# An Efficient Method to Maintain Resource Availability Information for Scheduling Applications

Stephen P. Smith
Automation Sciences Laboratory
Northrop Research & Technology Center
1 Research Park
Palos Verdes Peninsula, CA 90274

September 9, 1991

**Abstract**

It is vital for a scheduler to efficiently determine when a resource is unassigned to some previously scheduled operation. Efficient algorithms are needed for calculating, querying, and updating this information. This paper presents an approach for using height-balanced trees for storing resource availablity information. We show that availibility queries are then efficient and that updates can be done in time logarithmic in the size of the availability information. Further, we present an algorithm for calculating the availability of sets of resources which is linear in the number of previously scheduled operations and show that this is optimal.

Index Terms: Scheduling, Resource Allocation, Balanced Binary Trees

Submitted to *IEEE Robotics and Automation Conference*, 1992.

# 1 Introduction

Scheduling is an important function in any manufacturing operation. Good schedules are required to ensure: smooth functioning of the shop, efficient use of resources, and satisfaction of customer requirements.

One of the key components of any scheduler is resource allocation. Resource allocation deals with assigning the appropriate resource instances to perform an operation for some period of time. A common assumption is that resouces instances may perform only one operation at a time and thus it is vital for a scheduler to efficiently determine when a resource is unassigned to some previously scheduled operation.

For example, a machining operation, which takes a known fixed amount of time $t$ to complete, may require both an instance of the general resource class of machinists and an instance of the resource class of milling machines. The resource allocation problem for this operation is to find a time period longer than $t$ which ends before the required completion time for this machining operation and for which both a machinist and a milling machine are available for work and unassigned to any other work.

Clearly a key component of a resource allocation algorithm is the querying and updating of the availability information of resource instances. It is vital that that queries such as "Is resource $R$ available between times $t_1$ and $t_2$?" and "What operation holds resource $R$ at time $t$?" be answered in an efficient manner. Further, as the scheduler proceeds, it is vital that the resource availibility information be updated efficiently.

This paper presents an approach for using height-balanced trees for storing resource availablity information. We show that availibility queries and updates can be done in time logarithmic in the size of the availability information. Further, we show that calculating the availability of sets of resources is linear in the number of previously scheduled operations.

Section 2 provides an overview of past work on this class of problems. Section 3 contains some mathematical preliminaries. Section 4 defines the availablity of a resource instance while Section 5 defines updating algorithms for this availablity information. Section 6 analyzes these algorithms when height-balanced trees are used to represent availablity. Section 7 shows that this leads to an algorithm linear in the number of scheduled operations for computing resource availability over multiple resources instances. An Appendix contains a useful theorem about time interval set differences.

## 2 Previous Work

Resource allocation, and scheduling in general, is such an important problem that the literature addressing it is immense. The books [1, 2, 4] provide some important overviews. Most of the resource allocation literature looks at issues related to optimality. For instance, operation research approaches optimize some (simple) criterion function [5], normally the total time of the schedule, i.e., its *makespan*.

Artificial Intelligence (AI) and heuristic approaches [3, 8] attempt to do resource allocation in a quick and robust manner, without trying to achieve an optimal solution. The representation of availablity presented in this paper arose out of such work [7].

## 3 Mathematical Preliminaries

Let $\{O_i |\ i = 1, \ldots, N\}$ denote a set of *operations* and let $\{r_j |\ j = 1, \ldots, M\}$ be a set of *resources*.

Associated with each operation, $O_i$, is an non-negative integer $\mathsf{Res}(O_i)$ which denotes the number of resources required to perform the operation[1] and a positive real number $\mathsf{Time}(O_i)$ which represents the predicted duration of time for the operation.

