Technical Report #99010

Formulation and Tabu Search Algorithm for the Resource Constrained Project Scheduling Problem*

Koji Nonobe and Toshihide Ibaraki
Department of Applied Mathematics and Physics
Graduate School of Informatics, Kyoto University, Kyoto 606-8501, Japan
{nonobe,ibaraki}@i.kyoto-u.ac.jp

April 1999
(revised in September 1999)
(further revised in September 2000)

Abstract

The resource constrained project scheduling problem (RCPSP) can formulate many scheduling problems including jobshop and flowshop scheduling problems. In this paper, we extend the definition of RCPSP further so that various complicated constraints and objective functions arising in practice can be handled; for example, each activity can be processed in one of the selectable modes, the available amounts of renewable resources may vary depending on the periods, setup activities can be dealt with, and complex objective functions can be handled. Then, we develop a tabu search based heuristic algorithm, which contains elaborations in representing solutions and in constructing neighborhood. Our code was tested for many benchmarks of RCPSP, and also for some problems from real applications. For a number of RCPSP instances, we found better solutions than the best ones found so far. These computational results indicate the effectiveness and usefulness of our approach.

1 Introduction

The resource constrained project scheduling problem (RCPSP) asks to determine start times of the participating activities under the constraints of various resources that are consumed by the activities, as well as precedence constraints between activities. For the RCPSP, many exact and approximate methods have been proposed so far. Recently, to deal with problems of large scale, meta-heuristic algorithms have been studied and brought successful computational results. In these developments [1, 2, 5, 7, 8, 12, 13, 14, 15, 19, 21, 22], most literature employs a rather simple model of RCPSP, whose objective is to minimize the makespan, and whose constraints are only those of renewable resources and precedence constraints. In real situations, however, more complicated constraints often arise, which cannot be formulated in such a simple form of RCPSP.

*to appear in: C. Ribeiro and P. Hansen (eds.), Essays and Surveys in Metaheuristics, Kluwer Academic Publishers.

Our objective in this paper is to develop a general purpose RCPSP algorithm that can handle much wider range of scheduling problems. Some generalization of RCPSP have already been discussed in references such as [4, 9]. Our generalization consists in that each activity can be processed in one of the selectable modes (RCPSP with this generalization is called the multi-mode RCPSP [4, 9, 15]), nonrenewable resources can be included in addition to renewable resources, the available amounts of renewable resources may vary depending on the periods, and setup activities can also be dealt with. A wider class of objective functions, other than the makespan, can be included as soft constraints of our model. For this generalized formulation of RCPSP, we develop a tabu search algorithm. This code was applied to many benchmarks of RCPSP, and improved the best known solutions for a number of problem instances. Some problems from real applications were also solved to demonstrate that our generalization of RCPSP is essential to handle practical problems. We report that rather good solutions could be obtained to these instances.

Of course, general purpose algorithms like ours may not be efficient enough if compared with special purpose algorithms tailored to special cases of RCPSP (e.g., jobshop scheduling). But we believe that the generality is very important in order to be accepted by people in practical applications, since it can save a great deal of manpower, which is otherwise necessary to develop special purpose codes.

2 Formulation of RCPSP

We explain ingredients of RCPSP separately in the following subsections.

2.1 Resources and activities

Let R denote a set of resources, and J denote a set of activities. The set R is divided into the set of renewable resources R^{re} and the set of nonrenewable resources R^{non}. Renewable resources are those whose available amounts are restricted only on a period basis, and are independent of other periods, while nonrenewable resources are limited in total throughout the schedule horizon without any restriction in each period. For example, manpower and machines are regarded as renewable resources, while raw materials and budgets are typical examples of nonrenewable resources. Each activity $j \in J$ must be processed in one mode m_j chosen from a set M_j. For each mode $m_j \in M_j$, its processing time p_{m_j}, the set of required resources $R_{m_j} = R^{re}_{m_j} \cup R^{non}_{m_j}$ and their required amounts are prespecified. Once the processing of an activity j is initiated in mode m_j, it is processed in the consecutive p_{m_j} periods to finish. A schedule is represented by $(\boldsymbol{m}, \boldsymbol{s}) = ((m_j \mid j \in J), (s_j \mid j \in J))$, where m_j is the mode in which activity j is processed, and s_j is the *start time* of j (the *completion time* of j is then given by $c_j = s_j + p_{m_j}$). The values of p_{m_j}, s_j and c_j are all assumed to be non-negative integers, in this paper.

2.2 Resource constraints

Resource constraints are classified into the following two types.

(i) Renewable resource constraints: For each renewable resource $r \in R^{re}$, in each period $(t-1, t]$ $(t = 1, 2, \ldots)$, the total requirement of r by all activities must not exceed the available amount $K_{r,t}$. Here, unlike the standard assumption frequently used, the available amounts $K_{r,t}$ can vary depending on the periods $(t-1, t]$, and the amounts of r required during the processing periods of an activity may not be uniform.

Machines that can process only one activity at a time are special renewable resources that are often encountered in real scheduling problems. In the RCPSP models, a machine is regarded as a renewable resource r with unit of available amount in every period. When an activity is processed on the machine, it consumes unit amount in each period of processing.

