Discrete Applied Mathematics 18 (1987) 1-8
North-Holland

SCHEDULING JOBS WITH FIXED START AND END TIMES

Esther M. ARKIN and Ellen B. SILVERBERG
Department of Operations Research, Stanford University, Stanford, CA 94305, USA

Received 7 August 1985
Revised 28 July 1986

We analyze a scheduling problem in which each job has a fixed start and end time and a value. We describe an algorithm which maximizes the value of jobs completed by k identical machines. The algorithm runs in time $O(n^2 \log n)$, where n is the number of jobs. We also show that the problem is NP-complete under the following restriction: Associated with each job is a subset of the machines on which it can be processed. For a fixed number of machines k, an $O(n^{k+1})$ time algorithm is presented.

Keywords. Clique, graph coloring, interval graphs, NP-complete, perfect graphs, scheduling problems, shortest paths.

1. Introduction

The theory of scheduling encompasses a multitude of problems. Variations that have received much attention include looking at different machine environments (single machine, parallel machines, open shop, flow shop, job shop), job characteristics (independent vs. precedence constrained), and optimality criteria (flow time, maximum lateness, total tardiness, makespan). In this paper, we discuss a simple and natural variant. There are n jobs to be processed by k machines. A machine can execute at most one job at a time. Each job i has a fixed start and a fixed end time (s_i and t_i respectively). Thus the processing time of job i is equal to $t_i - s_i$. Clearly, no pre-emption is allowed in our model. Each job i has a positive value w_i. There are no precedence constraints other than those implied by the start and end times. We seek to select a feasible subset of the jobs maximizing the total value of jobs to be processed. Each job is to be processed once or not at all.

In Section 2 we consider the problem in which all machines are identical, with each machine being able to execute each job. We formulate the problem as a binary integer linear program and show that the constraint matrix is totally unimodular, thereby proving the problem is polynomial. We describe the similarities between this problem and that of coloring interval graphs. Using the properties of interval graphs we establish an $O(n^2 \log n)$ time algorithm for our problem.

In Section 3 we consider the case in which the machines are no longer identical. More specifically, associated with each machine is a subset of the jobs which it can process. This imposes an additional restriction on the subsets of jobs that can be processed feasibly. As often happens in this field, the added restriction makes this variant of the problem NP-complete. In fact, the problem remains NP-complete even if all values are equal and we are asked to determine whether all jobs can be scheduled. We exhibit a reduction of 3-SAT to the latter problem, proving this result.

Finally, in Section 4 we show that the problem treated in Section 3 becomes tractable again when the number of machines k is considered to be fixed. An $O(n^{k+1})$ time algorithm is presented.

2. Scheduling with identical machines

The first problem we consider is that of finding an optimal schedule for n jobs on k identical machines. Each job has a positive value reflecting the benefit gained by completing this job. Also, associated with each job is a fixed start and end time, between which it would be continuously processed (no pre-empting allowed) if it is selected. We seek to find a 'legal' schedule (one consistent with all start and end times and which schedules at most one job on each machine at any given time), maximizing the value of jobs completed.

This problem can be reformulated in terms of coloring interval graphs (for definitions see [7]). Let each job be an interval on the real line whose endpoints are specified by the start and end times of the job. Each interval has a value, (that of its corresponding job). Given k colors, our objective is to maximize the value of the intervals legally colored. The more well-studied problem of coloring interval graphs is to legally color the graph using the minimum number of colors. Although linear-time algorithms are known for this problem, it is not clear how they extend to our problem. As an aside, notice that this extension is not always possible for general graphs, e.g., determining whether or not a graph is two-colorable is easy; but given two colors and a graph with nodes of equal value, it is NP-complete to find the maximum value of legally colored nodes [12], [13].

