RESOLVE:

An algorithm for the Minimum Dummy Task Problem

Sharon G. Boswell,
Department of Mathematics, The University of Newcastle,
NSW, AUSTRALIA 2308

and

Roger B. Eggleton,
4520 Mathematics Department, Illinois State University,
Normal, Illinois, U.S.A. 61790-4520

ABSTRACT

Scheduling graphs are used by algorithms such as PERT/CPM in order to determine an optimal schedule for a given project. It is well-known that *dummy* tasks (requiring zero processing time) must sometimes be incorporated into a scheduling graph.

The main tool in the paper is a new algorithm RESOLVE, which creates a scheduling graph, typically with fewer dummy tasks than are produced by the Richards' algorithm (1967). A theoretical framework for scheduling graphs is systematically developed through several theorems, culminating in a demonstration of the validity of RESOLVE. A worked example illustrating the application of RESOLVE concludes the paper.

JCMCC **30** (1999), pp. 45-65

1. Introduction

Algorithms for project scheduling include Project Evaluation and Review Technique and the Critical Path Method (PERT/CPM). These are widely discussed in the literature; two readily accessible treatments are [4, 5]. They typically require as input a directed graph, which we shall call a scheduling graph, in which the edges represent the individual tasks, the vertices mark the beginnings and ends of tasks, and consecutive edges correspond to tasks which can only be performed sequentially. The terminology used in the literature varies widely, for example the graphs are called as a *event-node networks*, *PERT/CPM network*, and *arrow diagrams* [9] etc. We begin by clarifying our terminology and the problem itself.

A *task* is an activity which is performed in a single interval of time, of prescribed length, called its *processing time.* The processing time of a task a is denoted by $\tau(a)$, so $\tau(a) \geq 0$. A *project* comprises a finite set of tasks, the *task set* T, together with a specification that certain pairs of tasks can only be performed sequentially. (It is implicit that other pairs of tasks in T could be performed in overlapping intervals of time.) The total completion time of a project of $|T| = n$ tasks is

$$\tau(T) := \max_{r \leq n} \left\{ \sum_{i=1}^{r} \tau(a_i) : (a_1, a_2), (a_2, a_3), \ldots, (a_{r-1}, a_r) \in \text{sched}(T) \right\}$$

Note that the definition of $\tau(T)$ does not refer to any scheduling graph. However, if there is a scheduling graph G which properly represents T, then $\tau(T)$ is the maximum of the expressions

$$\sum_{a \in P} \tau(a)$$

taken over the directed paths P in G.

If task a must be completed before task b can be started then a is a *predecessor* of b; equivalently, b is a *successor* of a. If there is no task which is both a successor of a and a predecessor of b then a is an *immediate predecessor* of b and b is an *immediate successor* of a. Initially, the available sequencing information about the task set T is a binary relation which we call the *prescheduling relation*, presched(T). Obviously, it must not contain a directed cycle, this can be tested using Warshall's matrix algorithm ([**10**],[**12**]). The prescheduling relation comprises predecessor–successor pairs, but these pairs are not necessarily immediate. The subset of presched(T) comprising precisely those predecessor-successor pairs which are immediate is called the *scheduling relation*, sched(T). The scheduling relation is the transitive closure of the prescheduling relation.

If one plans to use an algorithm (such as PERT/CPM) which uses the scheduling graph as input, that graph must first be constructed: in practice, however, difficulties may arise when one attempts to construct a directed graph to represent a given scheduling relation. In many cases, as we shall show in Section 3, it is necessary to incorporate into the task set one or more *dummy tasks,* which have zero processing time, simply to make it possible for a scheduling graph to be drawn. The problem of finding the minimum set of dummy tasks has been shown to be NP-complete [**8**], formulated as finding a "minimal PERT network equivalent to a given precedence relation represented by an acyclic digraph". In this paper, we present an algorithm which inserts a minimal sufficient set of dummy tasks to achieve representation as a scheduling graph. However, it is not the first algorithm designed for this purpose: that credit apparently belongs to Richards [**9**]. On all test problems used to date, Richard's algorithm inserts at least as many dummy tasks into the graph as does our algorithm RESOLVE.

2. The scheduling graph

Each immediate predecessor-successor pair of tasks must correspond to a pair of consecutive edges in the scheduling graph. But this condition is not sufficient to ensure that the graph will yield an optimal schedule. We must also include its converse, as the following example shows. Take the task set $T_1 := \{a, b, c\}$ suppose that b and c can only be started after a has been processed. Then $\text{sched}(T_1) := \{(a, b), (a, c)\}$. Consider the two directed graphs shown in Figure 1.