(ii) Nonrenewable resource constraints: For each nonrenewable resource $r \in R^{non}$, its total requirement in the entire schedule horizon must be within the prespecified available amount. The required amount of r by an activity depends on only its mode, and is independent of its start time. Thus, these constraints prohibit some combinations of modes; if all activities have only one mode, the nonrenewable resource constraint is redundant.

2.3 Precedence constraints

Two types of precedence constraints are considered in our formulation of RCPSP.

(i) A *precedence constraint* $j_1 \prec j_2$ between activities, which says that activity j_2 cannot be started unless activity j_1 is finished, i.e.,

$$c_{j_1} \leq s_{j_2}. \tag{1}$$

(ii) An *immediate precedence constraint* $j_1 \prec\!\!\prec_r j_2$ on a renewable resource $r \in R^{re}$ (i.e., j_1 in mode m_{j_1} immediately precedes j_2 in mode m_{j_2} if both activities use $r \in R^{re}_{m_{j_1}} \cap R^{re}_{m_{j_2}}$), which is described by

$$c_{j_1} \leq s_{j_2} \quad \text{and} \quad s_{j_2} \leq s_{j'} \text{ for all } j' \text{ such that } r \in R^{re}_{m_{j'}} \text{ and } c_{j_1} \leq s_{j'}. \tag{2}$$

These immediate precedence constraints are restricted versions of the precedence constraint (1), but are useful to describe some complicated constraints which appear in the following situations.

(i) The setup activity $\sigma(j)$ for activity j must be processed just before j. This constraint can be described by an immediate precedence constraint. Setup activities will be discussed in more details in Section 2.4.

(ii) Two activities j_1 and j_2 must be initiated at the same time. This constraint can be described by introducing a fictitious activity j' and a fictitious renewable resource r which is consumed by j_1, j_2 and j', and adding two immediate precedence constraints $j' \prec\!\!\prec_r j_1$ and $j' \prec\!\!\prec_r j_2$.

(iii) The processings of two activities j_1 and j_2 must overlap. This constraint can be described by two immediate precedence constraints $j_1 \prec\!\!\prec_r j'$ and $j_2 \prec\!\!\prec_r j'$ for the fictitious activity j' as in 2.

(iv) The processing of j_2 must be started within q periods after j_1 is completed (i.e., $s_{j_2} \leq c_{j_1} + q$ holds), where $j_1 \prec j_2$ is assumed. This is described by introducing a fictitious activity j' whose processing time is $q+2$ and which overlaps both of j_1 and j_2. In particular, the constraint that no idle time is permitted between j_1 and j_2 can be represented by setting $q = 0$.

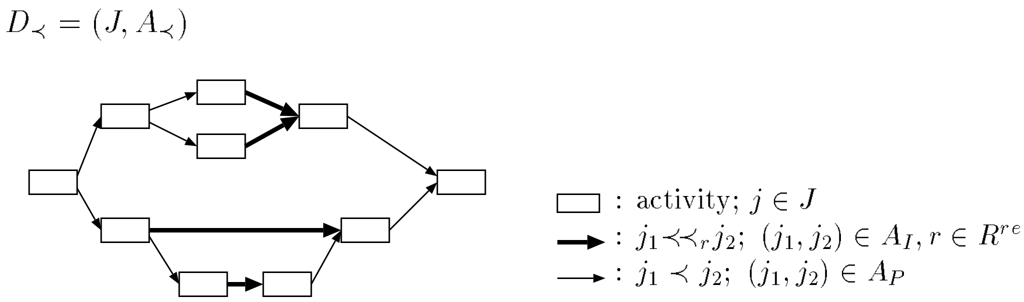


Figure 1: An example of $D_\prec = (J, A_\prec)$.

As a generalization of the precedence constraint, the temporal constraint is often introduced (e.g., [4, 9]), which specifies minimum and maximum start-start time lags between activities and can be described by

$$s_{j_2} - s_{j_1} \geq \delta_{j_1 j_2} \tag{3}$$

for an appropriate constant $\delta_{j_1 j_2}$ (which can be either positive or negative). Similarly, temporal constraints for completion-start, start-completion and completion-completion time lags are allowed. These temporal constraints can handle the above situations (ii), (iii) and (iv); however, they cannot represent the immediate precedence relations between activities and their setup activities. On the other hand, constraints of type (3) can be represented by using immediate precedence constraints (though we need to introduce the associated dummy resources and activities).

Here, we define a digraph $D_\prec = (J, A_\prec)$, which is often referred to as an *activity-on-nodes network*, a *project network* or a *precedence graph*. J is the set of activities, and $A_\prec$ is given by $A_\prec = A_I \cup A_P$, where

$$A_I = \{(j_1, j_2) \mid \exists r, j_1 \prec\!\!\prec_r j_2\}, \text{ and}$$

$$A_P = \{(j_1, j_2) \mid j_1 \prec j_2\} \setminus A_I.$$

An example of $D_\prec$ is illustrated in Figure 1. We assume that $D_\prec$ is acyclic, since otherwise there exists no schedule satisfying all the precedence constraints (unless all activities on all cycles have zero processing time).