We define a *time interval* $(s, e)$ as an interval on the real line with $e > s$. Define the time interval set difference as

$$(s,e) - (s',e') = \left\{ \begin{array}{ll} \{(s,e)\} & \text{if } (s,e) \cap (s',e') = \emptyset \\ \{(s,s')\} & \text{if } s \leq s' \leq e \leq e' \\ \{(e',e)\} & \text{if } s' \leq s \leq e' \leq e \\ \{(s,s'),(e',e)\} & \text{if } s \leq s' \leq e' \leq e \\ \emptyset & \text{if } s' \leq s \leq e \leq e' \end{array} \right\}$$

Recursively define the time interval set difference for a collection of time intervals as

$$(s,e) - \{(s_i,e_i)\}_{i=1}^{m} = \bigcup_{(s'',e'') \in (s,e)-(s_m,e_m)} \left[ (s'',e'') - \{(s_i,e_i)\}_{i=1}^{m-1} \right] \qquad (1)$$

Associated with each resource instance, $r$, is a set of non-overlapping time intervals $\mathsf{Up}(r)$, when the resource is up and available for assignment to an operation. We

[1] The extension of this work to operations which require multiple resource types is trivial.

say a resource instance is *up* during some arbitrary time interval $(s, e)$ if and only if there exists an interval $(s', e') \in \mathsf{Up}(r)$ such that $(s, e) \subseteq (s', e')$.

A scheduling algorithm solves the resource allocation problem by building and maintaining the assignment of resources to operations over time. We say operation $O_i$ *reserves* resource instance $r$ during $(s, e)$. Alternatively, we say $r$ is *assigned to* $O_i$ during $(s, e)$. An operation $O_i$ is *scheduled* when the scheduler reserves all the needed $\mathsf{Res}(O_i)$ resources for some valid time interval.

For each resource instance, we can define the relations:

$$\mathsf{Reservations}(r) = \{\langle O_i, (s, e)\rangle \quad | \quad \text{for all operations } O_i \text{ which reserve } r \text{ over the time interval } (s, e) \text{ such that } e - s = \mathsf{Time}(O_i)\}$$

and

$$\mathsf{RTimes}(r) = \{(s, e) | \ \exists \langle O_i, (s, e)\rangle \in \mathsf{Reservations}(r)\}$$

To satisfy the constraint on the meaning of $\mathsf{Up}(r)$, for all $(s, e) \in \mathsf{RTimes}(r)$, there must exist an $(s', e') \in \mathsf{Up}(r)$ such that

$$(s, e) \subseteq (s', e') \tag{2}$$

For a given resource instance $r$, its $\mathsf{Reservations}$ relation must follow the additional constraint that for all unequal pairs of entries, say $\langle O_i, (s_i, e_i)\rangle$ and $\langle O_k, (s_k, e_k)\rangle$,

$$(s_i, e_i) \cap (s_k, e_k) = \emptyset \tag{3}$$

We denote the cardinality of set $X$ as $|X|$.

# 4 Resource Availability

For the scheduler to assign an operation to a set of resources, it must be able to compute, for each resource, when that resource is available.

We say a resource instance is *available* over some time interval if and only if (a) that resource is up during that interval and (b) it is not already reserved by any other operation.

We now define, for each resource instance $r$, the set of time intervals $\mathsf{Avail}(r)$ by using the following algorithm:

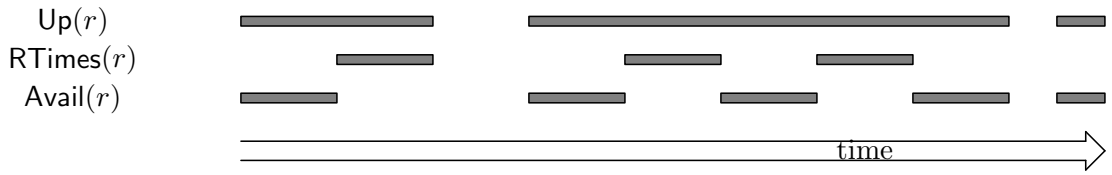


Figure 1: Graphical depiction of $\mathsf{Avail}(r)$

*Algorithm Avail*:
Let $\mathsf{Avail} = \emptyset$.
$\forall (s, e) \in \mathsf{Up}(r)$
  $\mathsf{Avail} = \mathsf{Avail} \cup [(s, e) - \mathsf{RTimes}(r)]$
Return Avail.

Figure 1 shows a depiction of the relationships between $\mathsf{Avail}(r)$, $\mathsf{RTimes}(r)$, and $\mathsf{Up}(r)$.