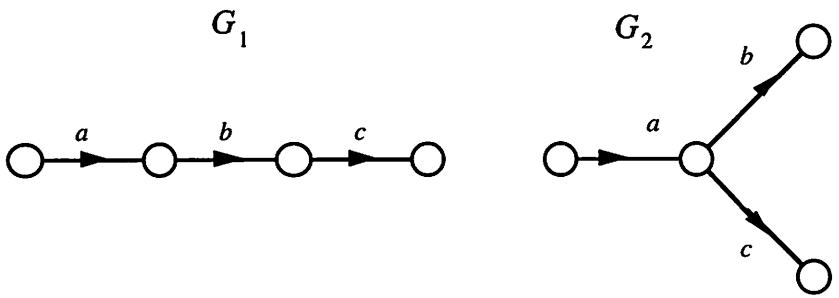

Figure 1. *Two scheduling graphs which represent* $\text{sched}(T_1)$.

The graph G_1 introduces the additional constraint (b, c) which obviously has the potential to increase the optimal completion time. The ordered pair (a, c) is represented in G_1 by a directed path in which the edges a and c are not consecutive, whereas this phenomenon does not occur in G_2. In general, for a scheduling graph G to *properly represent* $\text{sched}(T)$ we require that the ordered pairs of tasks in $\text{sched}(T)$ correspond to the pairs of consecutive edges in G, and vice versa. In other words, G has just the right collection of pairs of consecutive edges to represent the constraints in $\text{sched}(T)$ and no others.

We shall show in Section 3 that it is not always possible for a scheduling relation to be properly represented by a scheduling graph. When a scheduling graph which properly represents sched(T) does exist, the corresponding schedule is automatically optimal, because it respects all and only those sequencing restrictions generated by presched(T).

In fact, computing an optimal schedule for T is possible from sched(T) without reference to a scheduling graph. However, when we do have a scheduling graph G which properly represents T we have several advantages. The scheduling graph is a conceptual aid to dealing with the under-lying structure of sched(T), and helps us efficiently compute an optimal schedule. (Indeed, the standard PERT/CPM algorithms are written in terms of G, so effectively assume that one has a scheduling graph which properly represents T.) Moreover, even after an optimal schedule is determined, G remains useful as a visual aid for those working on the project represented by T.

3. Dummy tasks

We now show that a given scheduling relation may fail to have any corresponding scheduling graph which properly represents it. Consider the problem of scheduling four tasks a, b, c, d such that a is a predecessor of c and d, and b is a predecessor of d. Here the task set is $T_2 := \{a, b, c, d\}$ and the scheduling relation is

$$\text{sched}(T_2) := \{(a, c), (a, d), (b, d)\}.$$

Suppose there is a scheduling graph G which properly represents sched(T_2). Since c and d are both immediate successors of a, there must be a vertex in G which a enters and both c and d leave. But b is an immediate predecessor of d, so b must enter the same vertex in G . Then G represents sched(T_2). However,

the tasks b and c are represented by consecutive edges in G, yet the ordered pair (b, c) is not in sched(T_2). Thus G does not properly represent sched(T_2). This contradiction shows that sched(T_2) is not properly represented by any scheduling graph.

The traditional way of coping with this problem is to introduce *dummy tasks*, that is, tasks with zero processing time. Thus, by introducing a dummy task e it is possible to produce a scheduling graph (Figure 2) which leads to an optimal schedule for T_2. With $T_2' := T_2 \cup \{e\}$ and $\tau(e) = 0$, let us take

$$\text{presched}(T_2') := \text{sched}(T3) \cup \{(a, e), (e, d)\}.$$

Then

$$\text{sched}(T_2') := \{(a, c), (a, e), (b, d), (e, d)\}.$$

Note that the scheduling graph in Figure 2 properly represents sched(T_2').

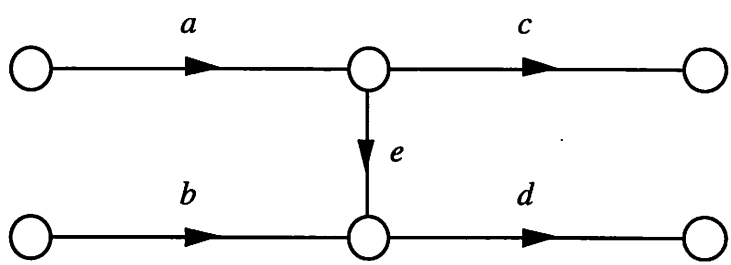

Figure 2. *An appropriate scheduling graph for* sched(T_2').