2.4 Setup activities

In many problems in practice, the setup operation $setup(j_1, j_2)$ becomes necessary between changing from activity j_1 to j_2 on a machine r. While processing the setup operation, no activity can be processed on r. Although setup operations are very important in real applications, the research on this aspect has been rather limited. In our formulation, such setup operations can be handled as follows.

We introduce the setup activity $\sigma(j_2)$, which is required for the processing of activity j_2, together with an immediate precedence constraint $\sigma(j_2) \prec\!\!\prec_r j_2$. The activity $\sigma(j_2)$ is also processed on machine r and its mode is determined according to the activity $prev(\sigma(j_2))$ that immediately precedes $\sigma(j_2)$ on machine r. In our algorithm, we consider only those schedules in which all setup activities are processed in these specified modes.

In the subsequent discussion, setup activities $\sigma(j)$ will be also regarded as standard activities in J, unless otherwise specified.

2.5 Hard and soft constraints, and objective functions

In our formulation, not only resources and precedence constraints, but also any other constraints imposed on modes m_j, start times s_j, completion times c_j and/or processing times p_j of activities j can be included. In our current implementation, only linear inequality constraints in the form of

$$\sum_{j \in J} \sum_{m_j \in M_j} \alpha_{j,m_j} x_{j,m_j} + \sum_{j \in J} \beta_j s_j \leq \gamma \tag{4}$$

are accepted as additional constraints, where α_{j,m_j}, β_j and γ are constants, and x_{j,m_j} is 0-1 variable that takes value 1 (resp., 0) if activity j is (resp., is not) processed in mode m_j. Note that constraint (4) may include processing times p_j and completion times c_j of activities j, since p_j and c_j are represented as $\sum_{m_j} p_{m_j} x_{j,m_j}$ and $\sum_{m_j} p_{m_j} x_{j,m_j} + s_j$, respectively.

The constraints are classified into *hard* constraints and *soft* constraints, meaning that the former must always be satisfied for the obtained schedules to be practically meaningful, while the latter may remain violated if it is difficult to satisfy them. For each soft constraint C_ℓ, a *penalty function* $p_\ell(\boldsymbol{m}, \boldsymbol{s})$ and a *weight* $w_\ell > 0$ are introduced, where $p_\ell(\boldsymbol{m}, \boldsymbol{s})$ takes 0 (resp., positive) value if C_ℓ is satisfied (resp., violated), and w_ℓ is associated to represent the importance of C_ℓ. Then, we try to find an optimal schedule in the sense of minimizing the sum of weighted penalties

$$p(\boldsymbol{m}, \boldsymbol{s}) = \sum_{\ell} w_\ell p_\ell(\boldsymbol{m}, \boldsymbol{s}) \tag{5}$$

subject to all hard constraints. In other words, the quality of a schedule is evaluated by the amounts of violations of these soft constraints. A schedule satisfying all hard constraints is called a *feasible schedule*.

In our algorithm, only the constraints on renewable resources and precedence relations can be specified as hard, although, if necessary, some of them may be specified as soft by putting them in the category of (4). Therefore, the constraints of type (4) are always considered as soft. If the weights are set to substantially large numbers, however, the corresponding constraints behave like hard constraints even if their feasibility is not always maintained. The constraints on nonrenewable resources are also specified in the form of linear inequalities (4); i.e., they are considered to be soft. However, all the immediate precedence constraints are regarded as hard, and the constraints on the modes of setup activities are also hard, as mentioned in Section 2.4.

Except for the above default rules, it is users' responsibility to specify which constraints are soft, and how much weights should be associated with them. Although these decisions must be carefully done so that acceptable schedules may be obtained, this flexibility permits a wide applicability of our formulation.

In concluding this section, we note that the objective function is optimized in the process of minimizing (5). For example, makespan can be minimized by introducing the following soft constraint for the activity *sink*, which succeeds all others by precedence constraints and whose processing time is 0,

$$c_{sink} \leq 0. \tag{6}$$

Then minimizing the penalty of this constraint (which is simply the value of c_{sink} in this case) enables us to minimize the makespan.

In real applications, in addition to makespan, we may encounter various objective functions, related to other parameters such as earliness and tardiness. For example, the minimization of the earliness and tardiness cost of an activity can be achieved by using the following soft constraint on a renewable resource. Let d_j be the due date of activity j, and $cost\_due(c_j, d_j)$ be a given cost function. We introduce a renewable resource $r_{\mathrm{due}}(j)$, which is consumed only in the last period of processing j by amount $k = \max_t cost\_due(t, d_j)$, and whose available amounts $K_{r_{\mathrm{due}}(j),t}$ are defined as $k - cost\_due(t, d_j)$ $(t = 1, 2, \ldots)$. If j is completed at time t, then its penalty is given by the excess of $r_{\mathrm{due}}(j)$, that is, $k - K_{r_{\mathrm{due}}(j),t} = cost\_due(t, d_j)$, and hence represents the earliness or tardiness cost $cost\_due(c_j, d_j)$. In this way, we can handle any cost function $cost\_due(c_j, d_j)$ defined by c_j and d_j.

