J Sched (2009) 12: 575–593
DOI 10.1007/s10951-008-0093-5

An exact algorithm for single-machine scheduling without machine idle time

Shunji Tanaka · Shuji Fujikuma · Mitsuhiko Araki

Received: 31 October 2007 / Accepted: 23 September 2008 / Published online: 6 November 2008
© Springer Science+Business Media, LLC 2008

Abstract This study proposes an exact algorithm for the general single-machine scheduling problem without machine idle time to minimize the total job completion cost. Our algorithm is based on the Successive Sublimation Dynamic Programming (SSDP) method. Its major drawback is heavy memory usage to store dynamic programming states, although unnecessary states are eliminated in the course of the algorithm. To reduce both memory usage and computational efforts, several improvements to the previous algorithm based on the SSDP method are proposed. Numerical experiments show that our algorithm can optimally solve 300 jobs instances of the total weighted tardiness problem and the total weighted earliness-tardiness problem, and that it outperforms the previous algorithms specialized for these problems.

Keywords Single-machine scheduling · Exact algorithm · Time-indexed formulation · Lagrangian relaxation · Successive sublimation dynamic programming method

1 Introduction

In this study, we treat a class of scheduling problems to sequence jobs on a single machine so that the sum of job completion costs is minimized. Consider that a set of n jobs $\mathcal{N} = \{1, \ldots, n\}$ is to be processed on a single machine without preemption. Job i ($i \in \mathcal{N}$) is given an integer processing time p_i and an integer-valued cost function $f_i(t)$. Machine idle time is not permitted and the machine cannot process more than one job at a time. All the jobs can be processed from time zero. Thus, the jobs should be processed in the interval $[0, T]$, where $T = \sum_{i=1}^{n} p_i$. The objective is to find an optimal job sequence to minimize $\sum_{i=1}^{n} f_i(C_i)$, where C_i denotes the completion time of job i.

For this single-machine scheduling problem without machine idle time, an efficient lower bound computation method based on state–space relaxation was proposed by Abdul-Razaq and Potts (1988), which was originally proposed by Christofides et al. (1981) for routing problems. In this approach, the constraint on the number of job occurrences such that each job should be processed exactly once is relaxed by Lagrangian relaxation. To improve the lower bound more, additional constraints on successive jobs and on state–space modifiers were introduced. These relaxations with and without the additional constraints can be solved efficiently by dynamic programming. Abdul-Razaq and Potts (1988) also proposed a branch-and-bound algorithm where this lower bounding approach is integrated into.

Following this research, Ibaraki and Nakamura (1994) applied the Successive Sublimation Dynamic Programming (SSDP) method (Ibaraki 1987) to this problem. This dynamic programming based exact algorithm starts from a relaxation obtained by relaxing the constraint on the number of job occurrences, and next adds the constraints on successive jobs to the relaxation. Then, the constraints on state–space modifiers are successively added until the gap between the lower and upper bounds becomes zero. To suppress the increase of dynamic programming states caused by the additional constraints, unnecessary states are eliminated in the course of the algorithm by computing lower

S. Tanaka (✉) · S. Fujikuma
Graduate School of Electrical Engineering, Kyoto University, Kyotodaigaku-Katsura, Nishikyo-ku, Kyoto 615-8510, Japan
e-mail: tanaka@kuee.kyoto-u.ac.jp

M. Araki
Matsue College of Technology, 14-4 Nishiikuma-Cho, Matsue City, Shimane 690-8518, Japan

bounds for passing through states. Ibaraki and Nakamura applied this algorithm to the single-machine total weighted earliness–tardiness problem without machine idle time, and reported that their algorithm is faster than the branch-and-bound algorithm by Abdul-Razaq and Potts (1988). However, they also pointed out that their algorithm will not outperform the specialized algorithm by Potts and Van Wassenhove (1985) when it is applied to the single-machine total weighted tardiness problem, and that their algorithm is hard to apply to those instances with a longer scheduling horizon T (total processing time) due to heavy memory usage.

However, rapid progress in computer technologies has changed the situation very much from when their paper was published. Large size memory is available at a cheaper cost nowadays, and memory size is not so restrictive a limitation as it once was. If this fact is taken into account, the framework of the SSDP method is still attractive although we should admit that some modifications and improvements are necessary.

The purpose of this study is to construct an exact algorithm based on the SSDP method. To reduce both the memory usage and computational efforts, we propose several improvements for the original algorithm by Ibaraki and Nakamura (1994):

(1) Lower bound improvement by the dominance of two adjacent jobs
(2) Further improvement by the dominance of four successive jobs
(3) Sophisticated step sizing in subgradient optimization
(4) Efficient upper bound computation by the enhanced dynasearch (Congram et al. 2002; Grosso et al. 2004)
(5) Improved choice of state–space modifiers

These improvements enable us to solve even 300 jobs instances optimally.

It should be noted that our algorithm is not restricted to a specific single-machine scheduling problem, but is applicable to general scheduling problems to minimize an additive job completion cost without machine idle time. Among these, one of the most extensively studied is the total weighted tardiness problem ($1\|\sum w_i T_i$, according to the standard classification). Potts and Van Wassenhove (1985) proposed a branch-and-bound algorithm for this problem, and it was so efficient that there had been no better exact algorithms for a long time. Recently, Pan and Shi (2007) reported that their branch-and-bound algorithm optimally solved all the open OR-library benchmark instances (available from http://people.brunel.ac.uk/~mastjjb/jeb/info.html) with 100 jobs. However, it still takes maximum 9 hours for the hardest instance on a 2.8 GHz Pentium 4 computer. On the other hand, as will be shown by numerical experiments, our algorithm can solve these 100 jobs instances only within 40 seconds and can solve 300 jobs instances within 1 hour on a 2.4 GHz Pentium 4 computer.

It will also be shown that the algorithm can solve almost all the 300 jobs instances of the total weighted earliness–tardiness problem without machine idle time ($1\|\sum(\alpha_i E_i + \beta_i T_i)$). To the best of authors' knowledge, the most recent exact algorithm for this problem is the branch-and-bound algorithm proposed by Liaw (1999). However, their algorithm cannot solve some of 40 jobs instances optimally within 1 hour on a 266 MHz Pentium II computer. This fact together with the results for $1\|\sum w_i T_i$ indicates the potential of our proposed algorithm.

This paper is organized as follows. In Sect. 2, our problem is formulated as a 0–1 integer programming problem, and it is then relaxed by introducing Lagrangian multipliers to obtain a tight lower bound. The method to improve the lower bound proposed by Abdul-Razaq and Potts (1988) is also introduced in this section. Based on this improvement, Sect. 3 states the exact algorithm proposed by Ibaraki and Nakamura (1994). The main contributions of this study are given in Sect. 4, and a new algorithm is proposed. Then, in Sect. 5 its effectiveness is examined by numerical experiments. Finally, our results and future research directions are summarized in Sect. 6.

2 Time-indexed formulation and Lagrangian relaxation

In this section, we will first formulate our problem as a 0–1 integer programming problem, which is known as time-indexed formulation (Pritsker et al. 1969; Dyer and Wolsey 1990; Sousa and Wolsey 1992; van den Akker et al. 1999). Next, the Lagrangian relaxation technique is applied to compute a lower bound of the problem. Then, it is further improved by the method proposed by Abdul-Razaq and Potts (1988), which was originally proposed by Christofides et al. (1981) for routing problems.

2.1 Time-indexed formulation and Lagrangian relaxation

Let us introduce binary decision variables x_{it} ($i \in \mathcal{N}$, $1 \le t \le T$) such that

$$x_{it} = \begin{cases} 1 & \text{if } t \ge p_i \text{ and job } i \text{ is completed at } t\ (t = C_i), \\ 0 & \text{otherwise.} \end{cases} \tag{1}$$

Then, our problem (P) can be formulated as follows.

$$F^{\text{opt}} = \min_{\boldsymbol{x}} F(\boldsymbol{x}) = \min_{\boldsymbol{x}} \sum_{i=1}^{n} \sum_{t=1}^{T} f_i(t) x_{it}, \tag{2}$$

$$\text{s.t.} \quad \sum_{i=1}^{n} \sum_{s=t}^{\min(T,\, t+p_i-1)} x_{is} = 1, \quad 1 \le t \le T, \tag{3}$$

$$\sum_{t=1}^{T} x_{it} = 1, \quad i \in \mathcal{N}, \tag{4}$$

$$x_{it} \in \{0, 1\}, \quad i \in \mathcal{N}, \ 1 \le t \le T. \tag{5}$$

In this formulation, (3) represents the machine capacity constraint that the machine should process exactly one job in each time slot. The constraint on the number of job occurrences (4) claims that each job should be processed exactly once.

It is well-known that a tight lower bound is obtained by solving the LP relaxation of (P) (cf. Dyer and Wolsey 1990). However, the relaxation has $O(nT)$ decision variables and it is, in general, hard to solve. An alternatively way is to apply the Lagrangian relaxation technique to (P). There are two types of Lagrangian relaxations: the relaxation of the machine capacity constraint (3) and the relaxation of the constraint on the number of job occurrences (4). One of the advantages of the former relaxation is that the original problem can be decomposed into trivial n subproblems corresponding to the n jobs. There is an early attempt by Fisher (1973) to use this relaxation for computing lower bounds in a branch-and-bound algorithm. On the other hand, the primary advantage of the latter relaxation is that it gives an easy way to obtain a tighter lower bound than for the LP relaxation as will be explained in the following subsections. This type of relaxation (state–space relaxation) is utilized by Abdul-Razaq and Potts (1988) and by Ibaraki and Nakamura (1994), as already mentioned in Introduction. It also appears when the column generation technique is applied to the LP relaxation of (P) (van den Akker et al. 2000).

In this study, we apply the latter relaxation. The relaxation of (4) by introducing Lagrangian multipliers μ_i ($i \in \mathcal{N}$) yields the following problem (LR):

$$\begin{aligned} L(\boldsymbol{\mu}) &= \min_{\boldsymbol{x}} \left\{ \sum_{i=1}^{n} \sum_{t=1}^{T} f_i(t) x_{it} + \sum_{i=1}^{n} \mu_i \left(1 - \sum_{t=1}^{T} x_{it} \right) \right\} \\ &= \min_{\boldsymbol{x}} \left\{ \sum_{i=1}^{n} \sum_{t=1}^{T} \big(f_i(t) - \mu_i \big) x_{it} + \sum_{i=1}^{n} \mu_i \right\}, \end{aligned} \tag{6}$$

s.t. (3), (5).

For a fixed set of $\boldsymbol{\mu}$, the relaxation (LR) can be solved by dynamic programming of time complexity $O(nT)$ (Abdul-Razaq and Potts 1988). To find multipliers that give a tighter lower bound, subgradient optimization is applied to the corresponding Lagrangian dual (D):

$$L\big(\boldsymbol{\mu}^{\text{opt}}\big) = \max_{\boldsymbol{\mu}} L(\boldsymbol{\mu}). \tag{7}$$

It is a standard framework of computing a lower bound by Lagrangian relaxation.

It is easy to see that $L(\boldsymbol{\mu}^{\text{opt}})$ is identical to the optimal objective value of the LP relaxation of (P). To improve it more, Abdul-Razaq and Potts (1988) imposed two types of additional constraints on the relaxation (LR). These will be explained in the following subsections.

2.2 Lower bound improvement by a constraint on successive jobs

Let us generate a problem (P_k) ($k \ge 0$) by adding the following constraint to (P):

$$\text{Job duplication is forbidden in any } (k+1) \text{ successive jobs.} \tag{8}$$

Since this constraint is satisfied by any feasible solution of (P), it is redundant for (P_k) and feasible regions of (P) and (P_k) are identical. However, it does make a difference when the constraint (4) is relaxed. By relaxing (4), we obtain the relaxation (LR_k) given by

$$L_k(\boldsymbol{\mu}) = \min_{\boldsymbol{x}} \left\{ \sum_{i=1}^{n} \sum_{t=1}^{T} \big(f_i(t) - \mu_i \big) x_{it} + \sum_{i=1}^{n} \mu_i \right\}, \tag{9}$$

s.t. (3), (5), (8).

The feasible region of (LR_k) is restricted compared to (LR), and

$$L_k(\boldsymbol{\mu}) \ge L_0(\boldsymbol{\mu}) = L(\boldsymbol{\mu}) \tag{10}$$

holds. This problem can be solved in $O(n^k T)$ time for $k > 0$ (Abdul-Razaq and Potts 1988; Péridy et al. 2003) if we choose appropriate dynamic programming states as will be shown in Appendix A. Since (LR_{n-1}) is equivalent to the original problem (P), we can improve the lower bound to a desired extent by choosing a larger k, if we admit the increase of computational efforts. In this study, we assume $k \le 2$.

2.3 Lower bound improvement by state-space modifiers

Next, we improve $L_k(\boldsymbol{\mu})$ further by introducing state–space multipliers. This constraint is to restrict the total weighted number of job occurrences in a solution, where the weights are called "state–space modifiers." By summing up (4) after multiplying nonnegative integer state–space modifiers q_i^l ($i \in \mathcal{N}$) for every l ($1 \le l \le m$), we obtain

$$\sum_{i=1}^{n} q_i^l \sum_{t=1}^{T} x_{it} = \sum_{i=1}^{n} q_i^l = Q^l, \quad 1 \le l \le m, \tag{11}$$

where $Q^l = \sum_{i=1}^{n} q_i^l$. Thus, it restricts to Q^l for every l ($1 \le l \le m$), the sum of the job modifiers q_i^l ($i \in \mathcal{N}$) in a

solution. Clearly, the feasible region of (P_k) does not change when (11) is added to (P_k) as a constraint. Hence, as in Sect. 2.2, we consider the relaxation (P_k^m) constructed from (P_k) by adding the constraint (11). Then, the problem (LR_k^m) is derived by relaxing the constraint (4) as follows:

$$L_k^m(\boldsymbol{\mu}) = \min_{\boldsymbol{x}} \left\{ \sum_{i=1}^{n} \sum_{t=1}^{T} \big(f_i(t) - \mu_i\big) x_{it} + \sum_{i=1}^{n} \mu_i \right\}, \tag{12}$$

s.t. (3), (5), (8), (11).