We now formulate the problem as a binary integer linear program:

$$\begin{array}{ll} \text{Maximize} & w^{\mathrm{T}} x \\ \text{subject to} & Ax \leq e^{\mathrm{T}} k, \\ & x \in \{0, 1\}, \end{array}$$

where w is an n-vector representing the value of the jobs (or intervals), x is a binary n-vector in which $x_j = 1$ implies that job j is to be executed ($x_j = 0$ otherwise), A is an $m \times n$ binary matrix in which m is the number of maximal cliques, where $A_{ij} = 1$ if interval j is in the ith maximal clique ($A_{ij} = 0$ otherwise), and e is an m-vector of 1's.

Lemma. *A polynomial algorithm exists for the problem of scheduling identical machines.*

Proof. In the above formulation, notice that the columns of matrix A have consecutive 1's since for interval graphs there is an ordering of the maximal cliques such that each vertex occurs in consecutive cliques [7]. Thus the matrix A is totally unimodular. Also note that the number of maximal cliques in an interval graph is $O(n)$. Therefore, we can relax the integrality constraint and solve the problem as a linear program in polynomial time [8]. For a more detailed explanation see the proof of Theorem 1 below. □

In fact, we suggest a more efficient algorithm which takes advantage of the special structure of the problem.

Theorem 1. *Given a set of n jobs* $J=\{J_1,\dots,J_n\}$, *the value* w_i *of each job* J_i, *and the start and end times* (s_i, t_i) *of each job* J_i, *the problem of finding a schedule to maximize the value of scheduled jobs on k identical machines can be done in* $O(n^2 \log n)$ *time for any given k.*

Proof. We reformulate the problem again, this time as an instance of minimum cost flow (see [10] for definition). First, notice that since interval graphs are perfect [7], the largest clique that can be colored is of size k. Thus our problem is equivalent to that of deleting a subset of intervals of smallest value such that all remaining cliques are of size k or less. To this end, we identify all maximal cliques $q_1,\dots,q_r$ of the interval graph. Assume the maximal cliques are linearly ordered according to time. Then, by an alternate definition of interval graphs, each job is contained in consecutive maximal cliques [7].

We construct a directed graph G as follows: Create nodes $v_0,\dots,v_r$ and arcs (v_i, v_{i-1}) for $i=1,\dots,r$. The costs on these arcs equal 0 and their capacities are infinite. We think of each arc (v_i, v_{i-1}) as representing clique q_i. For each job, if J_i is in cliques $q_j,\dots,q_{j+l}$, add an arc (v_{j-1}, v_{j+l}) of cost w_i and capacity 1. For each clique j that is not of maximum size, we add an arc (v_{j-1}, v_j) of cost 0 and capacity equal to (maximum-clique-size - size-of-q_j). We think of these as 'dummy' jobs. If we consider node v_0 to be a source S and node v_r to be a sink T and we require a flow from S to T of (maximum-clique-size - k), then we have formulated our problem as a min-cost flow. (See Fig. 1a, b for an example of this construction.) An optimal solution to this problem is equivalent to an optimal solution to our original problem: If an arc corresponding to job J_i has a non-zero flow on it in the min-cost flow solution, then we do not process this job. Thus we do not process a subset of jobs such that all remaining cliques are of size k or less. This subset is of smallest possible value since it corresponds to a min-cost flow solution.

To compute the complexity of this algorithm, we count the number of nodes and arcs. The number of maximal cliques in an interval graph is $O(n)$, thus the number of nodes in G is also $O(n)$. Note that the maximal cliques can easily be found in $O(n \log n)$ time and in $O(n)$ time if all endpoints are presorted. Clearly, the number of arcs is $O(n)$ as well. Min-cost flow problems can be solved in time proportional

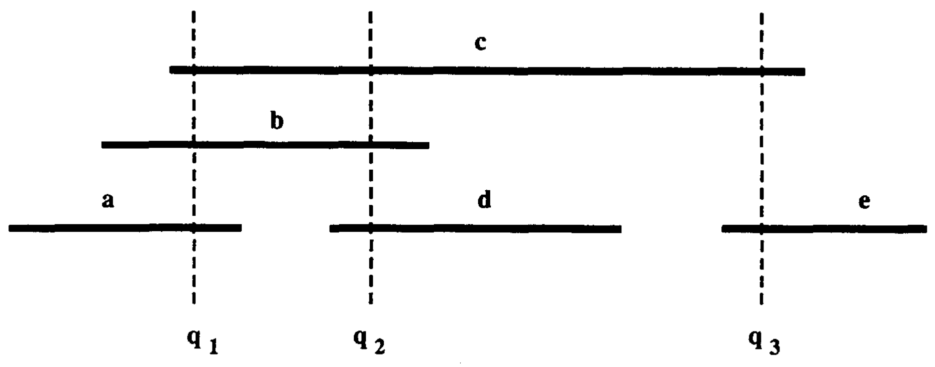


Fig. 1a. The jobs.

