) \) indicates a demand.

\( C(n) \) : The time of the optimal cut for node \( n \). \( 0 \leq C(n) \leq T \).

\( y(m,n,t) \) : The cumulative flow from node \( m \) to node \( n \) between time \( 0 \) and time \( t \), \( 0 < t < T \), \( (m,n) \in \mathcal{A} \).

\( \mathcal{K}(n) \) : A list of disjoint time intervals during which there is positive horizontal flow at node \( n \).

\( x(n,t) \) : The horizontal flow at node \( n \) at time \( t \), \( 0 \leq t \leq T \).

Section 3. Duality and Cuts.

The dual of problem (P) can be written as follows.

\[
\text{(D)} \quad \max \quad \sum_n \int_0^T z(n,t) \, ds(n,t) \\
\text{st:} \quad z(m,t) - z(n,t) \leq 0 \quad \forall \quad (m,n) \in \mathcal{A}, \quad \forall \quad t \in [0,T]. \quad (4)
\]

\( z(n,t) \) is a non-decreasing left-continuous function of \( t \in [0,T] \), \( \forall \quad n \in N \). \quad (5)

\[-z(n,0) \leq 1 \quad \forall \quad n \in N. \quad (6)
\]

\[z(n,T) \leq 1 \quad \forall \quad n \in N. \quad (7)
\]

Since \( s(n,t) \) is continuous the technical assumption that \( x(n,t) \) is left-continuous has no effect on the optimal value of (D). Heuristically we derive (D) from (P) by assuming that \( \delta \) is a small positive number, by defining \( \Delta y(m,n,t) = y(m,n,t+\delta) - y(m,n,t) \), and by defining \( \Delta s(n,t) \) similarly. We write (1) as
\[ \sum_{(n,k) \in \mathcal{A}} \Delta y(n,k,t) - \sum_{(m,n) \in \mathcal{A}} \Delta y(m,n,t) + x(n,t+\delta) - x(n,t) = \Delta s(n,t) \]

\[ \forall \ n \in \mathcal{N}, \ t \in [0,T-\delta]. \quad (8) \]

Mimicking the conventional approach for constructing duals of standard linear programming problems, the dual variable corresponding to (8) is \( z(n,t) \). The dual constraint corresponding the the primal variable \( \Delta y(m,n,t) \) is (4). For \( 0 < t < T \), the dual constraint corresponding to the primal variable \( x(n,t) \) is \( z(n,t-\delta) - z(n,t) \leq 0 \), which leads to (5). The dual constraints corresponding to the primal variables \( x(n,0) \) and \( x(n,T) \) are (6) and (7), respectively.

**Theorem 1 (Weak Duality and Complementary Slackness Conditions).** Suppose that \( x, y \) is a feasible solution to (P) and that \( z \) is a feasible solution to (D). Then \( \sum_n \int_0^T z(n,t) \ ds(n,t) \leq \sum_n \left[ z(n,T) + x(n,0) \right] \), i.e., weak duality holds. Furthermore the duality gap is zero, \( x \) and \( y \) are optimal for (P), and \( z \) is optimal for (D) if the following complementary slackness conditions hold:

\[ \sum_n \int_0^T z(n,t) \ dz(n,t) = 0, \quad (9) \]

\[ \sum_n (1 - z(n,T)) \ x(n,T) = 0, \quad (10) \]

\[ \sum_n (1 + z(n,0)) \ x(n,0) = 0, \quad \text{and} \]

\[ \sum_{(m,n)} \int_0^T [z(n,t) - z(m,t)] \ dy(m,n,t) = 0. \quad (12) \]

**Proof.** The second line below follows from (2) and (4), the fourth line follows from (1) and the integration by parts formula, and the fifth line follows from (3), (5), (6) and (7).

\[ \sum_n \int_0^T z(n,t) \ ds(n,t) \leq \]

\[ \sum_n \int_0^T z(n,t) \ ds(n,t) + \sum_{(m,n)} \int_0^T [z(n,t) - z(m,t)] \ dy(m,n,t) = \]

\[ \sum_n \int_0^T z(n,t) \ ds(n,t) + \sum_n \sum_{k} y(n,k,t) - \]}

\[ \sum_n \int_0^T \sum_n \left[ z(n,t) \ x(n,t) \ [z(n,T) - z(n,0)] \right] - \sum_n \int_0^T \sum_k \ x(n,t) \ dz(n,t) \leq \]

\[ \sum_n \ [z(n,T) - z(n,0)] \ x(n,0) \]
This proves weak duality. Clearly the first line above is equal to the last line if and only if (9)–(12)
hold. □

We define a cut in The Continuous-Time Network to be set of real numbers \(C(n)\), \(n \in \mathbb{N}\).
Each cut gives rise to a solution \(z\) for (D) through the formula
\[
z(n,t) = \begin{cases} 
-1 & \text{for } t \leq C(n) \text{ and } z(n,t) = 1 & \text{for } t > C(n).
\end{cases}
\]

A cut is feasible if the corresponding \(z\) is feasible for (D). We define the capacity of a cut to be the
corresponding value of the objective function of (D). Note that if \(z\) corresponds to a feasible cut, the
complementary slackness conditions (9)–(13) state that no flow moves from a source or from a location
on a bar at which \(z(n,t) = -1\), to a sink or to a location on a bar at which \(z(n,t) = 1\).

Because the objective of (D) is a linear function of \(z\), there is an optimal solution of (D) that is
an extreme point of the feasible region of (D). The following lemma states that the extreme points of
(D) are the points that correspond to feasible cuts.

**Lemma.** A feasible solution \(z\) of (D) is an extreme point of the feasible region of (D) if and only if \(z\)
corresponds to a feasible cut via (13).