3 Tabu search for RCPSP

There have been proposed many heuristic algorithms for RCPSP [1, 2, 5, 7, 8, 10, 12, 13, 14, 15, 18, 19, 21, 22]. Among them, some effective RCPSP algorithms (e.g., genetic algorithms by Hartmann [7, 8], an ant colony optimization algorithm by Merkle et al. [13] and a tabu search algorithm by Baar et al. [1]) represent a solution by a permutation of all the activities (which is often called a *list*), from which a schedule is generated by the serial or parallel schedule generation scheme [10]. The list representation is also used in our algorithms. The algorithms [1, 7, 8, 13] are developed for the standard RCPSP, that is, only constraints on renewable resources and precedence relations are considered, the available amounts of resources do not vary with periods, and the objective is to minimize makespan. In [19], Sampson and Weiss considered a generalization of RCPSP such that any objective function and constraints can be handled, and proposed a local search algorithm. In their algorithm, a solution is represented by a *shift vector*, each element of which gives the start time lag of an activity from the maximum completion time of its predecessors. A schedule constructed from a shift vector always satisfies the precedence constraints, but may violate other constraints. The feasibility is sought by minimizing the penalty function like (5). More recently, Möhring et al. have developed a Lagrangian-based heuristic algorithm [14] for the time-indexed integer programming formulation. Their approach can be easily extended to general RCPSPs with other objective functions than the makespan, and with time-dependent resource availabilities and requirements.

Our algorithm is developed for a generalized formulation of RCPSP, and is based on tabu search. All the schedules considered in our algorithm are generated from lists (i.e., permutations of activities), as in [1, 7, 8, 13], but require more careful consideration since we handle more general constraints, which are classified into hard and soft.

Tabu search [6] is a local search algorithm with a modification that the search continues to the best neighboring solution even if local optimum is attained. In tabu search, to avoid cycling of solutions, a set of solutions including those recently visited are kept in *tabu list*, and excluded from the candidate list of the next solution.

In this section, we first define basic components of local search such as search space and neighborhood, and then describe how we implement our tabu search.

3.1 Representation of solutions

As mentioned in Section 2, a schedule can be represented by a vector $(\boldsymbol{m}, \boldsymbol{s})$. However, this representation may not give rise to an efficient search algorithm, since changing the start time s_j or the mode m_j of one activity j (but not other variables) may cause a lot of constraint violations which are not straightforward to adjust. To avoid this complication, we represent a solution by a list $\boldsymbol{\pi} = (\pi(1), \pi(2), \ldots, \pi(|J|))$ (a permutation of all activities) and a vector of mode selections $\boldsymbol{m} = (m_j \mid j \in J)$ (in [1, 7], a solution is represented only by $\boldsymbol{\pi}$, as single-mode RCPSP is considered). The corresponding schedule is then constructed by algorithm CONSTRUCT whenever needed. In other words, CONSTRUCT is a mapping from the space of $(\boldsymbol{m}, \boldsymbol{\pi})$ into the space of schedules. In the subsequent discussion, we sometimes do not distinguish a solution $(\boldsymbol{m}, \boldsymbol{\pi})$ from its schedule constructed by CONSTRUCT.

The algorithm CONSTRUCT tries to construct a feasible (i.e., all hard constraints are satisfied) schedule from a given $(\boldsymbol{m}, \boldsymbol{\pi})$. Its algorithmic details will be explained later in this section. However, it should be noted that finding a feasible schedule may not be easy if we allow immediate precedence constraints. In fact, if immediate precedence constraints are allowed, the problem of determining the existence of a feasible schedule is NP-hard even if availability and requirement of every renewable resource do not vary with periods. In our implementation of CONSTRUCT, a program parameter T is introduced, which is a prespecified upper bound on the start times s_i of activities i; i.e., only those schedules satisfying $s_i \leq T$ can be constructed in CONSTRUCT. In the initial part of tabu search, we generate solutions $(\boldsymbol{m}, \boldsymbol{\pi})$ randomly and apply CONSTRUCT to find feasible schedules. If no feasible schedule is found after trying a certain number of random solutions, we simply conclude the failure. In practice, however, such situation rarely occurs (if a sufficiently large value is specified as T), and even if so, we are usually able to relax some of the hard constraints into soft ones so that meaningful results may still be attained.

As explained in Section 2.5, some soft constraints may be given large weights and are expected to behave like hard constraints. In this case, as they are distinguished only by their penalty weights w_ℓ, the border between hard and soft constraints becomes somewhat fuzzy. However, we found in practical applications that distinguishing two types of constraints is usually very easy. From the viewpoint of constructing an appropriate model of the problem at hand, this distinction appears to be very important.

Now we describe algorithm CONSTRUCT to construct a feasible schedule from a given solution $(\boldsymbol{m}, \boldsymbol{\pi})$. For simplicity of description, we first assume that there are no setup activities (how to handle setup activities will be described at the end of this subsection). In CONSTRUCT, activities are sequentially scheduled in the order of $\boldsymbol{\pi}$ by setting the start time $s_{\pi(i)}$ of each $\pi(i)$ to the earliest time t ($\leq T$) such that the feasibility is met within the activities scheduled so far. As described in Step 2, this t is temporarily set to the minimum value obtained by considering all the activities preceding $\pi(i)$ (by precedence and immediate precedence constraints) and renewable resource requirements. Then, if t is greater than the prespecified T, CONSTRUCT returns "failure"; otherwise the activity $\pi(i)$ is scheduled at $s_{\pi(i)} = t$. Note that the obtained schedule may not be feasible since it may not satisfy immediate precedence constraints (even if $t \geq c_j$ holds for the immediately