to (required-flow × complexity-of-shortest-path-algorithm). In our case, the required flow is $O(n)$. Edmonds and Karp have shown that it can be arranged for the shortest path computation to be carried out over nonnegative arc lengths [2]. Thus Dijkstra's shortest path algorithm can be applied [1]. For a graph with $|E|$ edges and $|V|$ nodes, the complexity of Dijkstra's algorithm is $O(|E|+|V|\log|V|)$ [4]. In our problem, both $|V|$ and $|E|$ are $O(n)$; therefore, the overall complexity of our algorithm is $O(n^2 \log n)$, concluding the proof of the theorem. □

Notice that in solving this problem for some k, we actually have solved the problem for any given number of machines greater than or equal to k. This can be seen by examining a min-cost flow algorithm which builds up flows (see [11]). At each step, an increase of the flow through the graph can be interpreted as a decrease in the number of available machines. Thus, the $O(n^2 \log n)$ algorithm solves the problem for all possible k in one execution.

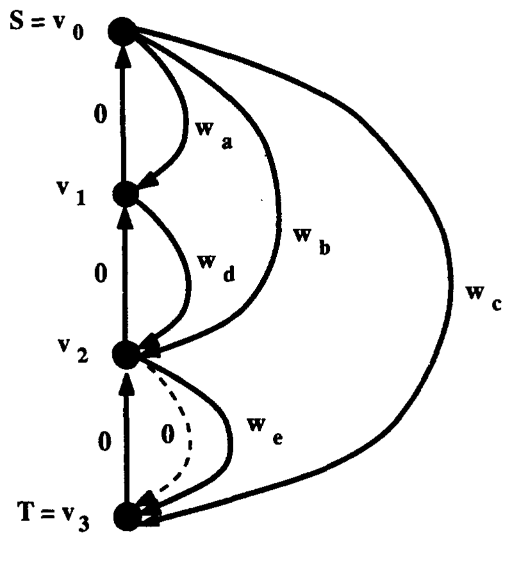


Fig. 1b. The directed graph.
Note: The dotted arc represents a dummy job.

3. Scheduling with non-identical machines is NP-complete

We now turn to the problem in which the machines are non-identical: Associated with each job is a subset of the machines on which it can be processed. (This specifies a job-machine mapping.) We show the following:

Theorem 2. *Given a set $J=\{J_1,\ldots,J_n\}$ of n jobs of equal value, the start and end times (s_i,t_i) of each job J_i, and a job-machine mapping between J and a set of k non-identical machines, it is NP-complete to determine whether all jobs can be processed.*

Proof. It is easy to see that the problem is in the class NP (see [5], [11] for definitions). To show completeness, we shall reduce the problem of satisfiability of Boolean expressions with three literals per clause (3-SAT) to the problem at hand. In 3-SAT we are given m variables, $v_1,\ldots,v_m$, and r clauses $C_1,\ldots,C_r$, with each clause C_j consisting of three literals. A literal is a variable or a negation thereof. We are asked to find an assignment of **true** or **false** to each variable, such that each clause contains at least one **true** literal.

Given such an instance of 3-SAT we construct the following scheduling problem: For each clause C_j, we have a job A_j with start time $s_j=j-1$ and end time $t_j=j$. For each variable v_i, we have a job B_i with $s_i=0$ and $t_i=r$, and we have two machines M_i and $\bar{M}_i$ corresponding to the variable and its negation (respectively). Job B_i can be processed either on machine M_i or $\bar{M}_i$. Machine M_i can process job A_j if variable v_i is a literal in clause C_j. (Similarly, machine $\bar{M}_i$ can process job A_j if $\bar{v}_i$, the negation of variable v_i, is in clause C_j.)