**Proof.** If (13) holds then \(z\) is clearly an extreme point. Let \(z\) be a feasible solution to (D). Define
\(w(n,t)\) by
\[
w(n,t) = \begin{cases} 
z(n,t) + 1, & z(n,t) \leq 0 \\
1 - z(n,t), & z(n,t) > 0.
\end{cases}
\]

Then
\[
z(n,t) + \alpha w(n,t) = \begin{cases} 
(1+\alpha)z(n,t) + \alpha, & z(n,t) \leq 0 \\
(1-\alpha)z(n,t) + \alpha, & z(n,t) > 0.
\end{cases}
\]

This function is feasible for (D) for \(-1 \leq \alpha \leq 1\), so \(z(n,t)\) is an extreme solution for (D) only if
\(w(n,t) = 0\) for all \(n\) and for all \(t\). The result follows from (5). □

Theorem 1 shows that the value of any feasible cut is greater than or equal to the value of any
feasible flow. In Section 4 we construct feasible solutions to (P) and (D) that satisfy (9)–(13), thus
proving strong duality and the existence of an optimal cut.

Anderson and Nash (1987) developed a duality theory for continuous-time maximum-flow
problems (also see Anderson et al (1982)). They formulate the problem in terms of the derivatives of
\(g(m,n,t)\) with respect to the time, and assume that those derivatives belong to \(L_\infty[0,T]\). (P) can be
formulated as a special case of their maximum flow problem if the functions \(g(m,n,t)\), \((m,n) \in \mathcal{A}\) have
the required properties. Anderson and Nash prove that their maximum-flow problem has a solution
and that dual solutions correspond to generalized cuts in a natural fashion, and they prove a maximum-flow, infimum-cut theorem. Philpott (1990) proves a max-flow, min-cut theorem, but he does not formulate a dual problem analogous to (D).

Earlier researchers face two obstacles that we have managed to avoid. The first obstacle is that the optimal cut may have an infinite number of “switches” along a bar. This would be analogous to having the function \( z(n,t) \), \( t \in [0,T] \), oscillate between -1 and 1 an infinite number of times for a given \( n \). We do not have this problem because the horizontal flows \( z(n,t) \) along the bars do not have upper bounds, so \( z(n,t) \) is non-decreasing. A related obstacle is that the optimal flows \( \frac{d}{dt} y(m,n,t) \) can have an infinite number of discontinuities, probably making it impossible to compute or even to represent them. Our assumption that the supply and demand functions \( s(n,t) \) are regular was devised to circumvent this obstacle. These obstacles seem to account for the fact that the augmenting path algorithms of Philpott (1982, 1990) and of Anderson et al (1982) have not been shown to converge in general. We conjecture that an assumption much like regularity will make the augmenting path algorithms of earlier researchers, or algorithms similar to them, finite.

Section 4. Solving (P).

In view of Theorem 1, our goal is to compute functions of time \( x, y, z \) such that (1)–(7) and (9)–(13) hold. The algorithm that follows computes flow functions \( y(m,n,t) \) and \( z(n,t) \) that are continuous, and a cut \( C(n) \) from which \( z \) can be constructed using (13).

We assume that the nodes of \( G \) are numbered with node \( N \) being the root node and node \( \pi(n) > n \) being the next node after node \( n \) in the unique path from \( n \) to the root. \( \sigma(n) = \{ m : n = \pi(m) \} \) is the set of successor nodes of \( n \). Let \( L = \{ n \in N : n \neq N, (n, \pi(n)) \in A \} \) be the set of lower nodes, and let \( U = \{ n \in N : n \neq N, (\pi(n), n) \in A \} \) be the set of upper nodes.

The horizontal flows \( z(n,t) \) along bar \( n \) and the vertical flows that connect \( n \) and \( \pi(n) \) are associated with bar \( n \). We calculate flows one bar at a time, in order, starting with the flows associated with bar 1 ending with the flows associated with bar \( N \). Before calculating the flows associated with bar \( n \) we will know the vertical flows that connect bar \( n \) to bars \( m, m \in \sigma(n) \). Once determined, these vertical flows have the effect of creating additional supplies and/or demands on bar \( n \). The sum of these flows and of the exogenous supplies and/or demands that occur along bar \( n \) can be interpreted as a total net supply/demand function for bar \( n \). This total net supply/demand function is given by

\[
S(n,t) = s(n,t) + \sum_{m \in \sigma(n) \cap L} y(m,n,t) - \sum_{k \in \sigma(n) \cap U} y(n,k,t) \quad \forall \quad n, t. \quad (14)
\]
We then compensate for the supplies and demands in \( \mathcal{S}(n,t) \) by creating the flows associated with bar \( n \), namely, the horizontal flows \( x(n,t) \) and the vertical flows that connect \( n \) and \( \pi(n) \).

Using (14) we can re-write the conservation of flow constraint (1) as follows.

\[
\begin{align*}
    x(n, t) - y(\pi(n), n, t) &= \mathcal{S}(n, t) + z(n, 0) \quad \forall \quad n \in U, \quad \forall \quad t \in [0, T], \\
    x(N, t) &= \mathcal{S}(N, t) + x(N, 0) \quad \forall \quad t \in [0, T]; \\
    x(n, t) + y(n, \pi(n), t) &= \mathcal{S}(n, t) - \mathcal{S}(n, T) + z(n, T) + y(n, \pi(n), T) \quad \forall \quad n \in L, \quad t \in [0, T], \quad \text{and} \\
    y(n, \pi(n), 0) &= 0 \quad \forall \quad n \in L.
\end{align*}
\]  

(15)  

The following algorithm computes optimal solutions for (P) and (D). It calls three subroutines, **Merge**, FlowForward, and FlowBackward.

**The Network Flow Algorithm**

Input: The network \( \mathcal{G} \) and the functions \( s(n,t) \quad \forall \quad n \in \mathcal{N} \).

Output: The functions \( x(n,t) \) and \( y(m,n,t) \), and \( C(n) \), \( \forall \quad n \in \mathcal{N} \).