**Theorem 1** *For each resource instance $r$, $\mathsf{Avail}(r)$ contains the set of maximal intervals over which $r$ is available for assignment to an operation.*

*Proof for Theorem 1:* Let $(s'', e'') \subseteq (s', e') \in \mathsf{Avail}(r)$. We need to show that adding an operation and time interval pair with the time interval $(s'', e'')$ to $\mathsf{Reservations}(r)$ does not violate the resource availability constraints (2) and (3).

Constraint (2), that is, $\exists (s, e) \in \mathsf{Up}(r)$ such that $(s, e) \supseteq (s'', e'')$, holds, since, by the definition of $\mathsf{Avail}(r)$ and Theorem 12(a), $\exists (s, e) \supseteq (s', e')$ which implies the desired result.

Constraint (3), that is, $\forall (s, e) \in \mathsf{RTimes}(r)$, $(s, e) \cap (s'', e'') = \emptyset$, also holds, since, by the definition of $\mathsf{Avail}(r)$ and Theorem 12(b), $(s, e) \cap (s', e') = \emptyset$, for all $(s, e) \in \mathsf{RTimes}(r)$, which implies the desired result.

We now must prove that $\mathsf{Avail}(r)$ contains only maximal intervals. Again let $(s', e') \in \mathsf{Avail}(r)$. By Theorem 12(c), either $s'$ is the starting time of some interval in $\mathsf{Up}(r)$ or there exists in $\mathsf{RTimes}(r)$ some interval whose ending time is $s'$. In either case it is impossible for there to exist an interval in $\mathsf{Avail}(r)$ containing $(s', e')$ with a start time less than $s'$. By the analogous argument, there cannot exist an interval in $\mathsf{Avail}(r)$ containing $(s', e')$ whose end time is greater than $e'$. Thus $(s', e')$ is maximal. □

**Theorem 2** Algorithm Avail*'s complexity for computing the intervals in* $\mathsf{Avail}(r)$ *is* $O(\ |\mathsf{Up}(r)|\ |\mathsf{Reservations}(r)|\ )$.

*Proof for Theorem 2:* This is easy to see, as the algorithm calls for looping over the $|\mathsf{Up}(r)|$ and, for each of these, looping over the $|\mathsf{RTimes}(r)| = |\mathsf{Reservations}(r)|$ intervals. □

We will need the following theorem about the size of $\mathsf{Avail}(r)$ in subsequent proofs.

**Theorem 3** $\mathsf{Avail}(r)$ *can contain at most* $|\mathsf{Up}(r)| + |\mathsf{Reservations}(r)|$ *distinct intervals.*

*Proof for Theorem 3:* From the fact that intervals in $\mathsf{Up}(r)$ are non-intersecting, and the constraint that intervals in $\mathsf{RTimes}(r)$ must be contained in at least one interval in $\mathsf{Up}(r)$, we conclude that they must be contained in one and only one such interval. Subtracting a non-intersecting interval from another yields the beginning interval. Thus, any interval in $\mathsf{Up}(r)$ which does not intersect an interval in $\mathsf{RTimes}(r)$ yields one resultant interval (itself) in $\mathsf{Avail}(r)$. From the definition of set difference for two intervals, we note that the maximal number of intervals generated is at most two, when the subtracted interval is strictly contained within the other. Since $\mathsf{RTimes}(r)$ contains non-overlapping intervals, for any interval in $\mathsf{Up}(r)$ which does intersect one or more intervals in $\mathsf{RTimes}(r)$, each intersection results in at most one more than the number of intersecting intervals. Thus the number of resulting intervals is as desired. □

# 5 Updating of $\mathsf{Avail}(r)$

We now proceed to show that a more efficient method than *Algorithm Avail* exists for updating $\mathsf{Avail}(r)$ as new resource assignments are made.