If T' is any task set which contains T, define the *restriction of* <sched(T')> *to* T, denoted by

$$\langle \text{sched}(T') \rangle |_T$$

to be the set of all ordered pairs $(a, b) \in$ <sched(T')> such that $a, b \in T$. When we have <sched(T')>$|_T$ = <sched(T)>, the tasks in $T' \setminus T$ do not lead to any

predecessor-successor pairs involving tasks in T which are not already implied by sched(T). Then we have

THEOREM 1. *Let T' be any task set which is derived from the task set T by adjoining dummy tasks in such a way that* $\langle \text{sched}(T') \rangle|_T = \langle \text{sched}(T) \rangle$. *Then* $\tau(T') = \tau(T)$.

PROOF. See [2]

COROLLARY. *Under the conditions of Theorem 1, any optimal schedule for T' yields an optimal schedule for T simply by suppressing the dummy tasks.*

PROOF. See [2]

Theorem 1 and its Corollary give us grounds for hope. Even though a given sched(T) may not be properly representable by any scheduling graph, if we can insert dummy tasks in the right way the resulting relation sched(T') will be properly representable and will yield the appropriate information about sched(T). In what follows, we shall show that it is in fact always possible to insert dummy edges to achieve this, and we give an algorithm for doing so.

When are dummy tasks required? We can answer this question with the following concepts. The *immediate predecessor set* and *immediate successor set* of a task $a \in T$ are the sets

$$\text{pred}(a) := \{t \in T: (t, a) \in \text{sched}(T)\},$$

$$\text{succ}(a) := \{t \in T: (a, t) \in \text{sched}(T)\}.$$

Two distinct tasks a, b in T are *predecessor–equivalent* if pred(a) = pred(b) $\neq \varnothing$. This defines an equivalence relation on T; let pred(T) be the resulting partition of T into equivalence classes. Similarly, two distinct tasks a, b are *successor–equivalent* if succ(a) = succ(b) $\neq \varnothing$. Let succ(T) denote the corresponding partition of T. The tasks with empty predecessor set form singletons of pred(T) and those with empty successor set form singletons of succ(T).

Now consider the scheduling graph G. Because G properly represents sched(T), the tasks represented by the incoming edges at a vertex v of G are all successor–equivalent, and those represented by the outgoing edges at v are all predecessor-equivalent. Each interior vertex of G can therefore be regarded as the nexus of a predecessor set P_i and a successor set S_i ; each initial vertex of G can be associated with a successor set; and each terminal vertex of G can be associated with a predecessor set. Then every vertex of G is an ordered pair of disjoint sets of tasks (P_i, S_i), where P_i is empty for an initial vertex and S_i is empty for a terminal vertex. Note that every task belongs to exactly one set P_j and exactly one set S_k, since pred(T) and succ(T) are partitions of T.

Recall that sched(T_2) := {(a, c), (a, d), (b, d)} is not properly represented by any scheduling graph, but extending T_2 to T_2' by adding a dummy task e and suitably extending the prescheduling relation yields a scheduling relation sched(T_2') which can be properly represented (Figure 2). What forced the introduction of e? Both c and d belong to succ(a); this requires c and d to exit from the same vertex. But task d is the only successor of task b; this requires c and d to exit from different vertices. This incompatibility is caused by the *overlap* of succ(a) and succ(b), that is, these two sets have *at least one, but not all, members in common.* This can also be expressed in a dual form, by

discussing the predecessor sets pred(c) and pred(d), with consideration of whether a and b should enter the same or different vertices.

THEOREM 2. *The following two properties are equivalent:*

1. *There exist tasks a and b such that* $\text{succ}(a) \neq \text{succ}(b)$ *and* $\text{succ}(a) \cap \text{succ}(b) \neq \emptyset$.
2. *There exist tasks c and d such that* $\text{pred}(c) \neq \text{pred}(d)$ *and* $\text{pred}(c) \cap \text{pred}(d) \neq \emptyset$.