**Step 1: Compute the Flows.**

For \( n \leftarrow 1 \) to \( N-1 \) do

**Merge**

- Input: \( s(n,t) \cup \{y(m,n,t) : m \in \sigma(n) \cap L\} \cup \{-y(n,k,t) : k \in \sigma(n) \cap U\} \);
- Output: \( S(n,t) \).

If \( n \in U \) then **FlowForward**

- Input: \( n, S(n,t) \); Output: \( x(n,t), y(\pi(n), n, t), \mathcal{H}(n) \).

If \( n \in L \) then **FlowBackward**

- Input: \( n, S(n,t) \); Output: \( x(n,t), y(n, \pi(n), t), \mathcal{H}(n) \).

End

**Merge**

- Input: \( s(S(n,t)) \cup \{y(m,N,t) : m \in \sigma(N) \cap L\} \cup \{-y(N,k,t) : k \in \sigma(N) \cap U\} \);
- Output: \( S(N,t) \).

\[
R = \min \{S(N,t) : 0 \leq t \leq T\},
\]

\( C(N) = \) a value of \( t \) that achieves the minimum above, and

\[
x(N,t) = -R + S(N,t) \quad \text{for all} \quad 0 \leq t \leq T.
\]

**Step 2: Compute the Cut for \( n < N \).**

For \( n \leftarrow N-1 \) down to 1 do

If \( n \in L \) then \( C(n) = \inf \{t : t \geq C(\pi(n)) : t \notin I \text{ for any interval } I \in \mathcal{H}(n)\} \).

If \( n \in U \) then \( C(n) = \sup \{t : t \leq C(\pi(n)) : t \notin I \text{ for any interval } I \in \mathcal{H}(n)\} \).

End.
Step 2 can be implemented by simply scanning through $\mathcal{H}(n)$ from left to right for each $n$. Assuming that $\mathcal{H}(n)$ is represented in list format, the time required to do this is less that the time required to create $\mathcal{H}(n)$ and the running time will not be constrained by Step 2. The procedure **Merge** computes the sum of the functions of time that are fed to it as arguments. The way in which it is implemented depends on the manner in which the functions of time are represented.

**Merge**

Input: A list of functions.
Output: The sum of the functions in the input list.

[The computations are omitted]

Procedure FlowForward and Procedure FlowBackward compensate for the supplies and demands in $S(n,t)$ by creating horizontal flows $\pi(n,t)$ and vertical flows that connect $n$ and $\pi(n)$. Procedure FlowForward is used for upper nodes.

Consider an upper node $n$. Procedure FlowForward traverses bar $n$ from left to right. If we encounter a supply in $S(n,t)$, the only thing that can be done with it is to accumulate it as we advance in the form of a positive horizontal flow $\pi(n,t)$. If we encounter a demand on bar $n$ and an accumulated supply is available in the form of a positive horizontal flow we will use that supply to meet the demand that we encounter as we advance. This continues until either we reach the end of the bar, or we encounter a demand on bar $n$ and no accumulated supply is available to meet it. Step 2 of Procedure FlowForward implements this portion of the algorithm.

Suppose that we encounter a demand on bar $n$ and no accumulated supply is available (i.e., $\pi(n,t) = 0$). The demand can be met by creating a vertical flow from $\pi(n)$ to $n$ at some time $s \leq t$ and a horizontal flow along bar $n$ from time $s$ to time $t$. The demand can also be met by creating a horizontal flow along bar $n$ from time $0$ to time $t$. The second option increases $\pi(n,0)$, which impacts unfavorably on the objective function of (P). The first option is preferable because it allows for the possibility that we might be able to find a supply at $\pi(n)$ or at some other node that we could use to satisfy this demand, without having an impact on the objective function of (P). If this option is adopted, creating the vertical flow from $\pi(n)$ to $n$ at time $t$ rather than at some time $s < t$ will increase the likelihood that a supply at some other node can be found to satisfy this demand.

Therefore, whenever we encounter a demand on bar $n$ that cannot be met from accumulated supply, Procedure FlowForward satisfies this demand by creating vertical flows from $\pi(n)$ to $n$. This continues until we encounter a supply, or until we reach the end of the bar. Step 3 of Procedure FlowForward implements this phase of the computation.
The logic of the preceding paragraphs indicates why this algorithm requires that the network $G$ be free of cycles. The Network Flow Algorithm constructs optimal horizontal flows for bar $n$ and optimal vertical flows between bar $n$ and bar $\pi(n)$ without observing $s(\pi(n), t)$. If $G$ has a cycle then we will be forced to consider nodes $n$ that have more than one predecessor, i.e., $|\pi(n)| > 1$. Suppose that $|\pi(n)| > 1$ and that all arcs in $G$ connecting $n$ to a node in $\pi(n)$ are oriented into node $n$. As we advance from left to right along bar $n$ we encounter a demand that cannot be met from accumulated supply. The logic of the preceding paragraph implies that we prefer to meet this demand by creating a vertical flow into bar $n$ at time $t$. However without observing $s(m, t)$, $m \in \pi(n)$ we cannot know which of the nodes in $\pi(n)$ we should select as the source of that flow. If $G$ has cycles it appears impossible to mimic The Network Flow Algorithm by determining optimal flows one bar at a time.

**Procedure FlowForward**

Input: $n$, $\mathcal{S}(n, t)$ where $u \in U$.

Output: $x(n, t)$, $y(\pi(n), n, t)$, $\mathcal{H}(n)$.

**Step 1:** Initialize.

1. $\mathcal{H}(n) \leftarrow \emptyset$; $w = 0$; $\pi(n, 0) = 0$; $y(\pi(n), n, 0) = 0$.

2. If $\mathcal{S}(n, \epsilon) > 0$ for all sufficiently small $\epsilon > 0$ then go to Step 2. Otherwise go to Step 3.