Obviously,

$$L_k^m(\boldsymbol{\mu}) \geq L_k(\boldsymbol{\mu}) \tag{13}$$

holds and the lower bound is improved. This relaxation (LR_k^m) can be solved in $O(n^k T \prod_{l=1}^{m}(1+Q^l))$ time for $k>0$ and $O(nT \prod_{l=1}^{m}(1+Q^l))$ for $k=0$ (Abdul-Razaq and Potts 1988). In this study, we only consider (LR_2^m).

3 An exact algorithm proposed by Ibaraki and Nakamura

Ibaraki and Nakamura (1994) proposed an exact algorithm for our single-machine scheduling problem, which is based on the Successive Sublimation Dynamic Programming (SSDP) method (Ibaraki 1987). The SSDP method is a dynamic programming based exact algorithm that starts from a relaxation of the original problem and then successively eliminates unnecessary dynamic programming states and constructs "sublimations." When it is applied to our problem, the construction of sublimations is interpreted as the addition of the constraints (8) and (11) to the relaxation (LR). An outline of the algorithm by Ibaraki and Nakamura (1994) will be stated in Sect. 3.2 and we will explain in Sect. 3.1 the elimination of dynamic programming states in our problem via graph representations.

3.1 Graph representation and state elimination

The relaxations (LR_k) and (LR_k^m) can be converted into constrained and unconstrained shortest path problems on directed graphs. Thus, it is easy to verify that they can be solved by dynamic programming. However, the number of dynamic programming states increases exponentially as k or m increases, and the relaxation becomes intractable. To avoid this, unnecessary dynamic programming states are eliminated by computing lower bounds for passing through states and comparing them with an upper bound. In graph representations, this corresponds to the elimination of nodes or arcs. Here, we will give graph representations, dynamic programming recursions and state elimination inequalities for (LR), (LR_1), (LR_2), and (LR_2^m). Since the complete expression of the dynamic programming recursion is presented only for (LR), please refer to Appendix A for the other relaxations.

First, we consider (LR), or its equivalent (LR_0). For simplicity of notation, we introduce a dummy job $n+1$ satisfying

$$p_{n+1} = 1, \qquad \mu_{n+1} = 0, \qquad f_{n+1}(t) = 0, \quad \forall t. \tag{14}$$

Consider a directed graph $G = (V, A)$ where the node set V is given by

$$\begin{aligned} V &= \{v_{00}\} \cup V_{\mathrm{O}}, \\ V_{\mathrm{O}} &= \{v_{it} \mid i \in \mathcal{N},\ p_i \leq t \leq T\} \cup \{v_{n+1,T+1}\}, \end{aligned} \tag{15}$$

and the arc set A is defined by

$$A = A_{\mathrm{A}} \cup A_{\mathrm{B}} \cup A_{\mathrm{C}}, \tag{16}$$

$$A_{\mathrm{A}} = \big\{(v_{j,t-p_i}, v_{it}) \mid i, j \in \mathcal{N},\ p_i + p_j \leq t \leq T\big\}, \tag{17}$$

$$A_{\mathrm{B}} = \big\{(v_{00}, v_{ip_i}) \mid i \in \mathcal{N}\big\}, \tag{18}$$

$$A_{\mathrm{C}} = \big\{(v_{iT}, v_{n+1,T+1}) \mid i \in \mathcal{N}\big\}, \tag{19}$$

where the arc from v' to v is denoted by (v', v). For each arc $a = (v_{j,t-p_i}, v_{it}) \in A$, let us define the length c_a by

$$c_a = f_i(t) - \mu_i. \tag{20}$$

Then, (LR) can be converted into the shortest path problem on G from v_{00} to $v_{n+1,T+1}$, where a node v_{it} ($i \in \mathcal{N}$) visited on the path corresponds to the completion of job i at t ($x_{it} = 1$). This shortest path is obtained by forward or backward dynamic programming. In forward dynamic programming, the shortest path length $h_0(t;\ \boldsymbol{\mu})$ from v_{00} to v_{*t} is recursively computed by

$$h_0(0;\ \boldsymbol{\mu}) = 0, \tag{21}$$

$$h_0(t;\ \boldsymbol{\mu}) = \min_{v_{it} \in V_{\mathrm{O}}} \big\{h_0(t - p_i;\ \boldsymbol{\mu}) + f_i(t) - \mu_i\big\}, \tag{22}$$

and $L(\boldsymbol{\mu}) = L_0(\boldsymbol{\mu})$ is given by

$$L(\boldsymbol{\mu}) = L_0(\boldsymbol{\mu}) = h_0(T+1; \boldsymbol{\mu}) + \sum_{i=1}^{n} \mu_i. \tag{23}$$

Therefore, the time complexity of (LR) is $O(nT)$, as already stated in the preceding section, because the minimization in the right-hand side of (22) can be computed in $O(n)$ time for every t.

When backward dynamic programming is applied, the shortest path length $H_0(t;\ \boldsymbol{\mu})$ from v_{*t} to $v_{n+1,T+1}$ is recursively computed by

$$H_0(T+1;\ \boldsymbol{\mu}) = 0, \tag{24}$$

$$H_0(t;\ \boldsymbol{\mu}) = \min_{v_{i,t+p_i} \in V_O} \big\{ H_0(t+p_i;\ \boldsymbol{\mu}) + f_i(t+p_i) - \mu_i \big\}, \tag{25}$$

and

$$L(\boldsymbol{\mu}) = L_0(\boldsymbol{\mu}) = H_0(0;\boldsymbol{\mu}) + \sum_{i=1}^{n} \mu_i. \tag{26}$$

From $h_0(t;\ \boldsymbol{\mu})$ and $H_0(t;\ \boldsymbol{\mu})$, it can be seen that the lengths of the shortest paths that pass through the nodes v_{*t} are not shorter than

$$h_0(t;\ \boldsymbol{\mu}) + H_0(t;\ \boldsymbol{\mu}). \tag{27}$$

Therefore, if there exists an upper bound UB of the original problem (P) satisfying

$$\text{UB} < h_0(t;\ \boldsymbol{\mu}) + H_0(t;\ \boldsymbol{\mu}) + \sum_{i=1}^{n} \mu_i, \tag{28}$$

then no jobs are completed at t in any optimal solution of (P). In this case, we can remove the nodes v_{it} $(i \in \mathcal{N})$ from V and the arcs connected to the nodes from A, when the shortest path problem is solved. It follows that the dynamic programming state $h_0(t;\ \boldsymbol{\mu})$, or $H_0(t;\ \boldsymbol{\mu})$, can be eliminated and, as a consequence, not only computational efforts but also memory usage reduces.

Similar arguments hold for (LR$_k$) $(k > 0)$. In the case of (LR$_1$), the shortest path from v_{00} to $v_{n+1,T+1}$ satisfying (8) is to be computed. It corresponds to the shortest path from v_{00} to $v_{n+1,T+1}$ on $G_S = (V, A_S)$, where

$$A_S = A_D \cup A_B \cup A_C, \tag{29}$$

$$A_D = A_A \backslash \big\{(v_{i,t-p_i}, v_{it}) \,|\, i \in \mathcal{N},\ 2p_i \le t \le T\big\}. \tag{30}$$

Let us define by $h_1(v_{it};\ \boldsymbol{\mu})$ the shortest path length from v_{00} to v_{it} on G_S, and by $H_1(v_{it};\ \boldsymbol{\mu})$ that from v_{it} to $v_{n+1,T+1}$ on G_S. To obtain the shortest path from v_{00} to $v_{n+1,T+1}$ on G_S, we recursively compute $h_1(v_{it};\ \boldsymbol{\mu})$ in forward dynamic programming and $H_1(v_{it};\ \boldsymbol{\mu})$ in backward dynamic programming. As shown in Appendix A, its time complexity is given by $O(nT)$. To eliminate unnecessary states, the inequality

$$\text{UB} < h_1(v_{it};\ \boldsymbol{\mu}) + H_1(v_{it};\ \boldsymbol{\mu}) + \sum_{i=1}^{n} \mu_i \tag{31}$$

is checked. If (31) is satisfied, job i is never completed at t in any optimal solution of (P), and the node v_{it} and the arcs connected to this node can be eliminated from G_S.

In the case of (LR$_2$), the problem corresponds to the shortest path problem from v_{00} to $v_{n+1,T+1}$ on G_S under the following constraint:

For any $i \in \mathcal{N}$, nodes corresponding to job i, i.e., v_{i*}, should not be visited twice in any three successive nodes on the path. (32)

To solve this problem, the shortest path length from v_{00} to v_{it} on G_S that satisfies the constraint (32) and that passes through $(v_{j,t-p_j}, v_{it})$ is defined by $h_2((v_{j,t-p_j}, v_{it});\ \boldsymbol{\mu})$ for forward dynamic programming. For backward dynamic programming, the shortest path length from v_{it} to $v_{n+1,T+1}$ on G_S that satisfies the constraint (32) and that passes through $(v_{it}, v_{j,t+p_j})$ is defined by $H_2((v_{it}, v_{j,t+p_j});\ \boldsymbol{\mu})$. Then, the problem can be solved in $O(n^2T)$ time (see Appendix A). To eliminate unnecessary states, the inequality

$$\text{UB} < h_2\big((v_{j,t-p_i}, v_{it});\ \boldsymbol{\mu}\big) + H_2\big((v_{j,t-p_i}, v_{it});\ \boldsymbol{\mu}\big) - (f_{it} - \mu_i) + \sum_{i=1}^{n} \mu_i \tag{33}$$

is checked, and the arc $(v_{j,t-p_i}, v_{it})$ is eliminated from G_S if (33) is satisfied.

Finally, for (LR$_2^m$), we should construct a new graph $G_S^m = (V^m, A_S^m)$ from G_S by setting

$$V^m = \big\{v_{00}^{\boldsymbol{0}}\big\} \cup V_O^m, \tag{34}$$

$$V_O^m = \big\{v_{it}^{\boldsymbol{b}} \,|\, v_{it} \in V_O \backslash \{v_{n+1,T+1}\},\ q_i^k \le b^k \le Q^k,\ 1 \le k \le m\big\} \cup \big\{v_{n+1,T+1}^{\boldsymbol{Q}}\big\}, \tag{35}$$

$$A_S^m = A_D^m \cup A_B^m \cup A_C^m, \tag{36}$$

$$A_D^m = \big\{\big(v_{j,t-p_i}^{\boldsymbol{b}-\boldsymbol{q_i}},\ v_{it}^{\boldsymbol{b}}\big) \,|\, (v_{j,t-p_i}, v_{it}) \in A_D,\ q_i^k + q_j^k \le b^k \le Q^k,\ 1 \le k \le m\big\}, \tag{37}$$

$$A_B^m = \big\{\big(v_{00}^{\boldsymbol{0}}, v_{p_i t}^{\boldsymbol{q_i}}\big) \,|\, (v_{00}, v_{p_i t}) \in A_B\big\}, \tag{38}$$

$$A_C^m = \big\{\big(v_{iT}^{\boldsymbol{Q}}, v_{n+1,T+1}^{\boldsymbol{Q}}\big) \,|\, (v_{iT}, v_{n+1,T+1}) \in A_C\big\}, \tag{39}$$

where $\boldsymbol{b} = (b^1, \ldots, b^m)$, $\boldsymbol{q_i} = (q_i^1, \ldots, q_i^m)$, $\boldsymbol{Q} = (Q^1, \ldots, Q^m)$, and $\boldsymbol{q}_{n+1} = (q_{n+1}^1, \ldots, q_{n+1}^m) = \boldsymbol{0}$. The length c_a^m of an arc $a = (v_{j,t-p_i}^{\boldsymbol{b}-\boldsymbol{q_i}}, v_{it}^{\boldsymbol{b}}) \in A_S^m$ is given by

$$c_a^m = f_i(t) - \mu_i. \tag{40}$$

Then, (LR$_2^m$) is equivalent to the shortest path problem from $v_{00}^{\boldsymbol{0}}$ to $v_{n+1,T+1}^{\boldsymbol{Q}}$ on G_S^m under the following constraint:

For any $i \in \mathcal{N}$, nodes corresponding to job i, i.e., v_{i*}^{*}, should not be visited twice in any three successive nodes on the path. (41)

This problem can be solved by dynamic programming in $O(n^2T\prod_{l=1}^{m}(1+Q^l))$ time. For this dynamic programming, $h_2^m((v_{j,t-p_i}^{\boldsymbol{b}-\boldsymbol{q_i}}, v_{it}^{\boldsymbol{b}});\ \boldsymbol{\mu})$ and $H_2^m((v_{it}^{\boldsymbol{b}}, v_{j,t+p_j}^{\boldsymbol{b}+\boldsymbol{q_j}});\ \boldsymbol{\mu})$

on G_S^m are defined similarly to $h_2((v_{j,t-p_j}, v_{it});\ \boldsymbol{\mu})$ and $H_2((v_{it}, v_{j,t+p_j});\ \boldsymbol{\mu})$ on G_S, respectively. The condition for state elimination is also similar to (33), and if

$$\mathrm{UB} < h_2^m\big((v_{j,t-p_i}^{\boldsymbol{b}-\boldsymbol{q}_i}, v_{it}^{\boldsymbol{b}});\ \boldsymbol{\mu}\big) + H_2^m\big((v_{j,t-p_i}^{\boldsymbol{b}-\boldsymbol{q}_i}, v_{it}^{\boldsymbol{b}});\ \boldsymbol{\mu}\big) - \big(f_i(t) - \mu_i\big) + \sum_{i=1}^{n} \mu_i \tag{42}$$

is satisfied, the arc $(v_{j,t-p_i}^{\boldsymbol{b}-\boldsymbol{q}_i}, v_{it}^{\boldsymbol{b}})$ can be eliminated from G_S^m.