Claim. *A schedule exists in which all jobs are processed if and only if a truth assignment exists to satisfy the formula.*

Proof. Suppose there is a schedule in which all jobs are processed. We construct a truth assignment as follows: If the schedule assigns job B_i to machine M_i (resp., $\bar{M}_i$), then variable v_i is assigned a value of **false** (resp., **true**). Clearly, this assignment is unique. The remaining machine $\bar{M}_i$ (resp., M_i) is now 'free' to process the jobs A_j corresponding to clauses C_j in which $\bar{v}_i$ (resp., v_i) appears as a literal. This is a satisfying truth assignment since each clause C_j must have a true literal corresponding to the machine that processes job A_j.

Conversely, suppose a truth assignment exists. We assign jobs to processors as follows: If variable v_i has the value **true** (resp., **false**), let machine M_i (resp., $\bar{M}_i$) process all jobs A_j for which C_j contains the literal v_i (resp., $\bar{v}_i$). (If a job is scheduled to be processed by more than one machine, pick one arbitrarily.) The remaining machine $\bar{M}_i$ (resp., M_i) is now 'free' to process job B_i. Clearly, all jobs can be processed in this manner, which completes the proof of the claim and the theorem. □

4. A polynomial algorithm for a fixed number of machines

As seen in the previous section, the problem of scheduling non-identical machines is NP-complete. However, if k, the number of machines, is considered to be fixed, then the problem is solvable in polynomial time, even if we allow the jobs to have non-equal values. We describe an $O(n^{k+1})$ algorithm for this case.

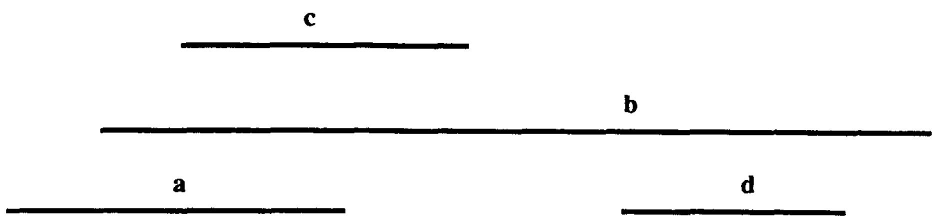
First, consider the simple case of a single machine ($k=1$). We construct the following directed acyclic graph (DAG): For each job i we have a node v_i. There is an arc from v_i to v_j if the end time of job i is earlier than the start time of job j, i.e., job i can be completed before job j is to begin. The 'length' of this arc is equal to the value of job j. We also have nodes S and T, the source and terminal respectively. We connect S to each node v_i with an arc of 'length' equal to the value of job i. Then we connect each v_i to T with an arc of 'length' 0. Clearly, finding a schedule that maximizes the value of completed jobs is equivalent to finding the longest path in the constructed DAG. This can be done in $O(n^2)$ time since the digraph is acyclic [10].

Next, we generalize this result in the following:

Theorem 3. *Given a set $J=\{J_1,\ldots,J_n\}$ of n jobs, the value w_i of each job J_i, the start and end times (s_i, t_i) of each job J_i, and a job-machine mapping, the problem of finding a schedule to maximize the value of processed jobs on a fixed number of machines k can be solved in $O(n^{k+1})$ time.*

Proof. The construction is similar to the single-machine case. Define 'events' to be the start or end times of each job. We may assume there are $2n$ distinct events. (If two or more events coincide, they can be perturbed to guarantee this fact.) For each event we construct nodes corresponding to legal combinations of jobs and the machines on which they are being processed at that time. We can think of these as k-vectors representing the status of the machines. We create a starting and an ending event and their corresponding nodes, S and T. These are k-vectors in which all machines are idle. Suppose all nodes and arcs up to event m have already been constructed. We construct the next 'layer': If event m represents the start time of J_i, then we connect all nodes at events prior to m in which a machine that can process J_i is idle to corresponding nodes at event m by an arc of 'length' equal to w_i. If event m represents the end time of job i, we connect all nodes at events previous to m in which J_i is being processed to corresponding nodes at event m by an arc of length 0. (See Fig. 2a, b for an example of this construction.)