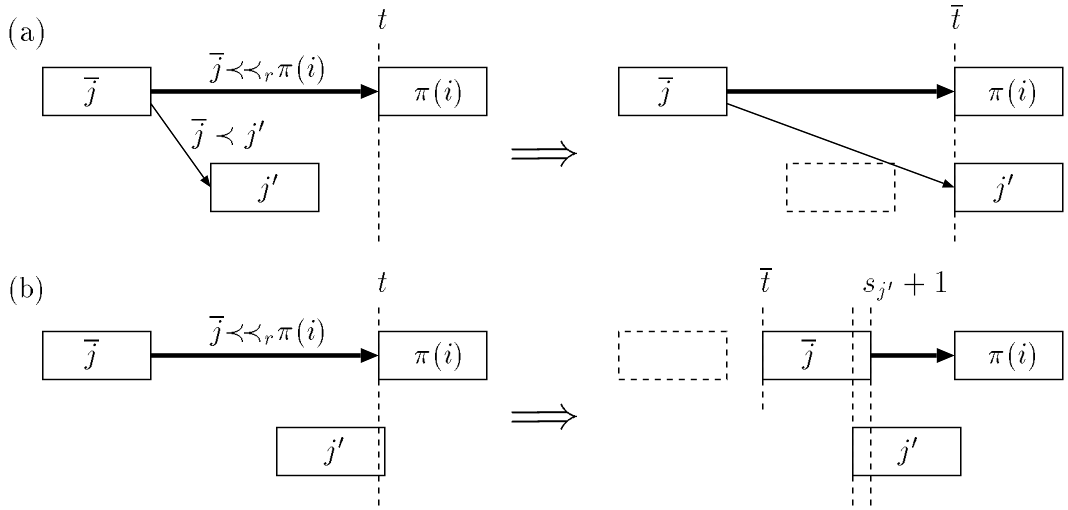


Figure 2: Examples of backtracking in Step 3, where immediate precedence constraint $\bar{j} \prec\!\prec_r \pi(i)$ (bold arrow) is violated: (a) if $\bar{j} \prec j'$, then j' will be delayed; (b) otherwise, $\bar{j}$ will be delayed.

preceding activities j). If the schedule is feasible, we move to the next activity $\pi(i+1)$. If it is not feasible, however, there are an immediate precedence constraint $\bar{j} \prec\!\prec_r \pi(i)$, and an activity j', such that j' has already been scheduled, its start time $s_{j'}$ satisfies $c_{\bar{j}} \le s_{j'} < t$, and it prevents $\pi(i)$ from being scheduled at time t. The situation is illustrated in Figure 2 (a) and (b), respectively. In these cases, the constraint $\bar{j} \prec\!\prec_r \pi(i)$ cannot be met by delaying the start time of $\pi(i)$, and a backtracking takes place (in Step 3) to delay the start time of activity j' or $\bar{j}$. The results are also illustrated in Figure 2 (a) and (b). In case (b), it is also possible to resolve the violation of $\bar{j} \prec\!\prec_r \pi(i)$ by delaying the start time of j' till that of $\pi(i)$ as in case (a). However, we do not adopt this, because when such j' is not unique, delaying $\bar{j}$ seems more effective to reduce the number of such j'.

In our search, we only generate $(\boldsymbol{m}, \boldsymbol{\pi})$ for which

$$\pi^{-1}(j_1) < \pi^{-1}(j_2) \qquad \text{for all pairs } j_1 \prec j_2, \tag{7}$$

holds, where $i = \pi^{-1}(j)$ is defined by $\pi(i) = j$. In other words, no activity is scheduled until all of its predecessors are scheduled. This property will facilitate and simplify some part of the algorithm.

CONSTRUCT

Input: A solution $(\boldsymbol{m}, \boldsymbol{\pi})$.

Output: A feasible schedule $(\boldsymbol{m}, \boldsymbol{s})$ or "failure" (in the latter case, $(\boldsymbol{m}, \boldsymbol{\pi})$ is regarded to have the penalty of infinity).

Step 0 (initialization): $c_j := 0$ $(j \in J)$, and $i := 1$.

Step 1 (lower bound of the start time of $\pi(i)$): $\bar{t} := \max_{j \prec \pi(i)} c_j$ ($\bar{t} := 0$ if there is no j with $j \prec \pi(i)$).

Step 2 (checking the feasibility of $\pi(i)$): Calculate the minimum t ($\leq T$) that satisfies the following three conditions (t is the start time of activity $\pi(i)$):

1. $t \geq \bar{t}$.
2. All hard renewable resource constraints are satisfied for activities $\pi(1)$, $\pi(2), \ldots, \pi(i)$.
3. For any immediate precedence constraint $j_1 \prec\!\prec_r j_2$ such that $\pi^{-1}(j_1)$, $\pi^{-1}(j_2) < i$ and $r \in R^{re}_{m_{j_1}} \cap R^{re}_{m_{j_2}} \cap R^{re}_{m_{\pi(i)}}$, it holds $t < c_{j_1}$ or $t \geq s_{j_2}$ (i.e., $\pi(i)$ does not interfere with $j_1 \prec\!\prec_r j_2$).