PROOF. Assume $\text{succ}(a) \neq \text{succ}(b)$ and $\text{succ}(a) \cap \text{succ}(b) \neq \emptyset$ for some tasks a and b. Let $c \in \text{succ}(a) \cap \text{succ}(b)$. At least one of succ($a$) and succ($b$) contains a task which is not in the other. Suppose $d \in \text{succ}(a) \setminus \text{succ}(b)$. Then a and b are both in pred(c), and a is in pred(d) but b is not. Similarly if $d \in \text{succ}(b) \setminus \text{succ}(a)$. Thus $\text{pred}(c) \neq \text{pred}(d)$ and $\text{pred}(c) \cap \text{pred}(d) \neq \emptyset$. Hence Property 1 implies Property 2. Likewise Property 2 implies Property 1. □

A scheduling relation sched(T) with Property 1 in Theorem 2 cannot be properly represented by any scheduling graph. For suppose $\text{succ}(a) \neq \text{succ}(b)$ and $\text{succ}(a) \cap \text{succ}(b) \neq \emptyset$. In any scheduling graph which could properly represent sched(T), the vertex entered by a must be entered by b, since $\text{succ}(a) \cap \text{succ}(b) \neq \emptyset$; but it cannot be entered by b, since $\text{succ}(a) \neq \text{succ}(b)$, so we have an impossibility.

If Property 1 does not hold for a given scheduling relation sched(T), then the successor sets form a *partition* of T, that is, any two distinct successor sets are disjoint, and together their union is T. (This is called an *improper partition* in [4] because it allows empty sets among the parts.) If the successor sets form a partition of T then sched(T) is properly represented by some scheduling graph. In fact, adapting theorems of Harary and Norman [5] and Geller and Harary [4]

to our present context we have the following characterisation of properly representable scheduling relations.

THEOREM 3. (Geller, Harary and Norman). *A scheduling relation* sched(T) *is properly represented by some scheduling graph if and only if either of the following holds:*

1. *The successor sets form a partition of T;*
2. *The predecessor sets form a partition of T.*

The negations of the properties in Theorem 3 are the properties in Theorem 2, and in that form they are well suited to practical testing for the nonexistence of a scheduling graph which properly represents a given scheduling relation. Theorems 2 and 3 form the basis for an algorithm to insert dummy tasks in a way which removes the obstacles to proper representation.

4. An algorithm for inserting dummy tasks

Given a scheduling relation sched(T) which cannot be properly represented by a scheduling graph, we can resolve this difficulty with the algorithm RESOLVE, presented below. It extends T to a new task set T' by the addition of dummy tasks and produces a scheduling relation sched(T') which is consistent with sched(T) and does have a proper representation.

Let adj(T) be the adjacency matrix of the scheduling relation sched(T). An *L-shape* in a (0,1)-matrix is a rectangular configuration of three 1's and one 0. The overlap of two immediate successor sets determined by sched(T) corresponds to the presence of an L-shape in adj(T). Thus Theorems 2 and 3 imply that sched(T) has no proper representation as a scheduling graph precisely when

adj(T) contains at least one L-shape. RESOLVE is designed to locate and remove L-shapes from adj(T).

RESOLVE is made up of three procedures, CONDENSE, IMPROVE, and REDUCE. CONDENSE removes redundant rows and columns, to avoid duplication of dummy tasks and reduce the computational effort required. IMPROVE does the real work. At each iteration it removes all L-shapes from one row of the matrix, and creates dummy tasks corresponding to a minimal set of later rows which participated in those L-shapes.

IMPROVE may introduce some dummy tasks which are unnecessary but this can only be determined by considering the whole scheduling graph. REDUCE completes the job by removing these redundant dummy tasks. There are two types of dummy tasks which are redundant, those that can be deleted and those that can be contracted. The operations of edge deletion and edge contraction are well-known in graph theory and apply equally to directed graphs (see for example [11]) The standard notation is that $G-e$ denotes the graph formed by deleting the edge e from the graph G, and $G.e$ denotes the graph formed by contracting e in G. We adapt this for our current purpose by using the adjacency matrices in place of the graphs. REDUCE uses the transitive closure of the scheduling relation to determine which dummy tasks, if any, can be deleted or contracted.

The all-zero rows in adj(T) correspond to *finish* tasks (tasks with no successors) and the all-zero columns correspond to *start* tasks (tasks with no predecessors). RESOLVE begins with the *abbreviated* adjacency matrix **A**, constructed as follows. From adj(T) omit each all-zero row and label each remaining row with

the set $\{t\}$, where t is the task represented by the row. Then omit each all-zero column and correspondingly label each remaining column.

Let **A** be the matrix being processed at any stage. We denote the ith row of **A** by $\mathbf{A}(i)$ and its label set by $L(i)$. We denote the ith column of **A** by $\mathbf{A}[i]$ and its label set by $L[i]$.