To achieve efficiency, we desire to maintain an $\mathsf{Avail}(r)$ data structure rather than recomputing it as needed. To do so, we must update $\mathsf{Avail}(r)$ to account for newly reserved time intervals. Let $(s, e)$ be the time interval over which operation $O_i$ is to reserve resource $r$. To update $\mathsf{Avail}(r)$, we must

*Algorithm Add:*
Find the interval $(s', e') \in \mathsf{Avail}(r)$ such that $(s', e') \supseteq (s, e)$. If there is no such interval, then the scheduler has requested an invalid resource

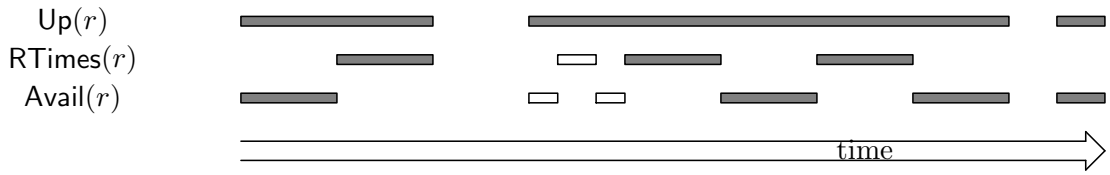


Figure 2: Graphical depiction of adding a reservation

assignment. Otherwise we must delete $(s', e')$ from $\mathsf{Avail}(r)$ and add back the intervals $(s', e') - (s, e)$.

Figure 2 shows a graphical depiction of adding a reservation interval to $\mathsf{RTimes}(r)$ and its effect on $\mathsf{Avail}(r)$ to the situation shown in Figure 1; Newly added intervals are shown unshaded.

**Theorem 4** *The adding scheme of* Algorithm Add *properly maintains* $\mathsf{Avail}(r)$.

*Proof for Theorem 4:* The change made for a new resource assignment is the addition of the time interval $(s, e) \in \mathsf{RTimes}(r)$. Since $(s, e)$ is a valid assignment interval, $\exists (s', e') \in \mathsf{Up}(r)$ such that $(s', e') \supseteq (s, e)$. Since intervals in $\mathsf{Up}(r)$ are non-intersecting, using the definition of $\mathsf{Avail}(r)$, the only interval which is changed is $(s', e')$. But, this change is precisely the set difference added back into $\mathsf{Avail}(r)$ by *Algorithm Add.* □

Likewise, let $(s, e)$ be the time interval over which operation $O_i$ reserves resource $r$ and we wish to unreserve it. To update $\mathsf{Avail}(r)$, we must:

*Algorithm Delete:*
Find the two intervals in $\mathsf{Avail}$, one with end time equal to $s$ and the other with start time equal to $e$, if they exist. Let the first interval be $(s', s)$ and the second be $(e, e')$. If neither exist, then add $(s, e)$ to $\mathsf{Avail}$, else if $(s', s)$ exists by not $(e, e')$, delete $(s', s)$ from $\mathsf{Avail}(r)$ and add the interval $(s', e)$, else if $(e, e')$ exists but $(s', s)$ doesn't, then delete $(e, e')$ from $\mathsf{Avail}(r)$ and add $(s, e')$, else delete both intervals and add the interval $(s', e')$ to $\mathsf{Avail}(r)$.

Figure 3 shows a graphical depiction of deleting a reservation interval from $\mathsf{RTimes}(r)$ and its effect on $\mathsf{Avail}(r)$ to the situation shown in Figure 1; Changed and deleted intervals are shown unshaded.

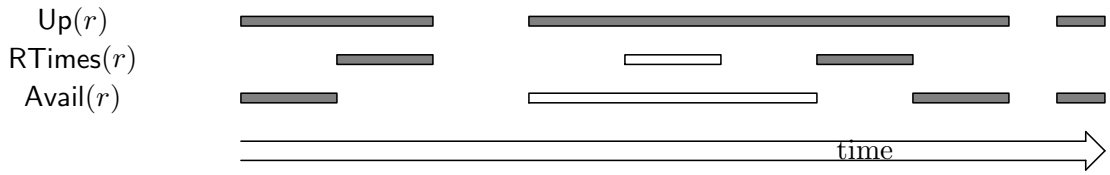


Figure 3: Graphical depiction of deleting a reservation

**Theorem 5** *The deleting scheme of* Algorithm Delete *properly maintains* $\mathsf{Avail}(r)$.

*Proof for Theorem 5:* The change made is the removal of the time interval $(s, e) \in \mathsf{RTimes}(r)$. By Theorem 12(c) and the definition of $\mathsf{Avail}(r)$, there may exists intervals in $(s', e') \in \mathsf{Avail}(r)$ with either $s' = e$ or $e' = s$. There are at most one of each such interval, since intervals in both $\mathsf{Up}(r)$ and $\mathsf{RTimes}(r)$ are non-overlapping.