If there exists no such t, return "failure". Otherwise (i.e., such t exists), check whether there are an immediate precedence constraint $\bar{j} \prec\!\prec_r \pi(i)$ and an activity j' such that $\pi^{-1}(j') < i$, $r \in R^{re}_{m_{\bar{j}}} \cap R^{re}_{m_{j'}} \cap R^{re}_{m_{\pi(i)}}$ and $c_{\bar{j}} \leq s_{j'} < t$ (see the left hand side of Figure 2 (a) and (b)). If yes, then go to Step 3; otherwise go to Step 4.

Step 3 (backtracking): The start time of j' or $\bar{j}$ is delayed.

(a) If $\bar{j} \prec j'$, then let $i := \pi^{-1}(j')$ and $\bar{t} := t$ (this is illustrated in Figure 2 (a)).

(b) Otherwise, let $i := \pi^{-1}(\bar{j})$ and $\bar{t} := s_{j'} + 1 - p_{m_{\bar{j}}}$ (this is illustrated in Figure 2 (b)).

In either case, return to Step 2, and reschedule this new $\pi(i)$ and all the activities after $\pi(i)$ in the list, with the updated $\bar{t}$.

Step 4 (setting the start time of $\pi(i)$): Let $s_{\pi(i)} := t$. If $i < |J|$, then $i := i + 1$ and return to Step 1; otherwise halt after outputting the constructed schedule $(\boldsymbol{m}, \boldsymbol{s})$.

If there is no immediate precedence constraint, no backtracking occurs in CONSTRUCT. In general, CONSTRUCT returns a feasible schedule or "failure" in finite steps, since the start time of the new $\pi(i)$ ($= j'$ or $\bar{j}$) is increased after backtracking in Step 3, while those of activities $\pi(i')$ remain the same for all $i' < i$. It is easily shown that CONSTRUCT can be executed in $O(|J|^2|R^{re}|)$ time if restricted to the standard RCPSP.

Now let $\mathcal{U}$ be the set of strongly connected components in digraph $D^*_{\prec} = (J, A^*_{\prec})$, which is obtained from $D_{\prec} = (J, A_{\prec})$ (defined in Section 2.3) by adding arc (j_2, j_1) for each arc $(j_1, j_2) \in A_I$ (see Figure 3). A precedence relation $\prec$ on $\mathcal{U}$ is defined by

$$U_1 \prec U_2 \Leftrightarrow U_1 \neq U_2 \text{ and } \exists j_1 \in U_1, \exists j_2 \in U_2 \text{ such that } j_2 \text{ is reachable from } j_1 \text{ in } D^*_{\prec}.$$

In order to suppress the number of activities rescheduled after backtracking in Step 3, we generate in our algorithm only those $\boldsymbol{\pi}$ that satisfy the following condition:

$$\text{All activities in one component } U \in \mathcal{U} \text{ are consecutively ordered in } \boldsymbol{\pi}. \tag{8}$$

If $\boldsymbol{\pi}$ satisfies conditions (7) and (8), backtracking in Step 3 occurs only within each component (i.e., once all activities in a component U are scheduled, their start times are fixed in the rest of computation of CONSTRUCT). This will facilitate the computation of CONSTRUCT substantially.

To process setup activities $\sigma(j)$ in the modes specified by its immediate predecessor $prev(\sigma(j))$ on the machine $r(\sigma(j))$, the procedure CONSTRUCT is modified as follows.

- Regardless of the mode $m_{\sigma(j)}$ of the input $(\boldsymbol{m}, \boldsymbol{\pi})$, the mode of a setup activity $\sigma(j)$ is determined in Step 2 by the activity $prev(\sigma(j))$ and the machine resource $r(\sigma(j))$.

- Once $\sigma(j)$ is scheduled, the immediate precedence constraint

$$prev(\sigma(j)) \prec\!\!\prec_{r(\sigma(j))} \sigma(j) \tag{9}$$

is introduced so that no activity between $prev(\sigma(j))$ and $\sigma(j)$ on $r(\sigma(j))$ is permitted. In case the start time of $\sigma(j)$ is canceled by backtracking in Step 3, constraint (9) is also removed.

3.2 Search space of tabu search

Our tabu search operates on the following search space:

$$\Lambda = \Big\{(\boldsymbol{m}, \boldsymbol{\pi}) \mid m_j \in M_j \text{ for all } j \in J,\ \boldsymbol{\pi} \in \Pi\Big\}, \tag{10}$$

where

$$\Pi = \{\boldsymbol{\pi} \mid \boldsymbol{\pi} \text{ satisfies (7) and (8)}\}. \tag{11}$$

The quality of a solution $(\boldsymbol{m}, \boldsymbol{\pi})$ is evaluated by the penalty to the schedule $(\boldsymbol{m}, \boldsymbol{s})$ constructed from $(\boldsymbol{m}, \boldsymbol{\pi})$ by CONSTRUCT.