**Step 2:** Create Horizontal Flows at Node $n$.

*Comment: We are at a point in time $w$ at which $\mathcal{S}(n, w+\epsilon) > \mathcal{S}(n, w)$ for all sufficiently small $\epsilon > 0$, and $x(n, w) = 0$. As we move right we will accumulate the excess supply as a horizontal flow $x(n, t)$ until either $x(n, t)$ becomes negative or until we reach $t = T$.)*

3. $u \leftarrow \sup \{v : w \leq v \leq T$ and $\mathcal{S}(n, t) \geq \mathcal{S}(n, w)$ for all $w \leq t \leq v\}$.

4. $x(n, t) \leftarrow \mathcal{S}(n, t) - \mathcal{S}(n, w)$ and $y(\pi(n), n, t) \leftarrow y(\pi(n), n, w)$ for all $w \leq t \leq u$.

5. Add the interval $(w, u)$ to $\mathcal{H}(n)$ if $u < T$, and add the interval $(w, T]$ to $\mathcal{H}(n)$ if $u = T$.

6. $w \leftarrow u$.

7. If $u = T$ then go to Step 4.

**Step 3:** Create Vertical Flows at Node $n$.

*Comment: We are at a point in time $w$ at which $x(n, w) = 0$ and $\mathcal{S}(n, t)$ is non-increasing for all sufficiently small $t \geq w$. As we move right we will satisfy the demands that we encounter by creating a vertical flow from $\pi(n)$ to $n$.)*

8. $u \leftarrow \sup \{v : w \leq v \leq T \text{ and } \mathcal{S}(n, t) \text{ is non-increasing on the interval } [w, v]\}$.

9. $y(\pi(n), n, t) \leftarrow y(\pi(n), n, w) + \mathcal{S}(n, w) - \mathcal{S}(n, t)$ and $x(n, t) \leftarrow 0$ for all $w \leq t \leq u$.

11
$w \leftarrow u.$

If $u < T$ then go to Step 2.

Step 4: End.

End of Procedure FlowForward.

Procedure FlowBackward is the equivalent of Procedure FlowForward for lower nodes. The main differences between the two procedures are as follows. In Procedure FlowBackward we traverse bar $n$ from right to left rather than from left to right. Any demands that we encounter can only be satisfied by horizontal flows, so we accumulate them in the form of horizontal flows as we progress. The supplies that we encounter are used to satisfy previously accumulated demands if there are any, and they are passed to node $\pi(n)$ for possible use elsewhere in the network if there are no accumulated demands.

Procedure FlowBackward

Input: $u, S(n, t)$ where $u \in L$.
Output: $x(n, t), y(n, \pi(n), t), \mathcal{H}(n)$.

Step 1: Initialize.

1. $\mathcal{H}(n) \leftarrow \emptyset; w = T; x(n, T) = T; y(n, \pi(n), T) = 0.$
2. If $S(n, T - \epsilon) > S(n, T)$ for all sufficiently small $\epsilon > 0$, go to Step 2. Otherwise go to Step 3.

Step 2: Create Horizontal Flows at Node $n$.

(Comment: We are at a point in time $w$ at which $x(n, w) = 0$ and $S(n, w - \epsilon) > S(n, w)$ for all sufficiently small $\epsilon > 0$. As we move left we will accumulate the excess demand as a horizontal flow $x(n, t)$ until either $x(n, t)$ becomes negative or until we reach $t = 0$.)

3. $u \leftarrow \inf \{v : 0 \leq v \leq w \text{ and } S(n, t) \geq S(n, w) \text{ for all } v \leq t \leq w\}.$
4. $x(n, t) \leftarrow S(n, t) - S(n, w) \text{ and } y(n, \pi(n), t) \leftarrow y(n, \pi(n), w)$ for all $u \leq t \leq w$.
5. Add the interval $(u, w)$ to $\mathcal{H}(n)$ if $u > 0$, and add the interval $[0, w)$ to $\mathcal{H}(n)$ if $u = 0$.
6. $w \leftarrow u$.
7. If $u = 0$ then go to Step 4.

Step 3: Create Vertical Flows at Node $n$.

(Comment: We are at a point in time $w$ at which $x(n, w) = 0$ and $S(n, t)$ is non-decreasing for all sufficiently large $t \leq w$. As we move left we will send the supplies that we encounter to node $\pi(n)$ in the form of a vertical flow.)
\begin{align*}
8 \quad & u \leftarrow \inf \{ v : 0 \leq v \leq w \text{ and } S(n, t) \text{ is non-decreasing on the interval } [u, w] \}. \\
9 \quad & y(n, \pi(n), t) \leftarrow y(n, \pi(n), w) + S(n, t) - S(n, w) \text{ and } z(n, t) \leftarrow 0 \text{ for all } u \leq t \leq w. \\
10 \quad & w \leftarrow u. \\
11 \quad & \text{If } u > 0 \text{ then go to Step 2.}
\end{align*}

Step 4: End.

Set \( y(n, \pi(n), t) \leftarrow y(n, \pi(n), t) - y(n, \pi(n), 0) \) for all \( 0 \leq t \leq T \).

12 \quad \text{End of Procedure FlowBackward.}

We use the term Flow Procedures to refer to Procedure FlowForward and Procedure FlowBackward.

Theorem 2. The Network Flow Algorithm is finite. It calculates optimal solutions to (P) and (D).

\textbf{Proof.} In light of Theorem 1, it suffices to show that the algorithm is finite and that it computes flows \( x, y \) and a dual solution \( z \) that satisfies equations (1)–(7) and (9)–(12). We begin by proving two claims.