3.2 An outline of the algorithm

The exact algorithm by Ibaraki and Nakamura (1994) consists of four stages. In short, it searches for a better set of Lagrangian multipliers by subgradient optimization: first for (LR) in Stage 1 and then for (LR_1) in Stage 2. Later, they are fixed and (LR_2) is solved in Stage 3. After that, (LR_2^m) is solved in Stage 4 by adding state–space modifiers (increasing m) until the gap between the lower and upper bounds vanishes. State elimination is performed in every stage to reduce the increase of dynamic programming states caused by the addition of the constraints. The detailed algorithm is given as follows.

Stage 1 Construct the following two schedules, and use the better as the initial upper bound UB:

(a) A schedule sequenced greedily in the forward direction
(b) An EDD (Earliest DueDate order) schedule

Apply subgradient optimization to the Lagrangian dual corresponding to (LR) for a better set of Lagrangian multipliers, where (LR) is solved by forward dynamic programming. An upper bound is computed by the method explained in Sect. 3.4 in every five iterations of the subgradient optimization, and UB is updated if it is dominated by the new upper bound. If there is no gap between $L(\boldsymbol{\mu})$ and UB, halt. Otherwise, backward dynamic programming is applied for the best multipliers $\boldsymbol{\mu}^{\mathrm{stage1}}$ obtained by the subgradient optimization, and state elimination is performed. That is, the nodes v_{it} for all $i \in \mathcal{N}$ are eliminated from $G = (V, A)$ if

$$\mathrm{UB} \le h_0\big(t;\ \boldsymbol{\mu}^{\mathrm{stage1}}\big) + H_0\big(t;\ \boldsymbol{\mu}^{\mathrm{stage1}}\big) + \sum_{i=1}^{n} \mu_i^{\mathrm{stage1}} \tag{43}$$

is satisfied.

Stage 2 Starting from the multipliers $\boldsymbol{\mu}^{\mathrm{stage1}}$, apply subgradient optimization to the Lagrangian dual corresponding to (LR_1) for a smaller number of iterations. When (LR_1) is solved, the dynamic programming states eliminated in Stage 1 are not considered. In other words, the graph $G_S = (V, A_S)$ is constructed by (30) from the reduced graph $G = (V, A)$ obtained after the state elimination in Stage 1. After the subgradient optimization is terminated, an upper bound is computed by using the solution of (LR_1) for the best multipliers $\boldsymbol{\mu}^{\mathrm{stage2}}$, and UB is updated if it is dominated. Halt if no gap exists. Otherwise, state elimination is performed: The node v_{it} is eliminated from of G_S if

$$\mathrm{UB} \le h_1\big(v_{it};\ \boldsymbol{\mu}^{\mathrm{stage2}}\big) + H_1\big(v_{it};\ \boldsymbol{\mu}^{\mathrm{stage2}}\big) + \sum_{i=1}^{n} \mu_i^{\mathrm{stage2}} \tag{44}$$

is satisfied.

Stage 3 Solve (LR_2) for the multipliers $\boldsymbol{\mu}^{\mathrm{stage2}}$ on the reduced graph G_S after the state elimination in Stage 2. An upper bound is also computed, and UB is updated if it is dominated. Halt if no gap exists. Otherwise, the arc $(v_{j,t-p_i}, v_{it})$ satisfying

$$\mathrm{UB} \le h_2\big((v_{j,t-p_i}, v_{it});\ \boldsymbol{\mu}^{\mathrm{stage2}}\big) + H_2\big((v_{j,t-p_i}, v_{it});\ \boldsymbol{\mu}^{\mathrm{stage2}}\big) - \big(f_i(t) - \mu_i\big) + \sum_{i=1}^{n} \mu_i^{\mathrm{stage2}} \tag{45}$$

is eliminated from G_S.

Stage 4 The following procedure is applied until the gap between $L_2^m(\boldsymbol{\mu}^{\mathrm{stage2}})$ and UB becomes zero.

0° $m := 0$ and $G_S^m = G_S$.

1° $m := m + 1$. State–space modifiers q_i^m are determined and G_S^m is constructed from G_S^{m-1} according to (34)–(40). Then, (LR_2^m) for the multipliers $\boldsymbol{\mu}^{\mathrm{stage2}}$ is solved by *forward* dynamic programming. An upper bound is also computed, and UB is updated if it is dominated. Then, the arc $(v_{j,t-p_i}^{\boldsymbol{b}-\boldsymbol{q}_i}, v_{it}^{\boldsymbol{b}})$ satisfying

$$\mathrm{UB} \le h_2^m\big((v_{j,t-p_i}^{\boldsymbol{b}-\boldsymbol{q}_i}, v_{it}^{\boldsymbol{b}});\ \boldsymbol{\mu}^{\mathrm{stage2}}\big) + H_2^{m-1}\big((v_{j,t-p_i}^{\boldsymbol{b}'-\boldsymbol{q}'_i}, v_{it}^{\boldsymbol{b}'});\ \boldsymbol{\mu}^{\mathrm{stage2}}\big) - \big(f_i(t) - \mu_i\big) + \sum_{i=1}^{n} \mu_i^{\mathrm{stage2}} \tag{46}$$

is eliminated from G_S^m, where $\boldsymbol{b}' = (b^1, \ldots, b^{m-1})$ and $\boldsymbol{q}'_i = (q_i^1, \ldots, q_i^{m-1})$.

2° $m := m + 1$. State-space modifiers q_i^m ($i \in \mathcal{N}$) are determined and (LR_2^m) for the multipliers $\boldsymbol{\mu}^{\mathrm{stage2}}$ is solved by *backward* dynamic programming. An upper bound is also computed, and UB is updated if it is dominated.

Then, the arc $(v_{j,t-p_i}^{\boldsymbol{b}-\boldsymbol{q}_j}, v_{it}^{\boldsymbol{b}})$ is eliminated from G_S^m if

$$\begin{aligned}\mathrm{UB} \leq\ & h_2^{m-1}\big((v_{j,t-p_i}^{\boldsymbol{b}'-\boldsymbol{q}_i'}, v_{it}^{\boldsymbol{b}'});\ \boldsymbol{\mu}^{\text{stage2}}\big) \\ &+ H_2^m\big((v_{j,t-p_i}^{\boldsymbol{b}-\boldsymbol{q}_i}, v_{it}^{\boldsymbol{b}});\ \boldsymbol{\mu}^{\text{stage2}}\big) \\ &- \big(f_i(t)-\mu_i\big) + \sum_{i=1}^n \mu_i^{\text{stage2}} \end{aligned} \tag{47}$$

is satisfied.

3° Go to 1°.

The conditions (43)–(47) of the algorithm imply that state elimination is performed even when the upper and lower bounds are identical. Therefore, it is possible that all feasible paths in the corresponding graph are eliminated and that dynamic programming becomes infeasible. However, it does not cause any problems because, in this case, the current upper bound is not dominated by any solutions, and thus its optimality is ensured.

In addition, the conditions for state elimination in Stage 4, i.e., (46) and (47), differ from (42). It is because state elimination for some m is performed by using the result for $m-1$ at the previous iteration instead of applying both backward and forward dynamic programming for this m.

How to apply subgradient optimization in Stages 1 and 2, how to compute upper bounds in every stage, and how to choose state-space modifiers in Stage 4 will be explained in the following subsections.

3.3 Subgradient optimization

A better set of Lagrangian multipliers is searched for in both Stages 1 and 2 by applying subgradient optimization to the corresponding Lagrangian duals. In the subgradient optimization, the multipliers are updated at the rth iteration by

$$\begin{aligned}\mu_i^{(r+1)} :=\ & \mu_i^{(r)} + \frac{\widetilde{\mathrm{UB}}^{(r)} - \mathrm{LB}^{(r)}}{\sum_{j=1}^n (1-\sum_{t=1}^T x_{jt}^{(r)})^2} \\ & \times \left(1-\sum_{t=1}^T x_{it}^{(r)}\right), \quad i\in\mathcal{N},\end{aligned} \tag{48}$$

where $\boldsymbol{\mu}^{(1)}=\mathbf{0}$, and $x_{it}^{(r)}$ $(i\in\mathcal{N}, 1\le t\le T)$ and $\mathrm{LB}^{(r)}$ denote the optimal solution and the optimal objective value of the corresponding relaxation for $\boldsymbol{\mu}^{(r)}$, respectively ($\mathrm{LB}^{(r)} = L(\boldsymbol{\mu}^{(r)})$ or $\mathrm{LB}^{(r)} = L_1(\boldsymbol{\mu}^{(r)})$). In (48), $\widetilde{\mathrm{UB}}^{(1)}$ is initialized by an upper bound and $\widetilde{\mathrm{UB}}^{(r)}$ is updated by the following rule.

(a) If $\mathrm{LB}^{(r)} \ge \widetilde{\mathrm{UB}}^{(r)}$,

$$\widetilde{\mathrm{UB}}^{(r+1)} := \mathrm{LB}^{(r)} + \big|\widetilde{\mathrm{UB}}^{(r)} - \widetilde{\mathrm{UB}}^{(r-1)}\big|/2. \tag{49}$$

(b) If the best lower bound is not updated during the last K_1 iterations,

$$\widetilde{\mathrm{UB}}^{(r+1)} := \big|\widetilde{\mathrm{UB}}^{(r)} + \overline{\mathrm{LB}}^{(r)}\big|/2, \tag{50}$$

where

$$\overline{\mathrm{LB}}^{(r)} = \max_{1\le k\le r} \mathrm{LB}^{(r)}. \tag{51}$$

(c) Otherwise,

$$\widetilde{\mathrm{UB}}^{(r+1)} = \widetilde{\mathrm{UB}}^{(r)}. \tag{52}$$

The iterations are terminated when $\widetilde{\mathrm{UB}}^{(r+1)}$ is updated K_2 times by (a) or (b). The parameters (K_1, K_2) are chosen as $(7, 5)$ in Stage 1 and $(7, 3)$ in Stage 2.

3.4 Upper bound computation

An upper bound is computed as follows. Assume that a dynamic programming solution of the relaxation (LR), (LR_1), (LR_2), or (LR_2^m) is already obtained. From this solution, we first construct a partial job sequence by removing jobs other than those occurring exactly once or only successively. For example, if the solution is 3, 5, 5, 1, 1, 7, 4, 5, the partial job sequence 3, 1, 7, 4 is obtained. Next, we solve the problem to find a job sequence minimizing the total cost without changing job precedence relations in the partial sequence. It can be done by dynamic programming of time complexity $O(N_2(N_1+1)2^{N_2})$, where N_1 is the length of the partial sequence and $N_2 = n - N_1$. This dynamic programming is time and space consuming and thus is applied only when $N_2 \le 9$.

In Stage 1 of the SSDP algorithm, partial job sequences are generated from both the current dynamic programming solution of (LR) and the solution constructed by always choosing the second best in the minimization of (22). Then, the above dynamic programming is applied to them.

3.5 The choice of state–space modifiers

The state–space modifiers are determined in Stage 4 by the following procedure. Two jobs, jobs i_1 and i_2, are selected from those that do not occur in the current dynamic programming solution of (LR_2^{m-1}). If there is only one such job, job i_2 is selected from those that occur more than once. Then, the modifiers q_i^m are chosen as

$$q_i^m = \begin{cases} 1 & \text{if } i = i_1, \\ 2 & \text{if } i = i_2, \\ 0 & \text{otherwise,} \end{cases} \tag{53}$$

and Q^m as $Q^m = 3$. In the case that states at the current iteration occupy more than 95% of the maximum available

memory, only one job is selected and its modifier is set to be one ($q_i^m = 1$ and $Q^m = 1$).

By choosing state–space modifiers as in (53), it is ensured that at least jobs i_1 should occur in the solution of (LR_2^m). Since the job chosen as "job i_1" differs for each m, Stage 3 terminates in a finite number of iterations.

4 Proposed algorithm

The algorithm stated in the preceding section was successfully applied to 35 jobs instances of the total weighted earliness–tardiness problem without machine idle time (Ibaraki and Nakamura 1994). However, it is hard to apply their algorithm to larger instances because of its heavy memory usage. Therefore, it is inevitable to reduce both the memory usage and computational efforts, although computers have become much powerful since the paper was published. To achieve this, we propose the following improvements:

(1) Lower bound improvement by the dominance of two adjacent jobs
(2) Further improvement by the dominance of four successive jobs
(3) Sophisticated step sizing in subgradient optimization
(4) Efficient upper bound computation by the enhanced dynasearch (Congram et al. 2002; Grosso et al. 2004)
(5) Improved choice of state–space modifiers

These improvements enable us to reduce memory usage, i.e., dynamic programming states. Thus, they also lead to the reduction of computational efforts because the reduction of states improves the efficiency of dynamic programming for the relaxations. Among these, our main contributions are (1), (2), and (5). In (1), an additional constraint is imposed on the relaxations, which enables us to reduce dynamic programming states and to improve the lower bound, at the same time without increasing the computational complexity. (2) also reduces dynamic programming states and improves the lower bound a little more, at the expense of additional computational efforts. (5) is to suppress the increase of dynamic programming states caused by the addition of state–space modifiers. The other two, (3) and (4), are also necessary for tight lower and upper bounds, and they improve the effectiveness of state elimination because it is performed by lower and upper bounds.