CONDENSE

Let $\mathbf{A}(r)$ be the first row of **A** with the property that $\mathbf{A}(r) = \mathbf{A}(s)$ for some $s > r$. Replace the label set $L(r)$ of $\mathbf{A}(r)$ by $L(r) \cup L(s)$. Delete the row $\mathbf{A}(s)$ and appropriately adjust the numbers of lower rows and their label sets. Iterate until all rows are distinct.

Let $\mathbf{A}[r]$ be the first column with the property that $\mathbf{A}[r] = \mathbf{A}[s]$ for some $s > r$. Replace the label set $L[r]$ of $\mathbf{A}[r]$ by $L[r] \cup L[s]$. Delete the column $\mathbf{A}[s]$ and appropriately adjust the numbers of columns to the right and their label sets. Iterate until all columns are distinct. {End CONDENSE}

We remark that after the first application of CONDENSE, the matrix **A** is similar to, but not identical with, the *coalesced* matrix used by Richards [9].

For completeness, we define two Boolean operations which we shall need for IMPROVE. If **X** and **Y** are any two (0, 1)-matrices of the same shape, then

$$\mathbf{X} \,\&\, \mathbf{Y} \qquad \text{and} \qquad \mathbf{X} \vee \mathbf{Y}$$

are (0, 1)-matrices of the same shape defined by entrywise application of the following rules. For all $x, y \in \{0, 1\}$:

$$x \& y = \begin{cases} 1 & \text{if } x = 1, y = 1 \\ 0 & \text{otherwise,} \end{cases} \qquad \text{and} \qquad x \vee y = \begin{cases} 0 & \text{if } x = 0, y = 0 \\ 1 & \text{otherwise.} \end{cases}$$

We call **X** & **Y** the *intersection* of **X** and **Y**, and $\mathbf{X} \vee \mathbf{Y}$ the *join* of **X** and **Y**. If **X**, **Y** and **Z** are any (0, 1)-matrices of the same shape, we shall say that **Y** *is equivalent to* **Z** *for* **X**-*intersection* if **X** & **Y** = **X** & **Z**.

IMPROVE

Let **A**(r) be the first row of **A** with the property that **A**(r) & **A**(s) $\neq$ **0** for some $s > r$. {Row r is the active row.} If **A**(r) & **A**(s) = **A**(r) for some $s > r$, interchange rows **A**(r) and **A**(s); iterate until this condition no longer holds.

Choose a minimal set of rows **A**(s_1), **A**(s_2), ..., **A**(s_k) with join **A**(s_1) $\vee$ **A**(s_2) $\vee$ ... $\vee$ **A**(s_k) which is equivalent to the join of all rows **A**(s), $s > r$, for **A**(r)-intersection.

Add a border of k new rows below **A**, denoted **B**(1), **B**(2), ..., **B**(k), where **B**(i) := **A**(r) & **A**(s_i). Let $a_1, a_2, ..., a_k$ be k new symbols not already used as labels. Let $\{a_i\}$ be the label set of **B**(i). Replace **A**(r) by **A**(r) – **A**(r) & [**A**(s_1) $\vee$ **A**(s_2) $\vee$ ... $\vee$ **A**(s_k)]. The row **A**(r) retains its label set $L(r)$. Add a new column **C**, to the right of **A** and **B**, which has 1 in row r and all other entries zero. Let $\{a_1, a_2, ..., a_k\}$ be the label set of **C**. {End IMPROVE}

The final procedure, REDUCE, uses the transitive closure of the scheduling relation. We define TRANS(**A**) to be the transitive closure of the scheduling relation represented by matrix **A**.

REDUCE

For each dummy task α do

if TRANS(**A**)=TRANS(**A**–α) then A:=A–α.

For each dummy task α do

if TRANS(**A**)=TRANS(**A**·α) then A:=A·α.

{End REDUCE}

The overall algorithm, RESOLVE, combines these procedures:

RESOLVE

Begin with the abbreviated adjacency matrix **A**.

Put Labelset := T.

Repeat

CONDENSE(**A**)

IMPROVE(**A**)

Augment Labelset with the labels $a_1, a_2, ..., a_k$

until **A** contains no L-shapes.