In any of the existence cases, after deleting the retrieved intervals, the proper interval to add to $\mathsf{Avail}(r)$ is the interval formed by performing the set union of $(s, e)$ with the two (or less) retrieved intervals (filling the zero length "holes" of end points) since this is precisely what $(s, e)$ was subtracted out of in forming $\mathsf{Avail}(r)$. For instance, if neither interval exists, then $(s, e)$ totally covers some interval in $\mathsf{Up}(r)$. In all cases, *Algorithm Delete* returns the proper interval to $\mathsf{Avail}(r)$. □

# 6 Computational Efficient Maintenance of $\mathsf{Avail}(r)$

Naive implementations of the above modification algorithms might represent $\mathsf{Avail}(r)$ as a list of unordered time intervals. However, since $\mathsf{Avail}(r)$ contains non-intersecting intervals, it may be totally ordered and a height-balanced tree [6] used to implement it.

We then have the following theorems:

**Theorem 6** *The time complexity to find the maximal available time interval less than some time $t$ (greater than some time $t$, contained in some other interval) is proportional to $O(\log n)$, where $n = |\mathsf{Up}(r)| + |\mathsf{Reservations}(r)|$ when $\mathsf{Avail}(r)$ is represented as a height-balanced tree.*

*Proof for Theorem 6:* It is well known that finding the required node in a height-balanced tree is $O(\log n)$ for an $n$ node tree. In our case, $n = |\mathsf{Up}(r)|+|\mathsf{Reservations}(r)|$, by Theorem 3. □

**Theorem 7** Algorithm Add *has time complexity* $O(\log n)$*, where* $n = |\mathsf{Up}(r)| + |\mathsf{Reservations}(r)|$ *when* $\mathsf{Avail}(r)$ *is represented as a height-balanced tree.*

*Proof for Theorem 7:* It is well known that adding and deleting nodes from a height-balanced tree is $O(\log n)$ for an $n$ node tree. Also, the containment query operation is $O(\log n)$ by the above theorem. □

**Theorem 8** Algorithm Delete *has time complexity* $O(\log n)$*, where* $n = |\mathsf{Up}(r)| + |\mathsf{Reservations}(r)|$ *when* $\mathsf{Avail}(r)$ *is represented as a height-balanced tree.*

*Proof for Theorem 8:* As above, the only operations on Avail required by *Algorithm Delete* are adding, deleting, and finding the nodes based on the sort key. Thus by Theorem 3, the result holds. □

We now state and prove the main Theorem on maintaining $\mathsf{Avail}(r)$:

**Theorem 9** $\mathsf{Avail}(r)$ *may be maintained in* $O(n)$ *space and updated in* $O(\log n)$*, where* $n = |\mathsf{Up}(r)| + |\mathsf{Reservations}(r)|$ *when* $\mathsf{Avail}(r)$ *is represented as a height-balanced tree.*

*Proof for Theorem 9:* The space result is an immediate consequence of Theorem 3. The time result is an immediate consequence of Theorems 7 and 8. □

# 7 Using Multiple Resources

In general, an operation, $O$, may require reserving more than one resource, i.e., $\mathsf{Res}(O) > 1$. Thus, it is imperative that a scheduler be able to quickly compute the availablity of sets of resources. In this section, we show how the height-balanced tree representation of $\mathsf{Avail}(r)$ leads to an efficient algorithm for this computation.