In the following subsections, we will explain these improvements step by step.

4.1 Lower bound improvement by the dominance of two adjacent jobs

First, we propose a simple but powerful method to improve the lower bound and to reduce dynamic programming states. The key notion for this improvement is *the dominance theorem of dynamic programming* (Potts and Van Wassenhove 1985). It compares two partial sequences consisting of identical subsets of jobs, and the one having the larger cost can be eliminated. If the costs are identical, either can be eliminated. Based on this theorem, Potts and Van Wassenhove (1985) proposed to improve the efficiency of their branch-and-bound algorithm. In their algorithm, the current node in the search tree is eliminated if the total cost decreases when the two jobs added most recently to the partial schedule are interchanged. Abdul-Razaq and Potts (1988) also utilized this method in their branch-and-bound algorithm.

In this study, we apply this theorem to improve the lower bound and to reduce memory usage. As we have already seen in Sect. 2, we can improve the lower bound by adding some constraint to a relaxation. Thus, we introduce a constraint for (LR_2) and (LR_2^m) that takes into account this theorem for two adjacent jobs.

Let $j \to i(t)$ denote such a situation that job i is completed at t ($C_i = t$) and job j is completed just before i ($C_j = t - p_i$). Define by $\Delta_{j\to i}(t)$ the cost sum of the two jobs i and j when $j \to i(t)$. More specifically,

$$\Delta_{j\to i}(t) = f_j(t - p_i) + f_i(t). \tag{54}$$

From the dominance theorem of dynamic programming, it is easily checked that we need not consider such schedules where $j \to i(t)$, if

$$\Delta_{j\to i}(t) > \Delta_{i\to j}(t) \tag{55}$$

is satisfied. On the other hand, if

$$\Delta_{j\to i}(t) < \Delta_{i\to j}(t) \tag{56}$$

is satisfied, we need not consider such schedules where $i \to j(t)$. This fact motivates us to introduce a constraint that restricts the processing order of adjacent jobs by checking whether $\Delta_{i\to j}(t)$ is larger than $\Delta_{j\to i}(t)$ or not.

It should be noted that adding such a constraint to (P) eliminates even feasible schedules and restricts the feasible region of (P), unlike the constraint (8) or (11). In the case that $\Delta_{i\to j}(t)$ is strictly less than or strictly greater than $\Delta_{j\to i}(t)$, it does not cause any problems because eliminated schedules are strictly dominated by some other schedules, and hence they are never optimal. However, it is not true when $\Delta_{i\to j}(t) = \Delta_{j\to i}(t)$, and optimal schedules may be eliminated. Therefore, the tie-breaking rule, i.e., which processing order is to be forbidden when $\Delta_{i\to j}(t) = \Delta_{j\to i}(t)$, should be determined carefully. The following proposition claims that at least one optimal schedule is not eliminated under a mild assumption that the tie-breaking rule is independent of t.

Proposition 4.1 *There exists at least one optimal schedule such that any two adjacent jobs j and i completed at t ($j \to i(t)$) satisfy*

$$\Delta_{j\to i}(t) < \Delta_{i\to j}(t), \tag{57}$$

or

$$\Delta_{j\to i}(t) = \Delta_{i\to j}(t), \quad j = R_{ij}, \tag{58}$$

where R_{ij} ($= R_{ji}$) defines a tie-breaking relation between jobs i and j. More specifically, R_{ij} determines which of jobs i and j should precede when these jobs are in an adjacent position and when interchanging them does not change the cost sum.

Proof See Appendix B. □

Remark 4.1 Péridy et al. (2003) also applied dominance properties for their problem (minimization of the weighted number of late jobs with release dates) to reduce possible states in dynamic programming for a similar type of relaxation. However, their method uses dominance properties of the targeted problem class, which, in general, requires problem specific analyses. On the other hand, our framework is more general and does not depend on such a priori information. This point is the primary advantage of our method because it enables us to apply the method to a wider class of problems.

Remark 4.2 Sourd (2006) independently proposed the same lower bound improvement for their problem, the single-machine total weighted earliness–tardiness problem with machine idle time (we also proposed this technique in the same year (Tanaka et al. 2006)). In his method, “reinforced Lagrangean relaxation,” the constraint on the dominance of two adjacent jobs is added to a relaxation corresponding to (LR_1) and it is solved by dynamic programming in $O(n^2T)$ time. It is quite interesting that in his recent study (Sourd 2008) elimination of dynamic programming states is performed as in Sect. 3.1, which is also parallel to this study. As a matter of course, there are several differences between his approach and ours: He imposed the dominance of two adjacent jobs not on (LR_2) but on (LR_1), he did not consider the difficulty arising from the tie-breaking rule, the state elimination was not motivated by the SSDP method, and so on. Among these, the primary difference is that he applied the method for computing lower bounds in a branch-and-bound algorithm, whereas our algorithm is based fully on dynamic programming. We also tried a branch-and-bound algorithm at first (Tanaka et al. 2006), but the SSDP method turned out to be far more efficient.

Now, we define by $\mathcal{P}_i(t)$ ($i \in \mathcal{N}$, $p_i + 1 \le t \le T$) the set of jobs j ($j \ne i$) satisfying $p_i + p_j \le t$ and either (57) or (58). If no job satisfies the conditions, $\mathcal{P}_i(t)$ becomes empty, i.e., $\mathcal{P}_i(t) = \phi$. By using $\mathcal{P}_i(t)$, the following constraint is introduced:

$$\text{Job } i \text{ completed at } t \text{ should be adjacently preceded by job } j \ (j \in \mathcal{P}_i(t)). \tag{59}$$

This constraint is added to (P_2) and (P_2^m), and then the constraint (4) is relaxed. The corresponding relaxations are denoted by $(\widehat{\mathrm{LR}}_2)$ and $(\widehat{\mathrm{LR}}_2^m)$, respectively.

It is easy to see that $(\widehat{\mathrm{LR}}_2)$ and $(\widehat{\mathrm{LR}}_2^m)$ can be solved by dynamic programming of the time complexities $O(n^2T)$ and $O(n^2T\prod_{l=1}^m(1+Q^l))$, respectively. Therefore, the lower bound can be improved by the additional constraint (59) without increasing the time complexity from (LR_2) or from (LR_2^m). In practice, both the computational efforts and the memory usage reduce. It can be explained by using the graph representation in Sect. 3.1.

To solve $(\widehat{\mathrm{LR}}_2)$, we are to consider the constrained shortest path problem on $\widehat{G}_\mathrm{S} = (V, \widehat{A}_\mathrm{S})$ instead of G_S, where $\widehat{A}_\mathrm{S}$ is defined by

$$\widehat{A}_\mathrm{S} = \widehat{A}_\mathrm{D} \cup A_\mathrm{B} \cup A_\mathrm{C}, \tag{60}$$

$$\widehat{A}_\mathrm{D} = \left\{(v_{j,t-p_i}, v_{it}) \mid (v_{j,t-p_i}, v_{it}) \in A_\mathrm{D},\ j \in \mathcal{P}_i(t)\right\}. \tag{61}$$

More specifically, $\widehat{A}_\mathrm{S}$ is defined by removing from A_S those arcs that do not satisfy the constraint (59). Therefore, the optimal objective value $\widehat{L}_2(\boldsymbol{\mu})$ of $(\widehat{\mathrm{LR}}_2)$ can be computed in a similar way as $L_2(\boldsymbol{\mu})$ by the dynamic programming recursions (A.15)–(A.26) (we define $\widehat{y}_2(v_{it};\ \boldsymbol{\mu}), \widehat{\lambda}_2(v_{it};\ \boldsymbol{\mu}), \ldots,$ on $\widehat{G}_\mathrm{S}$ as $y_2(v_{it};\ \boldsymbol{\mu}), \lambda_2(v_{it};\ \boldsymbol{\mu}), \ldots,$ on G_S, respectively). Moreover, the optimal objective value $\widehat{L}_2^m(\boldsymbol{\mu})$ of $(\widehat{\mathrm{LR}}_2^m)$ can also be computed by defining $\widehat{G}_\mathrm{S}^m$ from $\widehat{G}_\mathrm{S}$ as G_S^m in (34)–(39) and by solving the constrained shortest path problem on $\widehat{G}_\mathrm{S}^m$. Since the number of arcs reduces in both problems, not only the memory usage for storing states but also the efforts to compute the minimization in the dynamic programming recursions (corresponding to (A.16)–(A.18) and (A.22)–(A.24)) reduce.

As it will be summarized in Sect. 4.6, our algorithm consists of three stages. We first apply subgradient optimization to the Lagrangian dual corresponding to (LR_1) in Stage 1 and next to the dual corresponding to $(\widehat{\mathrm{LR}}_2)$ in Stage 2. Then, $(\widehat{\mathrm{LR}}_2^m)$ is solved in Stage 3 by increasing m. Unlike the original algorithm in Sect. 3.2, (LR) is not solved in our algorithm because the time complexities of (LR) and (LR_1) are the same $O(nT)$, and hence it seems unnecessary to solve (LR). One of the reasons why (LR) was solved in the original algorithm would be that less efficient dynamic programming was applied: The time complexities for (LR_1), (LR_2), and (LR_2^m) were $O(n^2T)$, $O(n^3T)$, and $O(n^3T\prod_{l=1}^m(1+Q^l))$, respectively. In this case, it is natural to solve all these problems if we take into account a trade-off between the improvement of the lower bound and the increase of computational efforts (or memory usage).

4.2 Lower bound improvement by the dominance of four successive jobs

As we have already seen in the preceding subsection, the dominance theorem of dynamic programming plays an important role in improving the lower bound and reducing dynamic programming states. Therefore, we further introduce the dominance of four successive jobs for $(\widehat{\mathrm{LR}}_2^m)$.

Let us consider an arc $(v_{j,t-p_i}^{\boldsymbol{b}-\boldsymbol{q}_i}, v_{it}^{\boldsymbol{b}})$ in the graph representation $\widehat{G}_{\mathrm{S}}^m = (V^m, \widehat{A}_{\mathrm{S}}^m)$ of $(\widehat{\mathrm{LR}}_2^m)$. In the forward dynamic programming for $(\widehat{\mathrm{LR}}_2^m)$, the constrained shortest path from $v_{00}^{\boldsymbol{0}}$ to $v_{it}^{\boldsymbol{b}}$ that passes through the arc is computed among all such paths. To apply the dominance theorem of four successive jobs, we concentrate on the last four nodes of the paths. More specifically, we consider a set of partial paths $\mathcal{E}((v_{j,t-p_i}^{\boldsymbol{b}-\boldsymbol{q}_i}, v_{it}^{\boldsymbol{b}}))$ defined by

$$\begin{aligned}\mathcal{E}\big((v_{j,t-p_i}^{\boldsymbol{b}-\boldsymbol{q}_i}, v_{it}^{\boldsymbol{b}})\big) &= \big\{\big(v_{l,t-p_i-p_j-p_k}^{\boldsymbol{b}-\boldsymbol{q}_i-\boldsymbol{q}_j-\boldsymbol{q}_k}, v_{k,t-p_i-p_j}^{\boldsymbol{b}-\boldsymbol{q}_i-\boldsymbol{q}_j}, v_{j,t-p_i}^{\boldsymbol{b}-\boldsymbol{q}_i}, v_{it}^{\boldsymbol{b}}\big) \\ &\qquad \big| \, l\neq j,\ l\neq i,\ k\neq i, \big(v_{l,t-p_i-p_j-p_k}^{\boldsymbol{b}-\boldsymbol{q}_i-\boldsymbol{q}_j-\boldsymbol{q}_k}, v_{k,t-p_i-p_j}^{\boldsymbol{b}-\boldsymbol{q}_i-\boldsymbol{q}_j}\big), \\ &\qquad \big(v_{k,t-p_i-p_j}^{\boldsymbol{b}-\boldsymbol{q}_i-\boldsymbol{q}_j}, v_{j,t-p_i}^{\boldsymbol{b}-\boldsymbol{q}_i}\big) \in \widehat{A}_{\mathrm{S}}^m\big\}. \end{aligned} \tag{62}$$

Now, assume that job i is completed at t and job j is completed at $t-p_i$ in an optimal schedule for the original problem (P). Then, it should include one of the partial schedules corresponding to the partial paths in $\mathcal{E}((v_{j,t-p_i}^{\boldsymbol{b}-\boldsymbol{q}_i}, v_{it}^{\boldsymbol{b}}))$. If this partial path is denoted by $(v_{l_{\mathrm{O}},t-p_i-p_j-p_{k_{\mathrm{O}}}}^{\boldsymbol{b}-\boldsymbol{q}_i-\boldsymbol{q}_j-\boldsymbol{q}_{k_{\mathrm{O}}}}, v_{k_{\mathrm{O}},t-p_i-p_j}^{\boldsymbol{b}-\boldsymbol{q}_i-\boldsymbol{q}_j}, v_{j,t-p_i}^{\boldsymbol{b}-\boldsymbol{q}_i}, v_{it}^{\boldsymbol{b}})$, the included partial schedule consists of jobs l_{O}, k_{O}, j, and i, and they are completed at t in this order. From the optimality of the schedule, this partial schedule should be optimal and never dominated by the other partial schedules of jobs l_{O}, k_{O}, j, and i completed at t. In other words, for the schedule to be optimal, it is necessary that the included partial schedule is optimal. Therefore, if all the partial schedules corresponding to the partial paths in $\mathcal{E}((v_{j,t-p_i}^{\boldsymbol{b}-\boldsymbol{q}_i}, v_{it}^{\boldsymbol{b}}))$ are not optimal, any schedule where job i is completed at t and job j is completed at $t-p_i$ cannot be optimal. In this case, the arc $(v_{j,t-p_i}^{\boldsymbol{b}-\boldsymbol{q}_i}, v_{it}^{\boldsymbol{b}})$ can be eliminated.