A feasible schedule is called *active*, if no activity can be made start earlier (while keeping others the same) without violating the feasibility [20]. When there are no immediate precedence constrains, the schedules obtained by CONSTRUCT are always active. However, there may exist active schedules that cannot be constructed by CONSTRUCT. To see this, we consider an example of two activities j_1 and j_2 that consume a renewable resource r as illustrated in Figure 4. The resource r is available by amount $K_{r,t} = 3$ in all periods $(t-1, t]$, except that we have $K_{r,2} = 1$. Activity j_1 has processing time 4, and consumes r by amounts 3, 1, 1 and 3 in each period of the processing, respectively, while the processing time of activity j_2 is 2, and the requirement of r by j_2 is 2 in all processing periods. The renewable resource constraint r is hard, and there is no precedence constraint between j_1 and j_2. Then we can see that the active schedule of Figure 4 (a) cannot be constructed

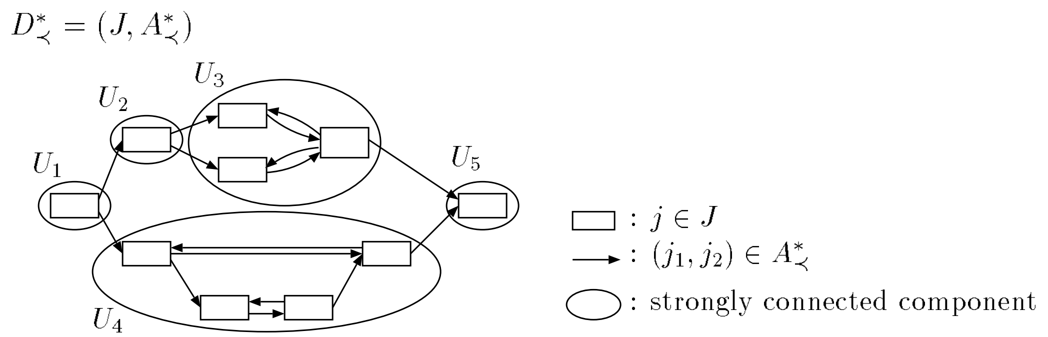


Figure 3: The digraph $D^*_\prec = (J, A^*_\prec)$ obtained from $D_\prec = (J, A_\prec)$ of Figure 1.

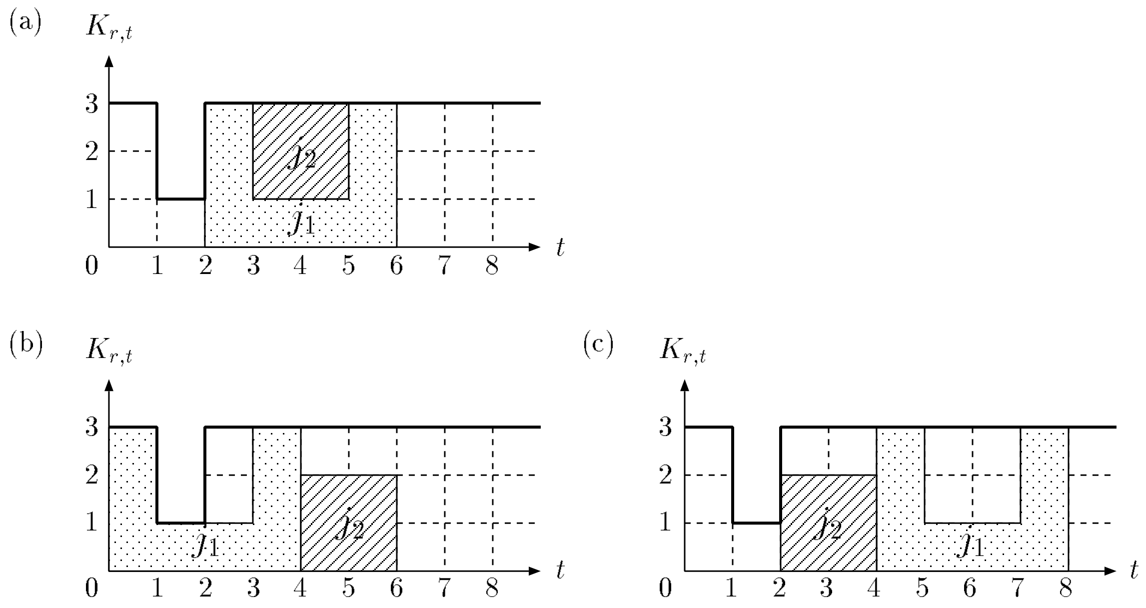


Figure 4: Examples of active schedules. The schedule (a) cannot be constructed by CONSTRUCT, while schedules (b) and (c) are constructed from lists (j_1, j_2) and (j_2, j_1), respectively.

by CONSTRUCT for any list. In fact, if a list $\boldsymbol{\pi}$ is (j_1, j_2) (resp., (j_2, j_1)), then the constructed schedule is Figure 4 (b) (resp., (c)), since each activity is sequentially scheduled at the earliest start time.