REDUCE(**A**)

Put T' := Labelset. {End RESOLVE}

Now construct adj(T') from the final abbreviated matrix **A** as follows. Note that the finish tasks in T' are those not used to label any row of **A**, and the start tasks are those not used to label any column of **A**. First replicate all rows of **A** which have more than one label. Let $l(i)$ be the number of labels in the label set $L(i)$. If $l(i) \geq 1$, adjoin $l(i) - 1$ new rows to the matrix, all identical with the row **A**(i), and assign each of the labels from $L(i)$ to one of the $l(i)$ identical rows. Create an all-zero row for each of the finish tasks and label accordingly. Replicate all columns which have more than one label in the corresponding way, adding all-zero columns for each of the start tasks. Now reorder the columns so they are in the same order as the rows. The resulting matrix is adj(T').

5. Proof of RESOLVE

We now prove that the algorithm RESOLVE really does what it is supposed to do. First we must show that it really does terminate, so is a genuine algorithm.

This is not trivial, for although each iteration of IMPROVE does remove all L-shapes involving the active row **A**(r), nevertheless the newly inserted rows **B**(1), **B**(2), ..., **B**(k) can introduce new L-shapes into the matrix, and the total number of L-shapes present can actually increase as a result of one iteration. For reasons of brevity the reader is referred to [**2**] for the proof.

THEOREM 4. *The algorithm* RESOLVE *does terminate.*

PROOF. See [**2**, Section 7]

In Section 4 we showed how to construct the adjacency matrix adj(T') corresponding to the matrix output by RESOLVE. From it we can directly construct a scheduling graph which properly represents sched(T'), as the following theorem shows.

THEOREM 5. *Let* sched(T') *be the scheduling relation resulting from the application of the algorithm* RESOLVE *to* sched(T). *Then* sched(T') *has a proper representation as a scheduling graph, and* $\langle \text{sched}(T') \rangle|_T = \langle \text{sched}(T) \rangle$.

PROOF. (1) The algorithm RESOLVE terminates when there are no L-shapes in the output matrix **A**. This is equivalent to both the properties in Theorem 2 failing. Therefore it is equivalent to each of the properties in Theorem 3 holding. Then Theorem 3 ensures that sched(T') is properly represented by a scheduling graph, where T' := RESOLVE(T).

(2) From the definition of adj(T) and the abbreviated adjacency matrix **A**, we see that sched(T) comprises exactly those ordered pairs (a, b) such that **A** contains a 1 at the intersection of the row labelled $\{a\}$ and the column labelled $\{b\}$. Let $\mathbf{A}^{(n)}$ be the current matrix **A** and $\text{Labelset}^{(n)}$ be the current Labelset at the start of

the nth iteration of the loop within RESOLVE. Then $\mathbf{A}^{(n)}$ can be regarded as an abbreviated adjacency matrix for the task set $T_n := \text{Labelset}^{(n)}$ with

$$\text{sched}(T_n) := \{(a, b): a \in L(i), b \in \text{L}[j], \mathbf{A}^{(n)}_{ij} = 1\}.$$

We claim that

$$\langle \text{sched}(T_n) \rangle |_T = \langle \text{sched}(T) \rangle \qquad (3)$$

holds for all iterations. It certainly holds initially, for $T_1 := T$. Suppose (3) holds when commencing the nth iteration of the loop within RESOLVE. CONDENSE($\mathbf{A}^{(n)}$) does not alter the ordered pairs represented by $\mathbf{A}$; it only combines the representation of certain pairs using fewer 1's. But IMPROVE typically alters the set of ordered pairs currently represented by $\mathbf{A}$. If $\mathbf{A}^{(n)}(r)$ is the active row, then IMPROVE($\mathbf{A}^{(n)}$) replaces certain entries 1 in $\mathbf{A}^{(n)}(r)$ by 0, and adjoins a 1 to the end of $\mathbf{A}^{(n)}(r)$. If the substitution $1 \leftarrow 0$ occurs at $\mathbf{A}^{(n)}_{rj}$, this corresponds to deletion of (a, b) from sched(T_n) for each $a \in L(r)$, $b \in L[j]$. The 1 adjoined to the end of $\mathbf{A}^{(n)}(r)$ is in the column labelled $\{a_1, a_2, ..., a_k\}$, so it corresponds to insertion of (a, a_i) into sched(T_{n+1}) for each $a \in L(r)$ and $1 \le i \le k$. IMPROVE($\mathbf{A}^{(n)}$) also inserts k new rows $\mathbf{B}(i)$ below $\mathbf{A}^{(n)}$, so that $\mathbf{B}(i) := \mathbf{A}^{(n)}(r)$ & $\mathbf{A}^{(n)}(s_i)$, with label set $\{a_i\}$.