\textbf{Claim 1:} If \( S(n, t) \) is right-continuous and regular for \( t \in [0, T] \) then the statement

\begin{equation}
\text{There is an } \epsilon > 0 \text{ such that } S(n, t) \text{ is non-increasing for } t \in [w, w+\epsilon]
\end{equation}

is equivalent to

\begin{equation}
\text{For all } \delta > 0 \text{ there is an } \epsilon, 0 < \epsilon \leq \delta, \text{ such that } S(n, w+\epsilon) \leq S(n, w),
\end{equation}

and (17) fails if and only if

\begin{equation}
S(n, w+\epsilon) > S(n, w) \text{ for all sufficiently small } \epsilon > 0.
\end{equation}

\textbf{Proof of Claim 1:} Clearly (17) implies (18), and (18) holds if and only if (19) fails. To see that (18) implies (17), suppose that (18) holds and (17) fails. Since (17) fails there is a strictly decreasing sequence \( \{t_1, t_2, \ldots\} \) that converges to \( w \) and that satisfies \( S(n, t_{k-1}) > S(n, t_k) \) for all even \( k \). If there is no \( M \) such that \( S(n, t_k) \) is non-increasing in \( k \) for all \( k \geq M \) then it is possible to select an infinite subsequence of \( \{t_1, t_2, \ldots\} \) of this sequence that satisfies \( S(n, t^*_{k-1}) > S(n, t^*_k) < S(n, t^*_{k+1}) \) for all even \( k \), a violation of Property V of regular functions. If there is an \( M \) such that \( S(n, t_k) \) is non-increasing in \( k \) for all \( k \geq M \) then \( S(n, t_k) > S(n, w) \) for all sufficiently large \( k \), and we can use (18) to generate a decreasing sequence \( \{t_1, t_2, \ldots\} \) that converges to \( w \) and that satisfies \( S(n, t^*_k) > S(n, w) \) for all even \( k \) and \( S(n, t^*_k) \leq S(n, w) \) for all odd \( k \), a violation of Property V of regular functions.

\textbf{End of Proof for Claim 1.}
Claim 2: Assume that \( n \in U \), that \( S(n,t) \) is continuous and regular, and that Procedure FlowForward is executed for \( n \). Then Procedure FlowForward will terminate after a finite amount of time, and whenever either Line 3 or Line 8 is about to be executed,

(a). \( 0 \leq w < T \) and

(b). \( x(n,w) = 0 \).

Whenever Line 3, Line 8 or Line 12 is about to be executed:

(c). \( x(n,t) \) is non-negative, continuous and regular for \( t \in [0,w] \).

(d). \( y(\pi(n),n,t) \) is non-decreasing, continuous and regular for \( t \in [0,w] \).

(e). (15) holds for \( t \in [0,w] \).

(f). If \( t \in [0,w] \) and \( t \notin I \) for any interval \( I \in \mathcal{I}(n) \) then \( x(n,t) = 0 \). If \( I \in \mathcal{I}(n) \) then \( y(\pi(n),n,t) \) is constant for \( t \in I \), and there is a \( t \in I \) such that \( x(n,t) > 0 \).

Furthermore,

(g). (17) is true whenever Line 8 is about to be executed, and (17) is false whenever Line 3 is about to be executed. Every time other than the first time that Line 8 is executed,

\[
\text{For all } \delta > 0 \text{ there is an } \epsilon, 0 < \epsilon \leq \delta, \text{ such that } S(n,w+\epsilon) < S(n,w). \tag{20}
\]

Proof of Claim 2: The proof of (a) – (g) is by induction on the times at which the lines in question are executed. The first time that one of Lines 3 and 8 is executed, Lines 1 and 2 clearly imply that (a) – (g) hold. Suppose that Line 3 is about to be executed and that the induction hypothesis holds. By (g) and Property V of regular functions, (19) holds. Therefore the set in Line 3 is not empty, and \( u \) \( > w \). Line 4 and (b) of the induction hypothesis imply that after Line 6 is executed, (c) – (f) will hold. If \( u < T \) then Lines 6, 7 imply that (a) will hold, the continuity of \( S(n,t) \) and Lines 3, 4 imply that (b) will hold, and Line 3 implies that (20) will hold, so (g) will hold.

Suppose that Line 8 is about to be executed and that the induction hypothesis holds. Then (g) implies that the set in Line 8 is not empty and that \( u > w \). Line 8 and (b) imply that after Line 10 is executed, (c) – (f) will hold. If \( u < T \) then Lines 8 and 11 imply that (a) will hold, Line 9 implies that (b) will hold, and Line 8 implies that (g) will hold. This ends the proof of (a) – (g).

Either Procedure FlowForward will terminate after a finite amount of time, or else Lines 3 and 8 are both executed an infinite number of times. In the latter case (g) implies that there is an infinite, strictly increasing sequence \( \{t_1, t_2, ... \} \) such that \( 0 \leq t_k < T \) for all \( k \), \( S(n,t_k) < S(n,t_{k+1}) \) if \( k = 4m + 1 \) for some integer \( m \), and \( S(n,t_k) > S(n,t_{k+1}) \) if \( k = 4m + 3 \) for some integer \( m \). This contradicts Property V of regular functions. End of Proof of Claim 2.
A result that is very similar to Claim 2 can be proven for \( n \in L \). The formal statement and proof of the claim are omitted. The statement of the claim is identical except that (a) becomes \( 0 < w \leq T \), and \([0,w]\) and \( y(\pi(n),u,t)\) are replaced by \([w,T]\) and \( y(n,\pi(n),t)\) wherever they appear, and \( w+\epsilon \) in equations (17)–(19) becomes \( w-\epsilon \). The proofs of the two claims are nearly identical, with the obvious changes caused by the fact that we are now moving backwards in time rather than forwards. For \( n = N \), (c) and (e) hold by Step 1 of the Network Flow Algorithm. Hereafter we will refer to properties (a)–(g) without differentiating between nodes in \( U \), nodes in \( L \), and node \( N \).