Whether the partial schedule corresponding to a partial path $(v_{l,t-p_i-p_j-p_k}^{\boldsymbol{b}-\boldsymbol{q}_i-\boldsymbol{q}_j-\boldsymbol{q}_k}, v_{k,t-p_i-p_j}^{\boldsymbol{b}-\boldsymbol{q}_i-\boldsymbol{q}_j}, v_{j,t-p_i}^{\boldsymbol{b}-\boldsymbol{q}_i}, v_{it}^{\boldsymbol{b}}) \in \mathcal{E}((v_{j,t-p_i}^{\boldsymbol{b}-\boldsymbol{q}_i}, v_{it}^{\boldsymbol{b}}))$ is dominated or not can be checked by enumerating all the permutations of jobs l, k, j, and i completed at t. However, this dominance check should be performed for all the partial paths in $\mathcal{E}((v_{j,t-p_i}^{\boldsymbol{b}-\boldsymbol{q}_i}, v_{it}^{\boldsymbol{b}}))$ to eliminate the arc $(v_{j,t-p_i}^{\boldsymbol{b}-\boldsymbol{q}_i}, v_{it}^{\boldsymbol{b}})$. Thus, it requires $O(n^2)$ time for one arc and causes additional computational efforts, unlike the improvement by the dominance of two adjacent jobs in the preceding subsection. Therefore, this improvement is applied only to $(\widehat{\mathrm{LR}}_2^m)$ for which heavy memory usage can be a bottleneck, although it is also applicable to $(\widehat{\mathrm{LR}}_2)$. Moreover, tie-breaking is not considered in the dominance check and a partial path is assumed to be dominated only when a strictly dominating partial schedule is found.

For example, suppose that a part of $\widehat{G}_2^m$ $(m=1)$ is given by Fig. 1. In this case, there are five partial paths passing through the arc $(v_{3,21}^0, v_{2,25}^0)$ on $\widehat{G}_2^m$: $(v_{1,16}^0, v_{5,18}^0, v_{3,21}^0, v_{2,25}^0)$, $(v_{4,16}^0, v_{5,18}^0, v_{3,21}^0, v_{2,25}^0)$, $(v_{2,13}^0, v_{1,18}^0, v_{3,21}^0, v_{2,25}^0)$, $(v_{4,13}^0, v_{1,18}^0, v_{3,21}^0, v_{2,25}^0)$, and $(v_{5,14}^0, v_{2,18}^0, v_{3,21}^0, v_{2,25}^0)$. Since v_{2*}^0 occurs twice in $(v_{2,13}^0, v_{1,18}^0, v_{3,21}^0, v_{2,25}^0)$ and $(v_{5,14}^0, v_{2,18}^0, v_{3,21}^0, v_{2,25}^0)$, $\mathcal{E}((v_{3,21}^0, v_{2,25}^0))$ consists only of the other three partial paths. For the partial schedules corresponding to these partial paths, dominance check is performed. Here, assume that $\Delta_{1\to5\to3\to2}(25) > \Delta_{5\to3\to2\to1}(25)$, $\Delta_{4\to5\to3\to2}(25) > \Delta_{5\to3\to2\to4}(25)$, and $\Delta_{4\to1\to3\to2}(25) > \Delta_{3\to4\to2\to1}(25)$ hold, where $\Delta_{l\to k\to j\to i}(t)$ denotes the cost sum of jobs l, k, j, and i when they are sequenced in this order, so that job i is completed at t. In this case, we can eliminate the arc $(v_{3,21}^0, v_{2,25}^0)$.

The above arguments are for forward dynamic programming, but it is easy to see that parallel arguments hold in backward dynamic programming. Thus, the improvement is performed in both forward and backward dynamic programming.

4.3 Sophisticated step sizing in subgradient optimization

To search for a better set of Lagrangian multipliers, subgradient optimization is applied to the Lagrangian duals corresponding to (LR_1) and $(\widehat{\mathrm{LR}}_2)$. It is a standard way (cf. Fisher 1985) to decrease the step size in subgradient optimization when the solution is not updated for some number of iterations. However, it sometimes happens that the step size becomes too small and premature convergence occurs. To avoid this, the step size is controlled in such a sophisticated way that it is both decreased and increased depending on the situation.

There are six controllable parameters $(\gamma^{\mathrm{ini}}, \delta_{\mathrm{T}}, \delta_{\mathrm{S}}, \varepsilon, \kappa_{\mathrm{S}}, \kappa_{\mathrm{E}})$ for our subgradient procedure. The parameter γ^{ini} specifies the initial step size parameter. The step size parameter $\gamma^{(r)}$ is shrunken by κ_{S} if the best lower bound, i.e., the best solution of the relaxation is not updated for the last δ_{S} iterations. To avoid too small $\gamma^{(r)}$, it is expanded by κ_{E} if the best lower bound is updated. By using this $\gamma^{(r)}$, the multipliers $\boldsymbol{\mu}^{(r)}$ is updated at the rth iteration as follows:

$$\begin{aligned}\mu_i^{(r+1)} &:= \mu_i^{(r)} + \gamma^{(r)} \frac{\mathrm{UB} - \mathrm{LB}^{(r)}}{\sum_{j=1}^n (1 - \sum_{t=1}^T x_{jt}^{(r)})^2} \\ &\qquad \times \left(1 - \sum_{t=1}^T x_{it}^{(r)}\right), \quad i \in \mathcal{N}, \end{aligned} \tag{63}$$

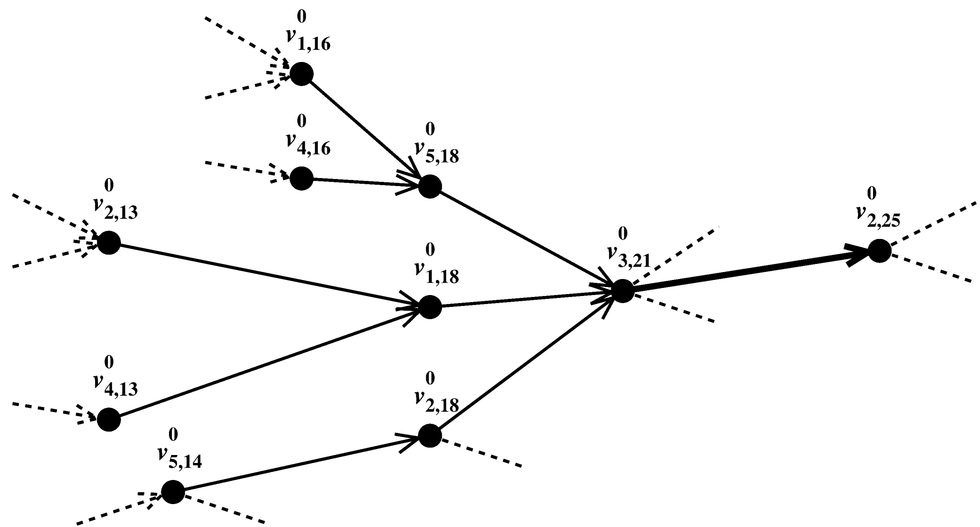

Fig. 1 An example of state elimination by the dominance of four successive jobs

where UB is the current upper bound. The procedure is terminated if

(a) The current upper bound is proved to be optimal

or if

(b) The best lower bound does not increase by $100\varepsilon/(1-\varepsilon)\%$, and the gap between the best lower and upper bounds does not decrease by $100\varepsilon\%$ for the last δ_{T} iterations

In the former case, the current upper bound is proved to be optimal if the gap between the best lower and upper bounds becomes less than one. It is because we assume that the job cost function $f_i(t)$ is integer-valued.

This procedure is applied first to (LR_1) and then to $(\widehat{\mathrm{LR}}_2)$ in our algorithm. For (LR_1), $\boldsymbol{\mu}^{(1)}$ is initialized by $\boldsymbol{\mu}^{(1)} = \mathbf{0}$, and UB by the initial upper bound. For $(\widehat{\mathrm{LR}}_2)$, $\boldsymbol{\mu}^{(1)}$ is initialized by the best multipliers obtained for (LR_1), and UB is updated if a better upper bound is found, which is searched for every five iterations. Moreover, state elimination is performed for $(\widehat{\mathrm{LR}}_2)$ every time when UB or the best lower bound is updated.

4.4 Upper bound improvement

The algorithm stated in Sect. 3.4 is effective only for small-size instances and is not applicable to large-size instances because the time and space complexities of the dynamic programming increase exponentially as N_2 increases. To cope with this difficulty, we combine two types of algorithms:

(a) A slightly modified version of the algorithm in Sect. 3.4
(b) A simple algorithm to avoid job duplication greedily

If (a) cannot be applied due to a large N_2, (b) is applied. Then, the enhanced dynasearch is applied to the solution to improve it more. Here, the enhanced dynasearch is an efficient neighborhood search for single-machine scheduling problems that utilizes the enhanced dynasearch neighborhood (Grosso et al. 2004) in the dynasearch (Congram et al. 2002).

To summarize, the upper bound computation procedure is given by the following:

1° Construct a partial job sequence by resolving job duplication in the current solution of a relaxation. It is done by removing all duplicated jobs except the one that appears the earliest (if the current direction is forward) or the latest (if the current direction is backward). For example, if the solution is $3, 5, 1, 5, 1, 7, 4, 5$, the partial job sequence $3, 5, 1, 7, 4$ is obtained if the direction is forward, and $3, 1, 7, 4, 5$ if backward.[1] Define the length of this partial schedule by N_1 and let $N_2 := n - N_1$. If $N_2 > 12$, go to 3°.

2° Solve the problem to find a job sequence minimizing the total job completion cost under the constraint that job precedence relations in the partial sequence are kept unchanged. It is done by the dynamic programming in Sect. 3.4. The current direction is reversed, and go to 4°.

3° Construct a feasible schedule by avoiding job duplication greedily. Here, only the algorithm corresponding to forward dynamic programming for $(\widehat{\mathrm{LR}}_2)$ is explained. On the graph $\widehat{G}_{\mathrm{S}}$, go along the shortest path from $v_{n+1,T+1}$ to v_{00} and append to a list $\mathcal{L}$ jobs corresponding to the visited nodes. If job duplication occurs, in other words, if the node v_{it} is visited although job i is

[1] Since (LR) is not solved in the proposed algorithm, jobs never occur successively in feasible solutions of the relaxations. Therefore, the example solution $3, 5, 5, 1, 1, 7, 4, 5$ in Sect. 3.4 that is taken from the original paper (Ibaraki and Nakamura 1994) is not appropriate here. If we dare to construct a partial sequence from this solution, the same sequences with those for $3, 5, 1, 5, 1, 7, 4, 5$ will be obtained.

already in $\mathcal{L}$, find $i' = \arg\min_{j \notin \mathcal{L}} \widehat{y}_2(v_{jt};\ \boldsymbol{\mu})$. If such i' does not exist, choose i' arbitrarily from $\mathcal{N} \backslash \mathcal{L}$. Then, append job i' to $\mathcal{L}$ and go along the shortest path again from $v_{i't}$ to v_{00}. Repeat the procedure until v_{00} is reached.

4° Improve the obtained solution by the enhanced dynasearch.

In the algorithm of Sect. 3.4, a partial schedule is constructed by removing all duplicated jobs (except those occurring successively). On the other hand, in our algorithm it is constructed by removing all duplicated jobs *except one*. Because the latter partial schedule is not shorter than the former, N_2 decreases and the dynamic programming in Sect. 3.4 is more likely to be applied.

4.5 Improved choice of state–space modifiers

In our algorithm, $(\widehat{\mathrm{LR}}_2^m)$ is solved successively by adding state–space modifiers. Since the way of assigning state–space modifiers to jobs affects the efficiency of the algorithm much, it is improved from the original algorithm in Sect. 3.5.

When state–space modifiers are determined as in Sect. 3.5, it can happen that the same job is chosen twice at different iterations. To be more precise, the state–space modifiers of some job i can be $q_i^{m_1} \neq 0$ and $q_i^{m_2} \neq 0$ for $m_1 \neq m_2$. Let us assume that the state–space modifiers of the two jobs i_1 and i_2 are chosen as in (53). Since the constraint (11) requires that the total modifier value should be 3 ($Q^m = 3$), the following two schedules are both feasible for the relaxation:

(a) Both the jobs i_1 and i_2 occur exactly once
(b) Job i_1 occurs three times, but job i_2 never occurs

Therefore, it is not ensured that job i_2 occurs in the solutions of the relaxation. As a result, a nonzero modifier may be assigned to job i_2 at another iteration.