First, we define in the obvious fashion the intersection of two sets of time intervals $A$ and $B$ as:

$$\{(s,e)|\ \exists(s_i,e_i) \in A \text{ and } \exists(s_j,e_j) \in B \text{ s.t. } (s,e) = (s_i,e_i) \cap (s_j,e_j)\}$$

The following theorem contains our main result:

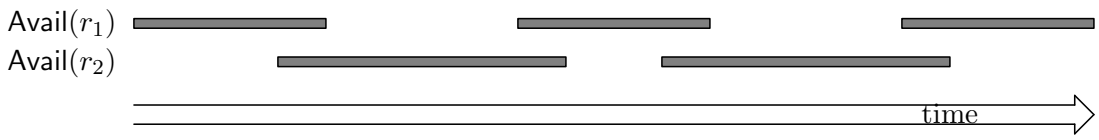


Figure 4: "Weave" pattern producing maximum number of intersections

**Theorem 10** *For $K$ resource instances $r_1, \ldots, r_K$, $\bigcap_{i=1}^{K} \mathsf{Avail}(r_i)$ is of $O(\sum_{i=1}^{K} n_i)$ cardinality and may be computed in $O(\sum_{i=1}^{K} n_i)$ time, where $n_i = |\mathsf{Avail}(r_i)|$.*

*Proof for Theorem 10:*

We first prove the size result, which we will do in detail for the case $K = 2$, with the result for general $K$ obvious by extension. We will show that the maximum number of intervals in the intersection of $\mathsf{Avail}(r_1)$ and $\mathsf{Avail}(r_2)$ is $O(n_1 + n_2)$. For any $(s, e) \in \mathsf{Avail}(r_1)$, the possible ways by which any $(s', e') \in \mathsf{Avail}(r_2)$ may generate an interval in the intersection set are:

**(a)** $(s, e) \subseteq (s', e')$. Thus $(s, e)$ is in the intersection.

**(b)** $(s', e') \subseteq (s, e)$. Thus, $(s', e')$ is in the intersection.

**(c)** The two intervals partially overlap and either $(s', e)$ or $(s, e')$ is in the intersection set, but not both.

Thus every interval in the intersection set must have as its start time a start time from some interval in $\mathsf{Avail}(r_1)$ or from $\mathsf{Avail}(r_2)$. Since there are $n_1 + n_2$ such start times, the order of the cardinality of the intersection set is no greater than that value.

Figure 4 demonstrates the the largest intersection set arises when two sets are related in the "weave" pattern shown. If $n_1 \leq n_2$, there can exist at most $(n_1 - 1)$ such weaves. Each such weave produces two intervals in the intersection set. The remaining $(n_2 - (n_1 - 1))$ intervals in $\mathsf{Avail}(r_2)$ can produce at most a single entry in the intersection set. Thus the maximum size of the intersection set is $n_1 + n_2 - 1$.

We now prove the time bound by exhibiting an algorithm with the proper time complexity. The basic idea of the algorithm is to run a sweep through the (sorted) merge of the sorted time intervals, adding an intersection interval to the result only when appropriate. We will sketch the algorithm next.

Since each of the Avails is represented as a height-balanced tree, we may view these as a priority queue and find the minimum intervals in $\sum \log n_i$ time. We start with a count variable equal to one and a sweep variable equal to the minimum start time. We then loop, performing the following tasks until we have been through all the intervals in all of the Avails. First, we update sweep to be the next time[2] (from the start or end of one of the $n$ minimum intervals) greater than sweep. If sweep is an end time, we decrement count by one and replace the interval we just passed over with the next one in its corresponding Avail. On the other hand, if sweep is a start time, we increment count by one. If count $= K$, we have the start of an intersection interval among the $K$ Avails. Thus we find the next time among the intervals greater than sweep (it must be an end time), add this start and end pair to our result, and perform the above described tasks on finding an end time.