To count the number of nodes in the DAG, we note that nodes in which all the machines are busy cannot repeat, whereas nodes in which one or more machines are idle can repeat up to $2n$ times. There are $O(n^k)$ nodes of the first type and at most $O(n^{k-1})$ nodes of the second type. Thus we have at most $O(n^k)$ nodes in the DAG. To count the number of arcs in the DAG, note that out of each node we can have two types of arcs: One corresponding to the start of jobs and one corresponding to



job:	machines	weight
a:	1	$w_a = 7$
b:	1,2	$w_b = 8$
c:	2	$w_c = 4$
d:	1	$w_d = 3$

Fig. 2a. The jobs.

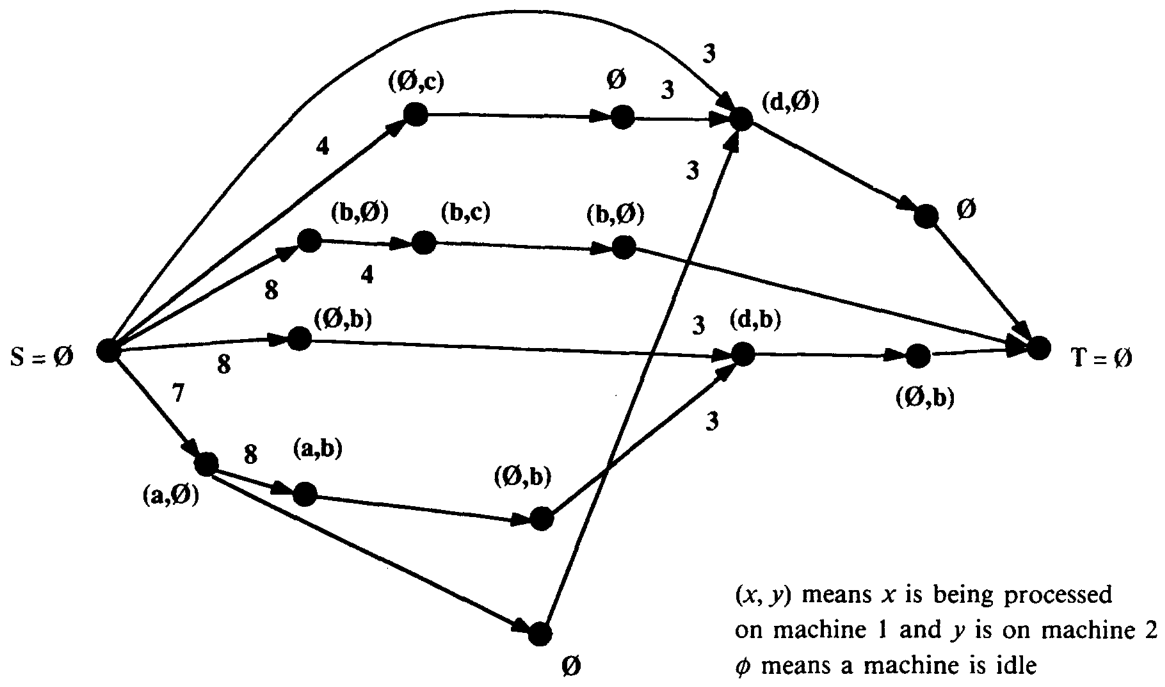

S = Ø
T = Ø
(Ø,c)
Ø
(d,Ø)
(b,Ø)
(b,c)
(b,Ø)
Ø
(Ø,b)
(d,b)
(Ø,b)
(a,Ø)
(a,b)
(Ø,b)
Ø
3
3
4
3
4
8
8
3
7
8
3
(x, y) means x is being processed on machine 1 and y is on machine 2
ϕ means a machine is idle

Fig. 2b. The DAG.