To avoid this, we simply assign an independent set of state–space modifiers to each job. When, for example, jobs i_1 and i_2 are selected, we choose state–space modifiers as follows:

$$q_i^m = \begin{cases} 1 & \text{if } i = i_1, \\ 0 & \text{otherwise,} \end{cases} \qquad q_i^{m+1} = \begin{cases} 1 & \text{if } i = i_2, \\ 0 & \text{otherwise.} \end{cases} \tag{64}$$

In this case, it is ensured that these two jobs should occur exactly once. This can also be checked by the fact that the constraint (11) reduces to the constraint (4) for a subset of jobs if all the modifiers are chosen as (64). In other words, the constraint on the number of job occurrences (4) is once relaxed for all jobs, but the constraint for a subset of jobs is recovered in the relaxation. Hence, it is easy to see that at most $(n-1)$ sets of state–space modifiers are necessary to obtain an optimal solution of the original problem (P). It is also clear that the time complexity of the dynamic programming is the same as that in the original algorithm.

In our algorithm, at maximum three jobs are assigned nonzero modifiers at each iteration. The number of such jobs, m_{c} (≤ 3), is determined by the current memory usage. Let M be the memory occupation ratio ($M =$ (current memory usage)/(maximum memory size)). Then, m_{c} is determined so that

$$m_{\mathrm{c}} = \begin{cases} 1 & \text{if } 2^{-2} < M, \\ 2 & \text{if } 2^{-3} < M \leq 2^{-2}, \\ 3 & \text{if } M \leq 2^{-3}. \end{cases} \tag{65}$$

It is because the memory usage is doubled in the worst case when one set of modifiers is added.

These m_{c} jobs are selected by the following two strategies:

(S1) As in the original algorithm stated in Sect. 3.5, the jobs that do not occur in the current solution of $(\widehat{\mathrm{LR}}_2^{m-1})$ are selected. If there are more than m_{c} jobs, the number of occurrences of v_{i*}^* in the graph $\widehat{G}_{\mathrm{S}}^{m-1}$ is counted for each job $i \in \mathcal{N}$, and those jobs that occur less frequently are selected first. If there are fewer than m_{c} jobs, those that occur more than once in the solution are selected.

(S2) Jobs occurring less frequently in the graph $\widehat{G}_{\mathrm{S}}^{m-1}$ are selected, regardless of whether they occur in the current solution or not.

The first strategy is similar to that in the original algorithm. The only difference is that it explicitly specifies how to break ties when there are many jobs that do not occur in the current solution. Since the influence of adding modifiers is assumed to be suppressed if we assign a nonzero modifier to a job occurring less frequently in the graph $\widehat{G}_{\mathrm{S}}^{m-1}$, ties are broken by the number of occurrences in $\widehat{G}_{\mathrm{S}}^{m-1}$. On the other hand, the second strategy selects jobs only by the number of occurrences in $\widehat{G}_{\mathrm{S}}^{m-1}$ to suppress the increase of dynamic programming states as much as possible. However, in this case, the lower bound may not improve even when state–space modifiers are added, and hence the number of iterations in Stage 3 tends to increase. Thus, it depends on the situation which strategy is better or not. To check this, we performed some preliminary experiments and the second strategy turned out to be more effective when the number of dynamic programming states is large. Therefore, the two strategies are switched by the memory occupation ratio M just before we start adding state–space modifiers. More specifically, if M is less than 2^{-6}, the strategy (S1) is applied. Otherwise, (S2) is applied.

4.6 Overall algorithm

The proposed algorithm is summarized as follows:

Stage 1 Compute the initial upper bound UB by applying the enhanced dynasearch to the best of the following three schedules:

(a) An SPT (shortest processing time order) schedule
(b) A schedule sequenced greedily in the forward direction
(c) A schedule sequenced greedily in the backward direction

Apply subgradient optimization to the Lagrangian dual corresponding to (LR_1) for a better set of Lagrangian multipliers. If the gap between UB and $L_1(\boldsymbol{\mu})$ is less than one, halt. After the subgradient optimization is terminated, forward dynamic programming is applied for the best multipliers $\boldsymbol{\mu}^{\mathrm{stage1}}$ obtained in the subgradient optimization. An upper bound is computed and UB is updated if UB is dominated. In this case, the gap is checked again. Then, backward dynamic programming is applied and state elimination is performed. That is, the node v_{it} is eliminated from the graph G if

$$\mathrm{UB}-1 < h_1\big(v_{it};\ \boldsymbol{\mu}^{\mathrm{stage1}}\big) + H_1\big(v_{it};\ \boldsymbol{\mu}^{\mathrm{stage1}}\big) + \sum_{i=1}^{n} \mu_i^{\mathrm{stage1}} \tag{66}$$

is satisfied.

Stage 2 Construct $\widehat{G}_{\mathrm{S}}$ from G_{S} and apply subgradient optimization to the Lagrangian dual corresponding to $(\widehat{\mathrm{LR}}_2)$, starting from the multipliers $\boldsymbol{\mu}^{\mathrm{stage1}}$. At the initial iteration, dynamic programming states are eliminated by the states for (LR_1). More specifically, the arc $(v_{j,t-p_i}, v_{it})$ is eliminated from $\widehat{G}_{\mathrm{S}}$ if

$$\mathrm{UB}-1 < \widehat{h}_2\big((v_{j,t-p_i}, v_{it});\ \boldsymbol{\mu}^{\mathrm{stage1}}\big) + H_1\big(v_{it};\ \boldsymbol{\mu}^{\mathrm{stage1}}\big) + \sum_{i=1}^{n} \mu_i^{\mathrm{stage1}} \tag{67}$$

is satisfied, where $\widehat{h}_2(\cdot)$ is defined in a similar way to $h_2(\cdot)$ in (A.15)–(A.26). In the course of the subgradient optimization, an upper bound is computed in every five iterations and UB is updated if UB is dominated. Every time when either the best lower bound or UB is updated, the arc $(v_{j,t-p_i}, v_{it})$ satisfying

$$\mathrm{UB}-1 < \widehat{h}_2\big((v_{j,t-p_i}, v_{it});\ \boldsymbol{\mu}\big) + \widehat{H}_2\big((v_{j,t-p_i}, v_{it});\ \boldsymbol{\mu}\big) - \big(f_i(t)-\mu_i\big) + \sum_{i=1}^{n} \mu_i \tag{68}$$

is eliminated from $\widehat{G}_{\mathrm{S}}$. Halt if the gap between $\widehat{L}_2(\boldsymbol{\mu})$ and UB is less than one.

Stage 3 Solve $(\widehat{\mathrm{LR}}_2)$ for the best multipliers $\boldsymbol{\mu}^{\mathrm{stage2}}$ by both forward and backward dynamic programming, and perform state elimination. Also consider the dominance of four successive jobs. Check the memory occupation ratio M and determine the modifier strategy. Then, state–space modifiers being successively added, $(\widehat{\mathrm{LR}}_2^m)$ is solved by forward or backward dynamic programming in turns until the gap between $\widehat{L}_2^m(\boldsymbol{\mu}^{\mathrm{stage2}})$ and UB becomes less than one. The arc $(v_{j,t-p_i}^{\boldsymbol{b}-\boldsymbol{q}_i}, v_{it}^{\boldsymbol{b}})$ is eliminated if

$$\begin{aligned}\mathrm{UB}-1 < {}& \widehat{h}_2^m\big((v_{j,t-p_i}^{\boldsymbol{b}-\boldsymbol{q}_i}, v_{it}^{\boldsymbol{b}});\ \boldsymbol{\mu}^{\mathrm{stage2}}\big) \\ &+ \widehat{H}_2^{m-m_c}\big((v_{j,t-p_i}^{\boldsymbol{b}''-\boldsymbol{q}_i''}, v_{it}^{\boldsymbol{b}''});\ \boldsymbol{\mu}^{\mathrm{stage2}}\big) \\ &- \big(f_i(t)-\mu_i\big) + \sum_{i=1}^{n} \mu_i^{\mathrm{stage2}},\end{aligned} \tag{69}$$

or

$$\begin{aligned}\mathrm{UB}-1 < {}& \widehat{h}_2^{m-m_c}\big((v_{j,t-p_i}^{\boldsymbol{b}''-\boldsymbol{q}_i''}, v_{it}^{\boldsymbol{b}''});\ \boldsymbol{\mu}^{\mathrm{stage2}}\big) \\ &+ \widehat{H}_2^m\big((v_{j,t-p_i}^{\boldsymbol{b}-\boldsymbol{q}_i}, v_{it}^{\boldsymbol{b}});\ \boldsymbol{\mu}^{\mathrm{stage2}}\big) \\ &- \big(f_i(t)-\mu_i\big) + \sum_{i=1}^{n} \mu_i^{\mathrm{stage2}},\end{aligned} \tag{70}$$

where $\boldsymbol{b}'' = (b^1, \ldots, b^{m-m_c})$ and $\boldsymbol{q}_i'' = (q_i^1, \ldots, q_i^{m-m_c})$. The dominance of four successive jobs is considered every time when $(\widehat{\mathrm{LR}}_2^m)$ is solved. An upper bound is computed only when the lower bound is updated, and UB is updated if necessary.

In every stage of the algorithm, a dynamic programming state is eliminated if the lower bound for passing through the state is greater than (UB − 1). It is because the job cost function $f_i(t)$ is assumed to be integer-valued.

5 Computational experiments

In this section, the effectiveness of our algorithm will be examined by computational experiments. We will test our algorithm on two types of single-machine scheduling problems: the TWT (Total Weighted Tardiness) problem and the TWET (Total Weighted Earliness–Tardiness) problem without machine idle time. In the TWT problem, the cost function of job i is given by

$$f_i(t) = w_i \max(t-d_i, 0), \tag{71}$$

and in the TWET problem it is given by

$$f_i(t) = \alpha_i \max(d_i - t, 0) + \beta_i \max(t-d_i, 0), \tag{72}$$

where w_i, α_i, and β_i are assumed to be positive integers, and d_i is a nonnegative integer.

With regard to the TWT problem, the benchmark instances by Crauwels et al. (1998) are used as 40, 50, and 100 jobs instances ($n = 40, 50, 100$), which are available from the OR-Library: http://people.brunel.ac.uk/~mastjjb/jeb/info.html. These instances were generated randomly by the following procedure. For each job i, an integer processing time p_i and an integer weight w_i were generated from the uniform distributions $[1, 100]$ and $[1, 10]$, respectively. An integer duedate d_i was generated from the uniform distribution $[T(1 - \mathrm{TF} - \mathrm{RDD}/2),\ T(1 - \mathrm{TF} + \mathrm{RDD}/2)]$, where $\mathrm{TF} = 0.2, 0.4, 0.6, 0.8, 1.0$ and $\mathrm{RDD} = 0.2, 0.4, 0.6, 0.8, 1.0$ are the parameters to control the tightness and the range of duedates, respectively. For each combination of n, TF and RDD, five instances were generated. Thus, 125 problem instances were generated for each n. The instances for $n = 150, 200, 250, 300$ are generated in a similar way to the OR-Library instances. The only difference is how to generate duedates. In the above procedure, it is possible that duedates become negative when TF and RDD are large. Since negative duedates can be converted to zero duedates without changing the problem complexity, they are set to be zero in the OR-library instances. As a result, more than half of the jobs have zero duedates in some instances. To avoid such a situation, duedates are generated from the uniform distribution $[\max(T(1 - \mathrm{TF} - \mathrm{RDD}/2), 0),\ T(1 - \mathrm{TF} + \mathrm{RDD}/2)]$ for $n = 150, 200, 250, 300$.

The TWET instances are generated from the TWT instances, where an integer earliness weight α_i of job i is generated from the uniform distribution $[1, 10]$, and an integer tardiness weight β_i is chosen as $\beta_i = w_i$.

Computation is performed on a 2.4 GHz Pentium 4 desktop computer with 512 MB RAM by running a code written in C (gcc). The maximum memory size for dynamic programming states (for storing the graph structure) is restricted to 384 MB. The tie-breaking rule for the dominance of two adjacent jobs in Sect. 4.1 is determined by some preliminary experiments, and the lexicographical order of (d_i, p_i, i) is used (the smaller should precede the larger). The six parameters $(\gamma^{\mathrm{ini}}, \delta_\mathrm{T}, \delta_\mathrm{S}, \epsilon, \kappa_\mathrm{S}, \kappa_\mathrm{E})$ for the subgradient optimization in Sect. 4.3 are also determined by preliminary experiments and are chosen as $(2.0, \lfloor n/4 \rfloor, 4, 0.02, 0.75, 2.0)$ in Stage 1, and $(2.0, \lfloor n/4 \rfloor, 4, 0.002, 0.8, 2.0)$ in Stage 2.

To compare with the proposed algorithm, we implemented an improved version of the algorithm by Ibaraki and Nakamura (1994). The improvements from the original algorithm are:

(a) Faster dynamic programming is applied: the time complexities of the dynamic programming for (LR_1), (LR_2), and (LR_2^m) are $O(nT)$, $O(n^2T)$, and $O(n^2T\prod_{l=1}^{m}(1 + Q^l))$, respectively (see the discussion in Sect. 4.1). Accordingly, the memory usage is reduced.
(b) Dynamic programming states are eliminated when (lower bound) $>$ (upper bound) $-$ 1. In addition, the current upper bound is assumed to be optimal when (upper bound) $-$ (lower bound) < 1.
(c) The algorithm in Sect. 3.4 to compute an upper bound is applied when $N_2 \le 12$ in Stage 1, and when $N_2 \le 15$ in the other stages.
(d) The two parameters (K_1, K_2) of the subgradient optimization in Sect. 3.3 are re-adjusted for each problem size and problem type by some preliminary experiments so that the number of optimally solved instances is maximized and CPU time is minimized.