Finding the minimum times in each iteration step is an $O(K)$ operation, while finding the next items in the priority queues is a constant time operation, since we maintain our place in the height-balanced trees. Since there are $O(\sum n_i)$ total intervals to loop through, we have the desired result. □

We now state and prove our main result on computing the availablity of multiple resource instances:

**Theorem 11** *For $K$ resource instances $r_1, \ldots, r_K$, $\bigcap_{i=1}^{K} \mathsf{Avail}(r_i)$ may be computed in time linear in the number of scheduled operations. This is optimal.*

*Proof for Theorem 11:* Note that, after $n$ operations have been scheduled, we have

$$\sum_{j}^{M} |\mathsf{Reservations}(r_j)| = \sum_{i}^{n} \mathsf{Res}(O_i) < Cn$$

for some constant $C$. Thus from Theorem 10 we know that the cardinality of intersection is $O(n)$ and that this may be computed in $O(n)$ time. Since we must process each intersection interval at least once, this is optimal. □

# 8 Summary

This paper has given a mathematical description of resource availablity. We defined a naive algorithm for computing availablity and then described methods to maintain

[2]Ties among times from intervals are always resolved by first considering all those arising from being the end of an interval before those arising from being the start of an interval.

availability information as new reservations/assignments are made. By using a height-balanced tree to represent availablity information, we have shown log time query, add, and delete operations. These lead to an algorithm for computing the availability of multiple resources in time proportional to the number of scheduled operations. This algorithm is optimal.

# 9 Appendix

We use the following simple lemma about time interval set differences in a number of places.

**Theorem 12** *Let* $(s', e') \in (s, e) - \{(s_i, e_i)\}_{i=1}^{m}$. *Then,* (a) $(s', e') \subseteq (s, e)$ *and* (b) $\forall i, (s', e') \cap (s_i, e_i) = \emptyset$ *and* (c) *ether* $s' = s$ *or* $s' = e_i$, *for some* $i$ *(and the analogous result holds for* $e'$*).*

*Proof for Theorem 12:* Induction will be used to prove each clause. Examination of the set difference of two time intervals revels that (a), (b), and (c) hold for the $m = 1$ case.

For the $m + 1$ general case, $(s', e')$ must be contained in one of the disjoint union making up the set difference definition, say $(s'', e'') - \{(s_i, e_i)\}_{i=1}^{m}$, where $(s'', e'') \in (s, e) - (s_{(m+1)}, e_{(m+1)})$.

By the inductive hypothesis, $(s', e') \subseteq (s'', e'') \subseteq (s, e)$, thus proving case (a).

To prove case (b), by the inductive hypotheis, $\forall i = 1, \ldots, m$, $(s', e') \cap (s_i, e_i) = \emptyset$. Using case (a), $(s', e') \subseteq (s'', e'')$, while from the definition of the set difference of two time intervals, $(s'', e'') \cap (s_{(m+1)}, e_{(m+1)}) = \emptyset$, which implies $(s', e') \cap (s_{(m+1)}, e_{(m+1)}) = \emptyset$.

By the inductive hypothesis, either there exists an $i$ such that $s' = e_i$ or $s' = s''$. In the former case, we are done. In the later, from the definition of the set difference of two time intervals, either $s'' = s$ or $s'' = e_{(m+1)}$. This proves case (c). □

# References


[1] K. R. Baker. *Introduction to Sequencing and Scheduling.* John Wiley & Sons, 1974.

[2] E.G. Coffman and P.J. Denning. *Operating Systems Theory*. Prentice-Hall, Englewoo Cliffs, New Jersey, 1973.

[3] B.R. Fox and K.G. Kempf. Reasoning about opportunistic schedules. In *Proceedings 1987 IEEE Interational Conference on Robotics and Automation*, pages 1876–1882, 1987.

[4] S. French. *Sequencing and Scheduling: An Introduction to the Mathematics of the Job-Shop*. Ellis Horwood Limited, West Sussex, England, 1982.

[5] T. Ibarkai and N. Katoh. *Resource Allocation Problems: Algorithmic Approaches*. The MIT Press, Cambridge, Massachusetts, 1988.

[6] D.E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley, Reading, Mass., 1973.

[7] S.P. Smith. Scheduling using schedule modification operators. In *Working Notes for the Workshop on Manufacturing Production Scheduling*. AAAI–SIGMAN, August 1989.

[8] R.J. Stokey. AI factory scheduling: Multiple problem formulations. *SIGART Newsletter*, 1(110):27–30, October 1989.