So the 1's in $\mathbf{B}(i)$ occur in those columns j where both $\mathbf{A}^{(n)}(r)$ and $\mathbf{A}^{(n)}(s_i)$ have an entry 1. For each such j, the corresponding entry 1 in $\mathbf{B}(i)$ represents insertion of (a_i, b) into sched(T_{n+1}) for each $b \in L[j]$.

Since $T_{n+1} \setminus T_n = \{a_1, a_2, ..., a_k\}$, the net result of the nth pass through the loop in RESOLVE is that sched(T_{n+1}) is obtained from sched(T_n) by deleting all those pairs (a, b) with $a \in L(r)$, $b \in L[j]$, $\mathbf{A}^{(n)}_{rj} = 1$ and inserting all pairs

$$(a, a_i), (a_i, b) \text{ with } a_i \in T_{n+1} \setminus T_n.$$

But (a, a_i), $(a_i, b) \in$ sched(T_{n+1}) implies $(a, b) \in$ <sched(T_{n+1})>, so the pairs in sched(T_n) which are not retained in sched(T_{n+1}) are contained in the closure of sched(T_{n+1}). Therefore

$$<\text{sched}(T_{n+1})>|_{T_n} = <\text{sched}(T_n)>$$

so

$$<\text{sched}(T_{n+1})>|_T = <\text{sched}(T_n)>|_T = <\text{sched}(T)>,$$

where the last step follows from hypothesis. By finite induction, (3) holds for all iterations, so at the last iteration

$$<\text{sched}(T')>|_T = <\text{sched}(T)>. \quad \square$$

In effect the algorithm RESOLVE shows us that the scheduling problem associated with a task set T always has a schedule which only requires time $\tau(T)$ to complete. In fact Theorem 1 and its Corollary combine with Theorem 5 to show the following.

COROLLARY. *Every scheduling relation* sched(T) *has an optimal schedule which processes all tasks in a total elapsed time equal to* $\tau(T)$.

6. Illustration of RESOLVE

The algorithm RESOLVE was programmed in Pascal, as was Richards' algorithm, as specified in [9]. (A copy of the code is available from the first author.) We have compared the two algorithms on a number of test problems and in each case tested we found that Richards' algorithm produces at least as many dummy tasks as RESOLVE, and frequently produces more. However, we have no proof that this is always the case.

To illustrate the effectiveness of RESOLVE, we have chosen the following scheduling problem of practical size. The task set is $T_5 := \{a, b, c, d, e, r, s, t, u, v\}$ and the scheduling relation, sched(T_5), is shown in abbreviated matrix

form in Figure 12. Note that this example has numerous overlaps in its successor sets.

	$\{r\}$	$\{s\}$	$\{t\}$	$\{u\}$	$\{v\}$
$\{a\}$	1	1	1	1	0
$\{b\}$	0	1	1	1	1
$\{c\}$	1	0	1	1	0
$\{d\}$	1	1	0	1	0
$\{e\}$	1	1	0	0	1

Figure 3. *The abbreviated adjacency matrix for* sched(T_5).

Our example shows that RESOLVE can handle equivalent problems from other contexts. In particular, it could be interpreted as a problem concerning a "black box" which has input channels a, b, c, d, e and output channels r, s, t, u, v. Output on channel r is a combination of inputs on channels a, c, d, e; output on channel s is a combination of inputs on channels a, b, d, e; and so on, as specified by the matrix in Figure 3. Determine a small internal network for the black box.

An obvious solution suggested by the "black box" version of the problem is to insert a single directed edge (dummy task) from each input channel to each appropriate output channel. This yields a solution with 17 dummy tasks. Richards algorithm yields precisely this result. However, in 11 iterations, IMPROVE produces a solution with only 15 dummy tasks of which 2 are removed by REDUCE, giving a final solution from RESOLVE with 13 dummy tasks as shown in Figure 4. The dummy tasks introduced at iteration n are

labelled An, Bn etc. The vertices are also labelled with the number of the iteration at which they were effectively introduced, except for the initial vertices of both the start and finish tasks and the final vertices of the finish tasks.