Claim 2 and the corresponding claim for upper nodes form the basis of a simple induction argument that proves the finiteness of Step 1 of the Network Flow Algorithm and the regularity and continuity of the functions \( S(n,t) \). Note that by Line 5 of the Flow Procedures,

\[
0 \notin I \text{ for any } I \in \mathcal{K}(u) \text{ if } u \in U, \quad \text{and} \quad T \notin I \text{ for any } I \in \mathcal{K}(u) \text{ if } u \in L.
\]

Therefore the sets in Step 2 of the Network Flow Algorithm are all non-empty and for all \( n \in N \),

\[
0 \leq C(n) \leq T \quad \text{and} \quad C(n) \notin I \text{ for any } I \in \mathcal{K}(u).
\]

This implies that

\[
\pi(n,C(n)) = 0 \quad \forall \ n \in N.
\] (21)

We now show that the flows and the cut computed by the algorithm are optimal by proving that (1)–(12) hold. Equation (15) holds by (c) of Claim 1, so Line 1 of the Flow Procedures implies (1). Equations (2) and (3) hold by (c) and (d). Equation (13) implies (5), (6) and (7). (13) also implies that (4) is equivalent to \( C(m) \geq C(n) \) for all \( (m,n) \in \mathcal{A} \), which holds by Step 2 of the Network Flow Algorithm.

This proves feasibility of the primal and dual solutions. We now show that the complementary slackness conditions hold. Equation (13) implies that equations (9)–(12) can be re-stated as follows.

\[
\sum_n \pi(n,C(n)) = 0, \quad (9^*)
\]

Either \( C(n) < T \) or \( \pi(n,C(n)) = 0 \) \( \forall \ n \),

\[
(10^*)
\]

Either \( C(n) > 0 \) or \( \pi(n,C(n)) = 0 \) \( \forall \ n \), and

\[
\gamma(m,n,C(n)) = \gamma(m,n,C(m)) \quad \forall \ (m,n).
\] (11*)

Equations (9*), (10*) and (11*) follow from (21). Step 2 of the Network Flow Algorithm implies that for each \( (m,n) \in \mathcal{A} \), either \( C(m) = C(n) \) or else the interval \((C(n),C(m))\) lies entirely within some interval \( I, I \in \mathcal{K}(n) \). This fact, (d) and (f) imply (12*). The result follows from Theorem 1. \( \square \)
This establishes the finiteness and the accuracy of the algorithm. The running time of the algorithm can not be completely analyzed without specifying the data structures that are used to represent the functions involved. However we can state that the number of times that any one of the lines of the algorithm is executed is at most linear in $\sum_n |\mathcal{K}(n)|$. In light of the fact that $|\mathcal{K}(n)|$ is essentially the number of times that $x(n, t)$ switches from positive to zero and from zero to positive (see (f)), it is probable that the total number of times that any line of the algorithm is executed is at most linear in the size of the data structures that are used to specify the flow variables $\{x(n, t); n \in N\}$.

We now give an example that illustrates the algorithm. The network $G$ is the network of Figure 1. Node 4 is the root. The input data and the computed flow functions are illustrated in Figure 2. All of the breakpoints of the piecewise linear functions in Figure 3 have integer coordinates. The key steps performed by The Network Flow Algorithm and by the Flow Procedures are summarized in Table 1.

| Table 1: An Example |

Procedure FlowForward, Node 1.
- Step 3: $u = 2$. Create increasing $g(3,1,t)$ and constant $x(1,t)$ for $t \in [0,2]$.

- Step 3: $u = 6$. Create non-decreasing $g(2,3,t)$ and constant $x(2,t)$ for $t \in [6,11]$.

Procedure FlowForward, Node 3.
- Step 2: $u = 8$. Create constant $g(2,3,t)$ and non-negative $x(2,t)$ for $t \in [0,6]$. $\mathcal{K}(2) = \{(0,6)\}$.

Step 1: $C(4) = 5$. (Any value in the interval [4,5] would do.)

Step 2: $C(3) = 4$. $C(2) = 6$. $C(1) = 2$. 
Figure 2. An Example

\[ \text{Diagram of a network with nodes 1, 3, 4, and 2.} \]

\[ \text{Graph with functions: } s(1,t), x(1,t), y(3,1,t), s(2,t), x(2,t), y(2,3,t). \]
Section 5. Implementation and Running Time for Piecewise Linear Net Supply Functions.

For the scheduling application that we have in mind, the net supply functions \( s(n, t) \) are piecewise linear functions of time with a finite number of pieces. In this section we present an informal discussion of the implementation and running time of the algorithm for this type of input data. In that case the functions \( S(n, t) \), \( y(m, n, t) \) and \( z(n, t) \) are also piecewise linear and continuous with a finite number of pieces. We assume that the piecewise linear functions of time are represented by lists of the points of discontinuity (the breakpoints) of the functions and their values at those points. These lists are doubly linked so that they can be scanned in either the forward or in the reverse direction, and breakpoints can be inserted or deleted in constant time.

As for the running time of the Flow Procedures, all of the lines can clearly be executed in constant time with the exception of Lines 3, 4, 8 and 9. These lines are implemented by continuing to scan the list representation of \( S(n, t) \) in the forward direction from the current point in time until we reach a point in time at which the condition in Line 3 or Line 8 starts to fail, performing the side calculations prescribed by Lines 4 and Line 9 as we progress. At least one breakpoint of the list representation of \( S(n, t) \) is passed each time we execute either Line 3 or Line 8, by (g) of Claim 2 in the proof of Theorem 2. Therefore the total time required to execute Lines 3, 4, 8 and 9 is linear in the number of breakpoints of \( S(n, t) \), as is the overall running time of the Flow Procedures.

Note that by (1), each breakpoint of \( S(n, t) \) must give rise to either a breakpoint in \( z(n, t) \), or to a breakpoint in one of \( y(\pi(n), n, t) \) and \( y(n, \pi(n), t) \), depending on whether \( n \) is in \( U \) or in \( L \). Therefore the running time of either of the Flow Procedures is linear in the size of the data structures that represent the flow variables that they create.

