Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Discrete Mathematics 25 (2006) 139–142
Electronic Notes in DISCRETE MATHEMATICS
www.elsevier.com/locate/endm

A dynamic programming algorithm for the single-machine scheduling problem with deteriorating processing times

Alberto Bosio

Dipartimento di Tecnologie dell'Informazione
Università degli Studi di Milano
via Bramante 65, 26013 Crema, Italy

Giovanni Righini [1]

Dipartimento di Tecnologie dell'Informazione
Università degli Studi di Milano
via Bramante 65, 26013 Crema, Italy

Keywords: combinatorial optimization, dynamic programming, scheduling

1 Introduction

Single-machine scheduling problems with deteriorating processing times were introduced by Browne and Yechiali [1], who considered random processing times increasing over time and presented an optimal scheduling policy to minimize the expected makespan. Cheng and Ding [2] studied the complexity of the single-machine scheduling problem with deterministic and linearly increasing/decreasing processing times in which release dates are also present; they proved that the makespan minimization problem with arbitrary release times and identical processing rates or identical normal processing times is strongly NP-hard and the problem remains at least NP-hard even if only one job has a non-zero release time.

[1] Email: righini@dti.unimi.it

doi:10.1016/j.endm.2006.06.079

In this paper we consider the single-machine scheduling problem with release times and linearly deteriorating processing times. An application to fire fighting was recently described by Rachaniotis and Pappis [3]. For this strongly NP-hard problem [2] we present a dynamic programming algorithm coupled with upper bounding and lower bounding techniques to compute exact solutions and we prove its effectiveness on a large set of randomly generated instances. We also study how the computing time is related to the correlation between the nominal processing times, their rate of growth and the release times.

2 Problem statement

A set $\mathcal{N}$ of N jobs must be processed without preemption on a single machine. Each job $j \in \mathcal{N}$ is characterized by three data: a non-negative release time r_j, a non-negative nominal processing time p_j and a non-negative rate of growth a_j. The processing time d_j of job j is given by $d_j = p_j + a_j(t_j - r_j)$, where t_j is the starting time of the job, which cannot be less than r_j. A solution consists of a sequence of jobs, which can include idle times, and the objective is to minimize the makespan, that is the completion time of the last job in the sequence.

The special case in which $r_j = 0$ for each job $j \in \mathcal{N}$ is polynomially solvable by scheduling the jobs in increasing order of the ratio p_j/a_j (see [1]).

3 A dynamic programming algorithm

Our dynamic programming algorithm iteratively extends a partial solution adding a new job to it. A partial solution is a feasible sequence of jobs and jobs are added to the end of the sequence in chronological order. In dynamic programming terminology each partial solution corresponds to a *state*. A state is a tuple $(\mathcal{S}, t)$, where $\mathcal{S}$ is the subset of jobs already scheduled and t is the makespan for that subset, that is the minimum time required to scheduled all jobs in $\mathcal{S}$. Each state is extended in all possible ways, by adding one of the unscheduled jobs. When an unscheduled job $j \notin \mathcal{S}$ is added to extend a state $(\mathcal{S}, t)$, a new state $(\mathcal{S}', t')$ is generated, where $\mathcal{S}' = \mathcal{S} \cup \{j\}$ and $t' = s_j + d_j + a_j(s_j - r_j)$ with $s_j = max\{t, r_j\}$.

Domination. A newly generated state is kept only if it is non-dominated; otherwise it is fathomed because it cannot lead to an optimal solution. The algorithm proceeds until all non-dominated states have been extended and only a final state $(\mathcal{N}, t^*)$ survives; then t^* is the optimal value. A state $(\mathcal{S}_1, t_1)$

dominates a state $(\mathcal{S}_2, t_2)$ only if $\mathcal{S}_2 \subseteq \mathcal{S}_1$ and $t_2 \geq t_1$ and at least one of the two inequalities is strict. Another domination criterion is the following: consider the extension of a state $(\mathcal{S}, t)$ to several successor states and let $(\mathcal{S} \cup \{i\}, t_i)$ and $(\mathcal{S} \cup \{j\}, t_j)$ be two of them. If $t_i \leq r_j$, then job j is not subject to any delay for being scheduled after job i. Hence every state generated from $(\mathcal{S} \cup \{j\}, t_j)$ can be improved by inserting job i before job j. Therefore state $(\mathcal{S} \cup \{j\}, t_j)$ can be fathomed. For this reason when our algorithm extends a state, it generates the successor states according to the release time order of the unscheduled jobs; the minimum time value t_{min} of the newly generated states is kept in memory and the generation of new states is stopped as soon as their release dates become larger than or equal to t_{min}.

The following property relies on a result due to Browne and Yechiali [1]:

Property 1. *Whenever the end time of the last scheduled job is not less than the largest release time, then the remaining subproblem is polynomially solvable by scheduling all the unscheduled jobs in non-decreasing order of* $\frac{p_i}{a_i} - r_i$.

This *basic order* is computed once in a preprocessing step before dynamic programming starts. When a state $(\mathcal{S}, t)$ must be extended and $t \geq max_{j \in \mathcal{N} \setminus \mathcal{S}}\{r_j\}$, the algorithm, exploiting Property 1, directly generates a final state by optimally scheduling all the remaining jobs in one shot.

Upper and lower bounds. We have enriched our dynamic programming algorithm with bounds to early fathom unpromising states. An upper bound to the optimal value is computed by means of a heuristic; two lower bounds to the cost of the solutions that can be obtained from any state $(\mathcal{S}, t)$ are computed and compared to the incumbent upper bound: if any of them is found to be greater than the upper bound, the state is fathomed.

4 Computational results

We designed our data-set in order to study how the time consumption of the algorithm depends on the data, i.e. the values of p, a and r data. We generated random instances in which the values of the data were selected at random from different ranges. After generating the data, we assigned them to the jobs, in order to maximize the "difficulty" of the resulting instance: to this purpose we enforced the constraint that for each pair of jobs, $p_i < p_j$ implies $a_i < a_j$; we also assigned the release dates in such a way that $r_i \geq r_j$ whenever $\frac{p_i}{a_i} < \frac{p_j}{a_j}$. In this way it is not trivial to decide whether any job i should better precede or follow any other job j. For the most difficult instances we could generate, the computing time never exceeded 6 seconds for $N = 20$.

References

[1] S. Browne, U. Yechiali, *Scheduling deteriorating jobs on a single processor*, Operations Research 38 (1990) 495-498

[2] T.C.E. Cheng, Q. Ding, *The complexity of scheduling starting time dependent tasks with release times*, Information Processing Letters 65 (1998) 75-79

[3] N.P. Rachaniotis, C.P. Pappis, *Scheduling fire fighting tasks using the concept of deteriorating jobs*, University of Piraeus, Department of Industrial Management, 2005.