First, the results for the TWT instances are shown in Table 1 where the average (ave) and maximum (max) CPU times over optimally solved instances (solved) are given in seconds. In this table, the computational results by Pan and Shi (2007) are also shown. Their branch-and-bound algorithm specialized for the TWT problem utilizes a lower bound based on the transportation problem relaxation and several known techniques for this problem are integrated into it. They reported that all the OR-Library instances were optimally solved for the first time by this algorithm. However, it took at maximum 9 hours to solve the 100 jobs instances on a 2.8 GHz Pentium 4 computer. On the other hand, our algorithm can solve these instances within 40 seconds and even the 300 jobs instances within 1 hour on a 2.4 GHz Pentium 4 computer. Clearly, our general algorithm outperforms the specialized algorithm by Pan and Shi (2007). Moreover, the improved version of the original algorithm failed to solve some of the 40 jobs instances. This fact shows the effectiveness of our algorithm.

Next, the results for the TWET instances are shown in Table 2 where our algorithm is compared only with the improved version of the original algorithm. The most recent exact algorithm for the TWET problem without machine idle time is, to the best of authors' knowledge, the branch-and-bound algorithm proposed by Liaw (1999), which is based on the results for the TWT problem by Potts and Van Wassenhove (1985). However, the improved version of the algorithm by Ibaraki and Nakamura (1994) seems to be much faster because the Liaw's algorithm failed to solve some of 40 jobs instances within one hour on a 266 MHz Pentium II computer.

From Table 2, we can see that our algorithm outperforms the improved version of the original algorithm, but not all of the 300 jobs instances are optimally solved due to the excess memory usage. However, two out of the unsolved five instances can be solved by tuning the parameters in subgradient optimization. The other three can be solved by tuning the parameters and by increasing the maximum memory size to 768 MB.

Table 1 Computational results for the TWT instances

n	Proposed			Original			Pan and Shi (2007)[a]		
	CPU time		Solved	CPU time		Solved	CPU time		Solved
	Ave	Max		Ave	Max		Ave	Max	
40	0.19	0.73	125	2.56	28.02	88	68.98	235	125
50	0.39	0.89	125				142.8	466	125
100	6.42	38.52	125				1811	32400	125
150	26.12	111.44	125						
200	74.24	256.08	125						
250	170.36	610.43	125						
300	353.61	2066.38	125						

[a]Results on a 2.8 GHz Pentium 4 computer

Table 2 Computational results for the TWET instances

n	Proposed			Original		
	Ave	Max	Solved	Ave	Max	Solved
40	0.23	0.45	125	0.98	22.54	125
50	0.51	1.16	125	5.18	41.72	125
100	10.04	21.03	125	133.35	409.44	69
150	55.17	141.36	125			
200	195.08	561.99	125			
250	538.20	2356.81	125			
300	1317.50	6391.81	120			

Table 3 CPU times and gaps in Stage 1 of the proposed algorithm

n	TWT instances					TWET instances				
	CPU time (s)		Gap (%)		Solved	CPU time (s)		Gap (%)		Solved
	Ave	Max	Ave	Max		Ave	Max	Ave	Max	
40	0.13	0.38	22.93	100.00	33/125	0.19	0.38	3.24	26.13	10/125
50	0.28	0.86	22.40	100.00	23/125	0.41	0.73	2.86	20.75	1/125
100	4.22	14.49	15.75	100.00	22/125	5.64	7.96	1.60	8.48	0/125
150	17.65	55.72	11.33	100.00	26/125	20.83	60.95	1.48	10.63	0/125
200	45.10	135.27	13.56	88.44	25/125	51.51	71.92	1.43	11.68	0/125
250	91.12	287.17	10.78	95.04	26/125	102.56	127.97	1.15	7.00	0/125
300	168.75	479.34	11.89	100.00	25/125	177.90	227.00	1.16	7.10	0/120

The detailed results of our algorithm are shown in Tables 3, 4, and 5. In Tables 3 and 4, CPU times, gaps between the lower and upper bounds (100(UB − LB)/UB), and the numbers of optimally solved instances in Stage 1 and Stage 2 are given separately. CPU times and the numbers of state–space modifiers added in Stage 3 are given in Table 5. From Tables 3 and 4, we can see that the lower bound for the TWT problem is so tight that 33 out of 125 instances with 40 jobs are optimally solved in Stage 1, and 91 out of 92 are optimally solved in Stage 2. The lower bound for the TWET problem is less effective, but 10 out of 125 instances with 40 jobs are optimally solved in Stage 1, and 95 out of 115 in Stage 2. The gap between the lower and upper bounds in Stage 1 for the TWT problem is relatively large because subgradient optimization sometimes fails to improve the lower bound from its initial value. Although the parameters in the subgradient optimization can be chosen so that the lower bound improves in Stage 1, they are tuned to minimize the average of the total CPU time in our experiments. Since Stage 1 can be regarded as a procedure to find good initial

Table 4 CPU times and gaps in Stage 2 of the proposed algorithm

n	TWT instances					TWET instances				
	CPU time (s)		Gap (%)		Solved	CPU time (s)		Gap (%)		Solved
	Ave	Max	Ave	Max		Ave	Max	Ave	Max	
40	0.05	0.66	0.00	0.11	91/92	0.02	0.29	0.29	6.28	95/115
50	0.10	0.74	0.02	0.57	93/102	0.08	0.49	0.41	5.99	90/124
100	2.39	35.58	0.03	0.61	69/103	3.78	14.19	0.22	1.99	22/125
150	9.37	104.92	0.04	0.48	50/99	28.27	82.20	0.19	1.81	5/125
200	32.34	239.49	0.03	0.56	47/100	106.18	374.53	0.20	3.01	1/125
250	88.94	456.37	0.03	0.33	36/99	297.42	2153.37	0.14	1.87	1/125
300	204.90	1722.53	0.03	0.39	35/100	643.63	2995.35	0.14	1.82	0/120

Table 5 CPU times and numbers of state–space modifiers in Stage 3 of the proposed algorithm

n	TWT instances				TWET instances			
	CPU time (s)		Modifiers		CPU time (s)		Modifiers	
	Ave	Max	Ave	Max	Ave	Max	Ave	Max
40	0.00	0.00	3.00	3	0.02	0.15	9.90	24
50	0.01	0.04	8.33	15	0.05	0.63	13.56	33
100	0.23	1.38	18.44	45	0.76	5.71	24.55	54
150	0.88	6.20	22.39	57	6.33	86.27	44.44	147
200	3.93	29.15	33.13	86	37.69	361.39	91.11	199
250	10.78	64.72	47.51	249	139.35	1506.94	162.92	249
300	29.40	117.16	90.75	297	495.97	3243.63	238.07	299

multipliers for Stage 2, it is more important whether tight lower and upper bounds are obtained or not in Stage 2. In this sense, the lower bound improvement by the dominance of two adjacent jobs and the sophisticated step sizing in subgradient optimization work very well, and the gap for the TWT problem is very small in Stage 2. For the TWET problem, the gap in Stage 1 is smaller than that for the TWT problem, while it is larger in Stage 2. This implies that subgradient optimization works better for the TWET problem than for the TWT problem, but the gap itself is larger for the TWET problem than for the TWT problem. This is consistent with the fact that the proposed algorithm is less efficient for the TWET problem, and some of 300 jobs instances are unsolved.

From Table 5, it can be checked that the number of state–space modifiers added in Stage 3 (m) increases a lot as n increases. For the larger instances, memory usage increases and the second modifier selection strategy is applied. In this strategy, it often happens that the lower bound is not improved even if modifiers are added, and hence many sets of modifiers are necessary. Table 5 also indicates that the number of added modifiers is larger for the TWET problem than for the TWT problem. It is because the gap between the lower and upper bounds is larger for the TWET problem,

Table 6 The effects of proposed improvements to maximum optimally solvable problem sizes

Method	Optimally solvable size	
	TWT	TWET
Original	<40	<100
A1	<40	<150
A2	≥ 300	<250
A3	≥ 300	<200
Proposed	≥ 300	<300

and thus the second modifier selection strategy is applied to even smaller instances due to heavier memory usage.

Finally, the following three algorithms are tested to examine the effects of the proposed improvements:

(A1) The proposed algorithm without the dominance of two adjacent jobs and without four successive jobs (without the improvements in Sects. 4.1 and 4.2),

(A2) The proposed algorithm without the dominance of four successive jobs (without the improvement in Sect. 4.2),

(A3) The proposed algorithm only with the first modifier selection strategy.

Table 6 shows the maximum problem sizes such that the algorithms can solve all the instances optimally. It can be seen that the dominance of two adjacent jobs is very effective, especially for the TWT problem. When duedates are not so tight, there are many optimal solutions of the TWT problem because adjacent pairs of on-time jobs in an optimal solution can be interchanged arbitrarily as far as they do not become tardy. This increases dynamic programming states and makes state elimination less effective. Hence, the original algorithm or the algorithm (A) fails to solve instances even with 40 jobs. With the constraint on two adjacent jobs, the processing order of such on-time jobs is restricted, which contributes much to the reduction of dynamic programming states. As a consequence, the algorithm succeeded in solving 300 jobs instances optimally. On the other hand, the dominance of four successive jobs and the second modifier selection strategy do not have so much impact on the scalability of the algorithm. Nevertheless, the algorithm cannot solve some of the TWET instances with 200 or 250 jobs optimally without these improvements. This fact confirms the effectiveness of our proposed improvements.

6 Conclusion

In this paper, we proposed an exact algorithm for the general single-machine scheduling problem without machine idle time to minimize the total job completion cost. We proposed several improvements for the previous algorithm based on the SSDP method to reduce both the memory usage and computational efforts. Numerical experiments showed that the proposed algorithm can solve instances even with 300 jobs. It was also shown that our algorithm outperforms the existing specialized algorithms for the single-machine total weighted tardiness problem and the single-machine total weighted earliness–tardiness problem without machine idle time.

Our algorithm can handle arbitrary job completion costs, but it is applicable only to the problem without machine idle time. Hence, it will be necessary to extend our algorithm to those problems with machine idle time. We are now working on this extension and some preliminary results show that this direction of research is quite promising. It will be also important to extend our results to more general problems with setup times, precedence constraints, and so on.

Acknowledgements This work is partially supported by Grant-in-Aid for Young Scientists (B) 19760273, from Japan Society for the Promotion of Science (JSPS).

Appendix A: Dynamic programming recursions for (LR_1), (LR_2), and (LR_2^m)

Here, we will present the dynamic programming recursions to solve the relaxations (LR_1), (LR_2), and (LR_2^m).

The relaxation (LR_1) can be solved by dynamic programming in $O(nT)$ time. The forward recursion is given by

$$L_1(\boldsymbol{\mu}) = y_1(T+1;\ \boldsymbol{\mu}) + \sum_{i=1}^{n} \mu_i, \tag{A.1}$$

$$y_1(t;\ \boldsymbol{\mu}) = \min_{v_{it}\in V_{\mathrm{O}}} h_1(v_{it};\ \boldsymbol{\mu}), \tag{A.2}$$

$$\lambda_1(t;\ \boldsymbol{\mu}) = \arg\min_{\substack{i\\ v_{it}\in V_{\mathrm{O}}}} h_1(v_{it};\ \boldsymbol{\mu}), \tag{A.3}$$

$$z_1(t;\ \boldsymbol{\mu}) = \min_{\substack{v_{it}\in V_{\mathrm{O}}\\ i\neq\lambda_1(t;\ \boldsymbol{\mu})}} h_1(v_{it};\ \boldsymbol{\mu}), \tag{A.4}$$

$$y_1(0;\ \boldsymbol{\mu}) = 0, \qquad z_1(0;\ \boldsymbol{\mu}) = +\infty, \qquad \lambda_1(0;\ \boldsymbol{\mu}) = \phi, \tag{A.5}$$

$$h_1(v_{it};\ \boldsymbol{\mu}) = \begin{cases} y_1(t-p_i;\ \boldsymbol{\mu}) + f_i(t) - \mu_i \\ \quad \text{if } i \neq \lambda_1(t-p_i;\ \boldsymbol{\mu}), \\ z_1(t-p_i;\ \boldsymbol{\mu}) + f_i(t) - \mu_i \\ \quad \text{if } i = \lambda_1(t-p_i;\ \boldsymbol{\mu}). \end{cases} \tag{A.6}$$

In (A.1)–(A.6), $y_1(t;\ \boldsymbol{\mu})$ and $z_1(t;\ \boldsymbol{\mu})$ denote the shortest and the second shortest path lengths from v_{00} to v_{*t} on G_{S}, respectively.

It would be more intuitive to compute $h_1(v_{it};\ \boldsymbol{\mu})$ by taking the minimum over the connected arcs:

$$\begin{aligned} h_1(v_{it};\ \boldsymbol{\mu}) &= \min_{(v_{j,t-p_i}, v_{it})\in A_{\mathrm{S}}} h_1(v_{j,t-p_i};\ \boldsymbol{\mu}) + f_i(t) - \mu_i \\ &= \min_{\substack{v_{j,t-p_i}\in V\\ j\neq i}} h_1(v_{j,t-p_i};\ \boldsymbol{\mu}) + f_i(t) - \mu_i. \end{aligned} \tag{A.7}$$

However, the time complexity for (LR_1) is given by $O(n^2T)$ when this is applied recursively. To reduce it, the relation

$$\min_{\substack{v_{j,t-p_i}\in V\\ j\neq i}} h_1(v_{j,t-p_i};\ \boldsymbol{\mu}) = \begin{cases} y_1(t-p_i;\ \boldsymbol{\mu}) & \text{if } i \neq \lambda_1(t-p_i;\ \boldsymbol{\mu}), \\ z_1(t-p_i;\ \boldsymbol{\mu}) & \text{if } i = \lambda_1(t-p_i;\ \boldsymbol{\mu}), \end{cases} \tag{A.8}$$

is utilized in the forward recursion (A.1)–(A.6).