We claim that that if \( n \in U \) then the number of breakpoints in \( y(\pi(n), n, t) \) is bounded by the number of breakpoints in \( S(n, t) \). This claim and (14) lead to a useful upper bound on the number of breakpoints in \( y(\pi(n), n, t) \). The bound is the number of breakpoints in \( s(n, t) \), plus the sum over all \( k \in \sigma(n) \cap U \) of the numbers of breakpoints in \( y(n, k, t) \), plus the sum over all \( m \in \sigma(n) \cap L \) of the numbers of breakpoints in \( y(m, n, t) \).

The claim is proven as follows. All of the breakpoints in \( y(\pi(n), n, t) \) are clearly included in one of the intervals \([w, u]\) that are defined in Line 8. Each of these breakpoints corresponds to a breakpoint of \( S(n, t) \) with the exception of \( w \), if \( w > 0 \). However (g) implies that in between any two such intervals \([w, u]\) there is a point in time at which \( S(n, t) \) changes from increasing to non-increasing. This is necessarily a breakpoint of \( S(n, t) \) that does not correspond to a breakpoint in \( y(\pi(n), n, t) \), so the claim holds. The corresponding statement for \( n \in L \) is similarly proven.

The procedure Merge computes the sum of piecewise constant functions of time, given in list format. The algorithm is a direct application of well known algorithms for merging a finite number of sorted lists into a sorted master list. The running time is \( O(M \log K) \) where \( M \) is the number of
elements in the master list and $K$ is the number of lists. In the Network Flow Algorithm the last three lines of Step 1 are easily implemented in time that is linear in the size of the list representation of $\mathcal{S}(N, t)$.

The total running time of the Network Flow Algorithm is therefore linear in the total size of the data structures that are used to represent the flow variables $x(n, t)$ and $y(m, n, t)$, times $\log(K)$ where $K = \max \{|\sigma(n)| : n \in N\}$. This looks like a "best possible" running time result, since a sort of the breakpoints in $\{y(m,n,t) : n \in \sigma(n) \cap L\} \cup \{y(n,k,t) : k \in \sigma(n) \cap U\}$ appears to be required to determine $\mathcal{S}(n, t)$. However the running time can be significantly larger than the size of the data structures that are used to represent the input data because a single breakpoint in a function $s(n, t)$ could give rise to a breakpoint in $\mathcal{S}(k, t)$ for each node $k$ that is on the path connecting node $n$ to the root. It might be possible to find optimal flows that can be represented in space that is linear in the size of the input data. In the scheduling application that motivated this work we are only interested in the optimal cost and in the optimal cut — the flows do not matter. Even if there are no optimal flows that require less data than the ones that this algorithm computes, it still might be possible to find a faster algorithm that computes optimal cuts.

**Section 6. Conclusions.**

The algorithm could easily be extended to a wide variety of other functional forms. For example, we could have piecewise quadratic functions $x(n, t)$, $s(n, t)$ and $y(m, n, t)$. The functions $x(n, t)$ represent the cuts, and they would still be piecewise constant.

For the manufacturing application described in the appendix, it would be valuable to have a fast version of this algorithm that solves a problem that is more general than (P) in two important ways. The first is to allow $\mathcal{G}$ to be any network that does not have any directed circuits. The second is to allow for time lags in the vertical arcs, i.e., to allow the flow $y(m, n, t)$ to go from time $t$ on bar $m$ to time $t + p(m, n)$ on bar $n$. In the manufacturing application the network flow algorithms are used as subroutines in an algorithm that computes approximate solutions to a more complex problem. Therefore running time of the algorithm is critical, but an algorithm that computes near-optimal solutions would be acceptable.

At the cost of added complexity, it appears that this algorithm can fairly easily be adapted for the case of discontinuous regular net supply functions $s(n,t)$ that have a finite number of discontinuities.
References


Appendix: The Job Shop Scheduling Problem

In the appendix we describe the job shop scheduling problem that motivated this research and the role played by the continuous-time maximum-flow problem in its solution. We are given a manufacturing system that consists of a number of different types of machines. We are also given a set of manufacturing operations that must be performed. Each operation \( k \) requires a certain amount of time \( p_k > 0 \) on a machine of type \( m_k \). The operations are modelled as the nodes \( k \in N \) of a network. If the output of an operation \( k \) is required as an input to an operation \( \ell \) then there is a directed arc from \( k \) to \( \ell \) in the network, indicating that operation \( k \) must be completed before operation \( \ell \) begins (a precedence constraint). \( \mathcal{F} \) is the set of operations that correspond to the completion of a product that will be shipped to a customer. Operations \( k \in \mathcal{F} \) have a due date \( d_k \) and a weight \( w_k \). Let \( C_k \) be the completion time of operation \( k \). The weighted tardiness job shop scheduling problem is to schedule the operations on the machines in such a way that no machine performs more than one operation at a time, the precedence constraints are all met, and the weighted tardiness \( \sum_{k \in \mathcal{F}} w_k \max(0, C_k - d_k) \) is minimized.

After listing our notation we will formulate the weighted tardiness job shop scheduling problem as a continuous-time integer linear programming problem.