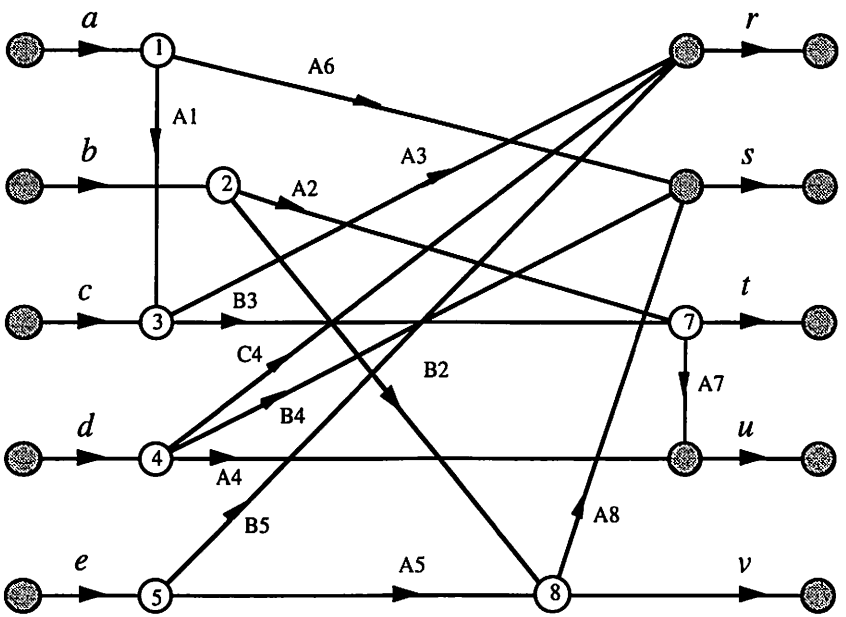

Figure 4. *The scheduling graph produced by* RESOLVE *for* sched(T_5').

7. Closing remarks

For RESOLVE, some important open problems remain. Is the set of dummy tasks produced by RESOLVE actually minimum as well as minimal? If not, is it always better than Richards' algorithm [9]?

Scheduling relations may be represented by a directed graph in an alternative manner – where the tasks are represented by the *vertices* of the graph, and there is a directed edge from vertex i to vertex j precisely when $(i, j) \in$ sched(T). In general, the vertex scheduling graph for sched(T) is the line digraph (see [4]) of

the scheduling graph for sched(T). Condensation of such graphs has also been investigated (see [1] for example).

Vertex scheduling graphs are easy to draw and we feel that there is much to recommend them. Edge scheduling graphs are in common use with PERT/CPM algorithms, but these algorithms can be adapted to operate on vertex scheduling graphs. However vertex scheduling graphs do lack the visual advantage of (edge) scheduling graphs where the lengths of the edges suggest the duration of the task. Moreover, the "black box" version of the example in Section 6 shows another context in which it is natural for the unknowns to be represented by directed edges rather than vertices.

Acknowledgment The authors are pleased to acknowledge that the research leading to this paper was partially supported by University of Newcastle Research Grant 45/290/207V.

References

[1] Armon, D. (1971), *On the condensation of network diagrams*, in Avi-Itzak, B. *et al* (eds.), Developments in Operations Research, vol.2 (Gordon & Breach, New York), pp. 297-309.

[2] Boswell, S.G and Eggleton R.B. (1993) *How to Create a Scheduling Graph*, Research Report, Department of Mathematics, The University of Newcastle, NSW 2308, Australia.

[3] Boswell, S.G and Eggleton R.B. (1996) *How to Create a Scheduling Graph II - reducing the number of dummy edges.* Research Report, Department of Mathematics, The University of Newcastle, NSW 2308, Australia.

[4] Geller, D.P. and Harary, F. (1968), *Arrow diagrams are line digraphs*, SIAM J. Appl. Math. **16**, pp. 1141-1145.

[5] Harary, F. and Norman, R.Z. (1960), *Some properties of line digraphs*, Rend. Circ. Mat. Palermo (2) **9**, pp. 161-168.

[6] Hastings, K.J. (1989), *Introduction to the Mathematics of Operations Research*, (Marcel Dekker, New York).

[7] Hillier, F.S. and Lieberman, G.J. (1980), *Introduction to Operations Research*, (Holden-Day, Oakland CA).

[8] Krishnamoorthy, M. S. and Deo, N. (1979), *Complexity of the Minimum-Dummy-Activities Problem in a PERT Network*, Networks **9**, pp 189-194.

[9] Richards, P.I. (1967), *Precedence constraints and arrow diagrams*, SIAM Review, **9**,
pp 548-553.

[10] Skvarcius, R. and Robinson, W.B. (1986), *Discrete Mathematics with Computer Science Applications*, (Benjamin/Cummings, Menlo Park CA).

[11] Tutte, W.T. (1984), *Graph Theory* , (Addison-Wesley, CA).

[12] Warshall, S. (1962), *A Theorem on Boolean Matrices*, J. Assoc. Comput. Mach. **9**,
pp 11-12.