the end of jobs. There can be at most $k \times n$ arcs of the first type and one arc of the second type leaving any node. Thus, we have $O(n^{k+1})$ arcs in the DAG. An optimal schedule will be equivalent to a longest path in this DAG. Since our digraph is acyclic we can solve the longest path problem by replacing each arc length by its negative value and finding the shortest path. This fact also allows us to again use the idea of [2] to carry out the shortest path computation over an equivalent network with non-negative arc lengths. A recent result by Fredman and Tarjan has shown that for a graph $G = (V, E)$ Dijkstra's algorithm can be implemented to run in time $O(|V| \log |V| + |E|)$ [4]. This yields an $O(n^{k+1})$ algorithm to find the longest path in our original DAG which, in turn, yields an optimal schedule, completing the proof of Theorem 3. □

5. Final remarks

We have also shown by an easy reduction from 1-in-3-SAT that an extension of the problem in Section 2 (scheduling with identical machines) to include precedence constraints becomes NP-complete. The precedence constraints we refer to are of the form: Pairs of jobs (J_i, J_j) such that J_j can be processed only if J_i is processed.

The version of the coloring problem discussed in Section 2 has been studied for comparability graphs and their complements, thus including interval graphs. An algorithm is given in [3] based on a characterization theorem by [6] for this more general family of graphs. However, our algorithm is more efficient for the specific case addressed here.

Acknowledgment

We wish to thank our advisor, Christos Papadimitriou, for many useful suggestions which led to the solution of the problem. We'd also like to thank Mark Perkins for suggesting the problem, and Joseph Mitchell, Mark Perkins and Arthur F. Veinott, Jr. for helpful discussions on the subject. This research was supported by the National Science Foundation Grant DCR 83-20000.

Related work on interval scheduling problems has been considered independently by A. Kolen, J.K. Lenstra, C. Papadimitriou and J. Orlin (see [9]).

References

[1] E.W. Dijkstra, A note on two problems in connexion with graphs, Numer. Math. 1 (1959) 269–271.

[2] J. Edmonds and R.M. Karp, Theoretical improvements in algorithmic efficiency for network flow problems, J. Assoc. Comput. Mach. 19 (2) (April 1972) 248–264.

[3] A. Frank, On chain and antichain families of a partially ordered set, J. Combin. Theory, (Ser. B) 29 (1980) 176–184.

[4] M.L. Fredman and R.E. Tarjan, Fibonacci heaps and their uses in improved network optimization algorithms, Proc. 25th IEEE Symp. on the Foundations of Computer Science, 1984.

[5] M.R. Garey and D.S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness (Freeman, San Francisco, 1979).

[6] C. Greene and D.J. Kleitman, The structure of Sperner k-families, J. Combin. Theory (Ser. A) 20 (1976) 41–68.

[7] M.C. Golumbic, Algorithmic Graph Theory and Perfect Graphs (Academic Press, New York, 1980).

[8] L.G. Khachiyan, A polynomial algorithm in linear programming, Dokl. Akad. Nauk USSR 244 (5) (1979) 1093–1096. (English translation: Soviet Math. Dokl. 20 191–194.)

[9] A. Kolen, J.K. Lenstra, C. Papadimitriou and J. Orlin, Interval scheduling problems, Manuscript, Centre of Mathematics and Computer Science, C.W.I., Kruislaan 413, 1098 SJ Amsterdam.

[10] E.L. Lawler, Combinatorial Optimization: Networks and Matroids (Holt, Rinehart and Winston, New York, 1976).

[11] C.H. Papadimitriou and K. Steiglitz, Combinatorial Optimization: Algorithms and Complexity (Prentice-Hall, Englewood Cliffs, NJ, 1982).

[12] M. Yannakakis, The node deletion problem for hereditary properties, Report No. TR-240, Computer Science Laboratory, Princeton University, Princeton, NJ.

[13] M. Yannakakis, Node- and edge-deletion NP-complete problems, Proc. 10th Ann. ACM Symp. On Theory of Computing (Assoc. Comput. Mach., New York) 253–264.