Notation

\( N(m) \): the number of machines of type \( m \).
\( N \): the set of nodes or operations
\( p_k \): the processing time of operation \( k \).
\( m_k \): the type of machine required for operation \( k \).
\( \mathcal{E} \): the set of arcs or precedence constraints.
\( \mathcal{F} \): the set of operations that correspond to the completion of a job.
\( d_k, k \in \mathcal{F} \): the due date for operation \( k \).
\( w_k, k \in \mathcal{F} \): the weight for operation \( k \).
\( z(k,t) \): A function that is equal to one if operation \( k \) starts before time \( t \), and equal to zero otherwise.

\[
\text{(JSP) min:} \sum_{k \in \mathcal{F}} \int_{d_k - p_k}^T w_k (1 - z(k,t)) \, dt
\]

\[
\text{st:} \quad z(t, t) - z(k,t-p_k) \leq 0 \quad \forall \quad (k,t) \in \mathcal{E}, \quad \forall \quad t \in [0, T]. \quad (22)
\]

\[
z(k,t) \quad \text{is a non-decreasing, left-continuous function of } t \in [0, T] \quad \forall \quad k \in N. \quad (5)
\]

\[
z(k,t) \in \{0,1\} \quad \forall \quad k \in N, \quad \forall \quad t \in [0, T]. \quad (23)
\]

\[
\sum_{\{k: m_k = m\}} [z(k,t) - z(k, t-p_k)] \leq N(m) \quad \forall \quad m, \quad \forall \quad t \in [0, T]. \quad (24)
\]
We are aware of no algorithms for solving problems of this sort. However it is not our intention to solve (JSP) to optimality. In practice, pre-computed solutions to the job shop scheduling problem are difficult or impossible to implement because unforeseen disruptions make the schedule infeasible almost as soon as it is generated. What is needed is an intelligent tool that can provide real-time support for sequencing decisions as they arise on the shop floor. We developed such a tool for a discrete-time version of a similar job shop scheduling problem and tested it. We found it to be very effective in supporting sequencing decisions (see Roundy et al (1991)), but the approach was computational feasible only for very small manufacturing systems. With the continuous time formulation in (JSP) manufacturing systems of realistic size have become computationally feasible. The continuous time formulation also has the advantage that time is modelled as being continuous rather than discrete, so processing times can be modelled very accurately. Computational tests of this approach to job shop scheduling are in progress.

In order to implement our approach to job shop scheduling, we need a solution to (JSP) that is approximately feasible and is approximately optimal, and we need an estimate of the dual prices for equation (24). This is achieved by applying lagrangian relaxation to (24), and using a dual price function $\lambda(m,t)$ that is piece-wise constant in $t$. We define $w(k,t)$, $k \in N$, $t \geq 0$, to be equal to $w_k$ if $k \in E$ and $t \geq d_k - p_k$, and equal to zero otherwise. After dualizing (24), the resulting lagrangian optimization problem is

$$(LJSP) \quad \text{min:} \quad \sum_{k \in E} \int_0^T \left[ -w(k,t) + \lambda(m_k,t) - \lambda(m_k,t+p_k) \right] z(k,t) \, dt$$

subject to (22), (5) and (23).

(LJSP) reduces to (D) as follows. (LJSP) decomposes into subproblems, one for each connected component of $(N,E)$. The orientation of the arcs in $E$ is reversed in $A$. Assuming that the $(N,E)$ is a forest of directed trees, the lag in the time variable in (22) can be eliminated simply by changing our definition of $z(k,t)$. This reduces (22) to (4). Another change of variable and the Lemma above imply that (23) can be replaced by (6) and (7). The function $s(n,t)$ is defined by comparing the objective functions of (LJSP) and (D). It is an integral of the term in brackets in (93) above, and is a continuous, piece-wise linear function of time.

The algorithm that we have implemented for approximately solving (JSP) applies 90 iterations of a standard subgradient optimization technique to obtain the dual price functions $\lambda(m,t)$. We are doing this with data taken from a machine shop that includes about 1600 jobs, a total of about 3300 operations, and about sixty different types of machines. The computation takes four minutes on a SUN Sparc 1. We added a routine that periodically reduces the number of discontinuities in $\lambda(m,t)$ on an as-needed basis, by combining adjacent time intervals.
Some discussion of the topology of the network $(\mathcal{N}, \mathcal{E})$ is appropriate. In the classical job shop $(\mathcal{N}, \mathcal{E})$ would consist of a collection of serial networks, with each node having at most one predecessor and at most one successor. $\mathcal{F}$ would consist of the nodes that have no successor. The data set that we are using in our computational tests has these properties. If assembly takes place part way through the manufacturing process then a node can have multiple predecessors, but would still have at most one successor. In that case $(\mathcal{N}, \mathcal{E})$ is a forest of trees, each having the arcs oriented toward the root of the tree. Most manufacturing systems probably fit within this modelling framework.

A node will have multiple successors in either of two events. The first is the case of disassembly, a manufacturing process in which one item is converted into two or more different items which are then processed separately. This is unusual in the industrial world, but it does occur.

The second event that gives rise to multiple successors of a node is much more common. It has to do with lot sizing. When production occurs in batches rather than one part at a time, we model the processing of a production batch at a machine as a single node in $(\mathcal{N}, \mathcal{E})$. The type of lot sizing technique most commonly used in industry is lot-for-lot, meaning that each production batch corresponds to the shipment of a finished product. Under this approach the bill of material network for the finished product becomes one of the connected components of $(\mathcal{N}, \mathcal{E})$.

The most common alternative to lot-for-lot is lot splitting. Under lot splitting a production batch can subdivided into two or more sub-batches, each of which proceeds separately. That means that the node in $(\mathcal{N}, \mathcal{E})$ that corresponds to the last operation performed on the combined batch will have a successor node for each of the different sub-batches.

If lot splitting takes place in a classical job shop, meaning that the bill of material networks of the finished products are serial, the resulting network $(\mathcal{N}, \mathcal{E})$ is a forest of trees, each having the arcs oriented away from the root of the tree.

If lot splitting takes place in a manufacturing system that also has assembly operations, the resulting network $(\mathcal{N}, \mathcal{E})$ is a union of connected networks that may have cycles, but do not have directed circuits. This is the only case that we have discussed for which the algorithm of this paper is not applicable. Because of the circuits we can not eliminate the lag in the time variable in (22) by changing our notation, so (LJSP) will have a constraint of the form of (22) in place of (4). In the network flow problem analogous to (P), this will lead to time lags in the vertical arcs of $\mathcal{G}$ that connect different bars in (P). Time lags of this type were addressed by Philpott (1990).