The backward recursion is given in a similar way by

$$L_1(\boldsymbol{\mu}) = Y_1(0;\ \boldsymbol{\mu}) + \sum_{i=1}^{n} \mu_i, \tag{A.9}$$

$$Y_1(t;\ \boldsymbol{\mu}) = \min_{v_{i,t+p_i}\in V_{\mathrm{O}}} \big\{H_1(v_{i,t+p_i};\ \boldsymbol{\mu}) + f_i(t+p_i) - \mu_i\big\}, \tag{A.10}$$

$$\Lambda_1(t;\ \boldsymbol{\mu}) = \arg\min_{\substack{i\\ v_{i,t+p_i}\in V_{\mathrm{O}}}} \big\{H_1(v_{i,t+p_i};\ \boldsymbol{\mu}) + f_i(t+p_i) - \mu_i\big\}, \tag{A.11}$$

$$Z_1(t;\ \boldsymbol{\mu}) = \min_{\substack{v_{i,t+p_i}\in V_{\mathrm{O}}\\ i\neq \Lambda_1(t;\ \boldsymbol{\mu})}} \{H_1(t+p_i, i;\ \boldsymbol{\mu}) + f_i(t+p_i) - \mu_i\}, \tag{A.12}$$

$$Y_1(T+1;\ \boldsymbol{\mu}) = 0, \qquad Z_1(T+1;\ \boldsymbol{\mu}) = +\infty, \quad \Lambda_1(T+1;\ \boldsymbol{\mu}) = \phi, \tag{A.13}$$

$$H_1(v_{it};\ \boldsymbol{\mu}) = \begin{cases} Y_1(t;\ \boldsymbol{\mu}) & \text{if } i \neq \Lambda_1(t;\ \boldsymbol{\mu}),\\ Z_1(t;\ \boldsymbol{\mu}) & \text{if } i = \Lambda_1(t;\ \boldsymbol{\mu}), \end{cases} \tag{A.14}$$

where $Y_1(t;\ \boldsymbol{\mu})$ and $Z_1(t;\ \boldsymbol{\mu})$ denote the shortest and the second shortest path lengths from v_{*t} to $v_{n+1,T+1}$ on G_{S}, respectively.

In the case of the relaxation (LR$_2$), the forward recursion is given by

$$L_2(\boldsymbol{\mu}) = y_2(v_{n+1,T+1};\ \boldsymbol{\mu}) + \sum_{i=1}^{n} \mu_i, \tag{A.15}$$

$$y_2(v_{it};\ \boldsymbol{\mu}) = \min_{(v_{j,t-p_i}, v_{it})\in A_{\mathrm{S}}} h_2\big((v_{j,t-p_i}, v_{it});\ \boldsymbol{\mu}\big), \tag{A.16}$$

$$\lambda_2(v_{it};\ \boldsymbol{\mu}) = \arg \min_{\substack{j\\ (v_{j,t-p_i}, v_{it})\in A_{\mathrm{S}}}} h_2\big((v_{j,t-p_i}, v_{it});\ \boldsymbol{\mu}\big), \tag{A.17}$$

$$z_2(v_{it};\ \boldsymbol{\mu}) = \min_{\substack{(v_{j,t-p_i}, v_{it})\in A_{\mathrm{S}}\\ j\neq \lambda_2(v_{it};\ \boldsymbol{\mu})}} h_2\big((v_{j,t-p_i}, v_{it});\ \boldsymbol{\mu}\big), \tag{A.18}$$

$$y_2(v_{00};\ \boldsymbol{\mu}) = 0, \qquad z_2(v_{00};\ \boldsymbol{\mu}) = +\infty, \quad \lambda_2(v_{00};\ \boldsymbol{\mu}) = \phi, \tag{A.19}$$

$$h_2\big((v_{j,t-p_i}, v_{it});\ \boldsymbol{\mu}\big) = \begin{cases} y_2(v_{j,t-p_i};\ \boldsymbol{\mu}) + f_i(t) - \mu_i\\ \quad \text{if } i \neq \lambda_2(v_{j,t-p_i};\ \boldsymbol{\mu}),\\ z_2(v_{j,t-p_i};\ \boldsymbol{\mu}) + f_i(t) - \mu_i\\ \quad \text{if } i = \lambda_2(v_{j,t-p_i};\ \boldsymbol{\mu}), \end{cases} \tag{A.20}$$

and the backward recursion is given by

$$L_2(\boldsymbol{\mu}) = Y_2(v_{00};\ \boldsymbol{\mu}) + \sum_{i=1}^{n} \mu_i, \tag{A.21}$$

$$Y_2(v_{it};\ \boldsymbol{\mu}) = \min_{(v_{it}, v_{j,t+p_j})\in A_{\mathrm{S}}} H_2\big((v_{it}, v_{j,t+p_j});\ \boldsymbol{\mu}\big), \tag{A.22}$$

$$\Lambda_2(v_{it};\ \boldsymbol{\mu}) = \arg \min_{\substack{j\\ (v_{it}, v_{j,t+p_j})\in A_{\mathrm{S}}}} H_2\big((v_{it}, v_{j,t+p_j})\ \boldsymbol{\mu}\big), \tag{A.23}$$

$$Z_2(v_{it};\ \boldsymbol{\mu}) = \min_{\substack{(v_{it}, v_{j,t+p_j})\in A_{\mathrm{S}}\\ j\neq \Lambda_2(v_{it};\ \boldsymbol{\mu})}} H_2\big((v_{it}, v_{j,t+p_j});\ \boldsymbol{\mu}\big), \tag{A.24}$$

$$Y_2(v_{n+1,T+1};\ \boldsymbol{\mu}) = 0, \quad Z_2(v_{n+1,T+1};\ \boldsymbol{\mu}) = +\infty, \quad \Lambda_2(v_{n+1,T+1};\ \boldsymbol{\mu}) = \phi, \tag{A.25}$$

$$H_2\big((v_{it}, v_{j,t+p_j});\ \boldsymbol{\mu}\big) = \begin{cases} Y_2(v_{j,t+p_j};\ \boldsymbol{\mu}) + f_j(t+p_j) - \mu_j\\ \quad \text{if } i \neq \Lambda_2(v_{j,t+p_j};\ \boldsymbol{\mu}),\\ Z_2(v_{j,t+p_j};\ \boldsymbol{\mu}) + f_j(t+p_j) - \mu_j\\ \quad \text{if } i = \Lambda_2(v_{j,t+p_j};\ \boldsymbol{\mu}), \end{cases} \tag{A.26}$$

where their time complexities are $O(n^2T)$.

The forward and backward recursions in the dynamic programming for (LR$_2^m$) are given by changing A_{S} to A_{S}^m and the other variables accordingly in (A.15)–(A.26): L_2 to L_2^m, y_2 to y_2^m, Y_2 to Y_2^m, and so on. Hence, (LR$_2^m$) can be solved in $O(n^2T\prod_{l=1}^{m}(1+Q^l))$.

Appendix B: Proof of Proposition 4.1

Consider that there exists an optimal schedule S_{opt} that does not satisfy the condition of the proposition. We will show that this schedule can be converted into another optimal schedule satisfying the condition by interchanging adjacent jobs that break the condition. From the optimality of S_{opt} we assume, without loss of generality, that any adjacent jobs j and i in S_{opt} with $j \to i(t)$ satisfy

$$\Delta_{j\to i}(t) \leq \Delta_{i\to j}(t). \tag{B.1}$$

Otherwise, S_{opt} can be converted to a better schedule, and thus it is not optimal. Let us define by $\mathcal{B}_i(S_{\mathrm{opt}})$ $(i \in \mathcal{N})$ the set of jobs j that precede job i in S_{opt} and that satisfy $j \neq R_{ij}$. In other words, $\mathcal{B}_i(S_{\mathrm{opt}})$ denotes the set of jobs preceding job i that should be interchanged with job i when they become an immediate predecessor of job i. Similarly, we define by $\mathcal{A}_i(S_{\mathrm{opt}})$ $(i \in \mathcal{N})$ the set of jobs j that are preceded by job i in S_{opt} and that satisfy $j = R_{ij}$. Assume that jobs l and k satisfy $l \to k(t)$ in S_{opt}, and that

$$\Delta_{l\to k}(t) = \Delta_{k\to l}(t), \quad l \neq R_{kl} \tag{B.2}$$

is satisfied. Since these jobs break the condition (58), $l \in \mathcal{B}_k(S_{\mathrm{opt}})$ and $k \in \mathcal{A}_l(S_{\mathrm{opt}})$ from their definitions. Therefore, if we construct a new optimal schedule S'_{opt} by interchanging the two jobs k and l, then

$$\mathcal{B}_i(S'_{\mathrm{opt}}) = \begin{cases} \mathcal{B}_i(S_{\mathrm{opt}})\backslash\{l\} & \text{if } i = k,\\ \mathcal{B}_i(S_{\mathrm{opt}}) & \text{otherwise,} \end{cases} \tag{B.3}$$

and

$$\mathcal{A}_i(S'_{\mathrm{opt}}) = \begin{cases} \mathcal{A}_i(S_{\mathrm{opt}})\backslash\{k\} & \text{if } i = l,\\ \mathcal{A}_i(S_{\mathrm{opt}}) & \text{otherwise} \end{cases} \tag{B.4}$$

hold. Since $\mathcal{B}_i(S_{\mathrm{opt}})$ and $\mathcal{A}_i(S_{\mathrm{opt}})$ are finite sets, S_{opt} can be converted to another optimal schedule satisfying (58) after a finite number of interchanges.

References

Abdul-Razaq, T. S., & Potts, C. N. (1988). Dynamic programming state–space relaxation for single-machine scheduling. *Journal of the Operational Research Society*, *39*, 141–152.

Christofides, N., Mingozzi, A., & Toth, P. (1981). State–space relaxation procedures for the computation of bounds to routing problems. *Networks*, *11*, 145–164.

Congram, R. K., Potts, C. N., & van de Velde, S. L. (2002). An iterated dynasearch algorithm for the single-machine total weighted tardiness scheduling problem. *INFORMS Journal on Computing*, *14*, 52–67.

Crauwels, H. A. J., Potts, C. N., & Van Wassenhove, L. N. (1998). Local search heuristics for the single machine total weighted tardiness scheduling problem. *INFORMS Journal on Computing*, *10*, 341–350.

Dyer, M. E., & Wolsey, L. A. (1990). Formulating the single-machine sequencing problem with release dates as a mixed integer problem. *Discrete Applied Mathematics*, *26*, 255–270.

Fisher, M. L. (1973). Optimal solution of scheduling problems using Lagrange multipliers: part I. *Operations Research*, *21*, 1114–1127.

Fisher, M. L. (1985). An applications oriented guide to Lagrangian relaxation. *Interfaces*, *15*, 10–21.

Grosso, A., Della Croce, F., & Tadei, R. (2004). An enhanced dynasearch neighborhood for the single-machine total weighted tardiness scheduling problem. *Operations Research Letters*, *32*, 68–72.

Ibaraki, T. (1987). Enumerative approaches to combinatorial optimization. *Annals of Operations Research, 10–11*.

Ibaraki, T., & Nakamura, Y. (1994). A dynamic programming method for single machine scheduling. *European Journal of Operational Research*, *76*, 72–82.

Liaw, C.-F. (1999). A branch-and-bound algorithm for the single machine earliness and tardiness scheduling problem. *Computers & Operations Research*, *26*, 679–693.

Pan, Y., & Shi, L. (2007). On the equivalence of the max-min transportation lower bound and the time-indexed lower bound for single-machine scheduling problems. *Mathematical Programming, Series A*, *110*, 543–559.

Péridy, L., Pinson, É., & Rivreau, D. (2003). Using short-term memory to minimize the weighted number of late jobs on a single machine. *European Journal of Operational Research*, *148*, 591–603.

Potts, C. N., & Van Wassenhove, L. N. (1985). A branch and bound algorithm for the total weighted tardiness problem. *Operations Research*, *33*, 363–377.

Pritsker, A. A. B., Watters, L. J., & Wolfe, P. M. (1969). Multiproject scheduling with limited resources: a zero-one programming approach. *Management Science*, *16*, 93–108.

Sourd, F. (2006). A reinforced Lagrangean relaxation for non-preemptive single machine problem. In *Proceedings of tenth international workshop on project management and scheduling*, Poznań, Poland, 26–28 April 2006.

Sourd, F. (2008). New exact algorithms for one-machine earliness-tardiness scheduling. *INFORMS Journal on Computing*. doi: 10.1287/ijoc.1080.0287

Sousa, J. P., & Wolsey, L. A. (1992). A time indexed formulation of non-preemptive single machine scheduling problems. *Mathematical Programming*, *54*, 353–367.

Tanaka, S., Fujikuma, S., & Araki, M. (2006). A branch-and-bound algorithm based on Lagrangian relaxation for single-machine scheduling. In *Proceedings of international symposium on scheduling 2006* (pp. 148–153), Tokyo, Japan, 18–20 July 2006.

van den Akker, J. M., van Hoesel, C. P. M., & Savelsbergh, M. W. P. (1999). A polyhedral approach to single-machine scheduling problems. *Mathematical Programming*, *85*, 541–572.

van den Akker, J. M., Hurkens, C. A. J., & Savelsbergh, M. W. P. (2000). Time-indexed formulations for machine scheduling problems: Column generation. *INFORMS Journal on Computing*, *12*, 111–124.