This observation tells that the set of schedules constructed from search space (10) by CONSTRUCT may not contain any optimal schedule $(\boldsymbol{m}^*, \boldsymbol{s}^*)$ of the given problem instance, if the constraints are complicated. Furthermore, the optimal schedule is not necessarily active in general, since the objective function represented by soft constraints (as explained in Section 2.5) may not be *regular* [20] in our RCPSP model. However, if the availabilities and requirements of all renewable resources do not vary with periods, and there is no immediate precedence constraint (as in the standard RCPSP), then it can be shown that any active schedule $(\boldsymbol{m}, \boldsymbol{s})$ can be constructed by CONSTRUCT from the solution $(\boldsymbol{m}, \boldsymbol{\pi}) \in \Lambda$ such that all activities j are listed in $\boldsymbol{\pi}$ in the nondecreasing order of their start times s_j. In this case, for any regular objective function, our search space contains an optimal solution that provides an optimal schedule.

To guarantee the existence of optimal schedules for all instances of our generalized RCPSP, it seems necessary to use the direct representations $(\boldsymbol{m}, \boldsymbol{s})$ as solutions to search. However, the local search over this search space appears less efficient, as mentioned in the beginning of Section 3.1, and we face the trade-off between completeness and efficiency. As we shall see in Section 4, our algorithm, relying on the search space Λ of (10), turns out to be reasonably powerful, even though the search space may not contain optimal solutions for the full generality of RCPSP.

3.3 Neighborhood

We define the neighborhood $N(\boldsymbol{m}, \boldsymbol{\pi})$ of the current solution $(\boldsymbol{m}, \boldsymbol{\pi})$ as the set of solutions obtained from $(\boldsymbol{m}, \boldsymbol{\pi})$ by applying one of the following three types of moves:

change_mod(j, m'_j), where $j \in J$ is a non-setup activity and $m'_j \in M_j$ satisfies $m'_j \neq m_j$ (where m_j is the current mode of j): Modify $\boldsymbol{m}$ by changing the mode m_j of j into another mode m'_j. For a setup activity $\sigma(j)$, this move is not defined, since its mode is uniquely determined in CONSTRUCT from its immediate predecessor $prev(\sigma(j))$.

$shift\_aft(j_1, j_2)$, where j_1 $(= \pi(i_1))$ and j_2 $(= \pi(i_2))$, which belong to strongly connected components U_1 and U_2 in digraph $D^*_\prec$ $(U_1, U_2 \in \mathcal{U})$, respectively, satisfy

$$i_1 < i_2,\ j_1 \not\prec j_2, \text{ and } U_1 \not\prec U_2. \tag{12}$$

If $U_1 = U_2$, then shift j_1 to the position immediately after j_2, together with all the activities $j' = \pi(i')$ such that $i_1 < i' < i_2$ and $j_1 \prec j'$; otherwise (i.e., $U_1 \neq U_2$), shift all activities in U_1 to the positions immediately after the last activity in U_2, together with all activities $j' = \pi(i') \in U'$ such that $i_1 < i' < i_2$ and $U_1 \prec U'$. The relative positions among all the activities moved in this process are preserved. Notice that the resulting list $\boldsymbol{\pi}'$ also belongs to the Π of (11).

$shift\_bef(j_1, j_2)$, where j_1 $(= \pi(i_1) \in U_1)$ and j_2 $(= \pi(i_2) \in U_2)$ satisfy (12): Modify $\boldsymbol{\pi}$ by shifting j_2 to the position before j_1, together with some other activities, in a symmetric manner to $shift\_aft$ so that the resulting list also belongs to the Π of (11).

Since it appears too time consuming to generate all solutions in neighborhood $N(\boldsymbol{m}, \boldsymbol{\pi})$ in each iteration, we try only those moves which may lead to a reduction of the penalty (5). To do this, for each of the currently violated constraint C_ℓ, we first calculate the move set $MV_\ell(\boldsymbol{m}, \boldsymbol{\pi})$, which consists of the moves that temporarily reduce the penalty of C_ℓ (its precise definition will be given later). The whole set is then denoted as

$$MV(\boldsymbol{m}, \boldsymbol{\pi}) = \bigcup_{C_\ell \text{ is violated}} MV_\ell(\boldsymbol{m}, \boldsymbol{\pi}).$$

Let $\tilde{N}(\boldsymbol{m}, \boldsymbol{\pi})$ denote the set of solutions obtained by applying the moves in $MV(\boldsymbol{m}, \boldsymbol{\pi})$ to $(\boldsymbol{m}, \boldsymbol{\pi})$. The neighborhood $N(\boldsymbol{m}, \boldsymbol{\pi})$ is reduced to $\tilde{N}(\boldsymbol{m}, \boldsymbol{\pi})$. Since the size of $\tilde{N}(\boldsymbol{m}, \boldsymbol{\pi})$ depends on the number of violated constraints, $\tilde{N}(\boldsymbol{m}, \boldsymbol{\pi})$ may still be too large if there are many violated constraints. In such a case, we also try a strategy that decreases the size of neighborhood by randomly choosing a subset of moves from $MV(\boldsymbol{m}, \boldsymbol{\pi})$ (some computational results will be reported in Sections 4.2.1 and 4.2.2).

The definition of $MV_\ell(\boldsymbol{m}, \boldsymbol{\pi})$ depends on the type of constraint C_ℓ. We now describe how we constructed $MV_\ell(\boldsymbol{m}, \boldsymbol{\pi})$ in our implementation, but other choices may also be possible. Let us first define the digraph $G(\boldsymbol{m}, \boldsymbol{\pi}) = (J, A(\boldsymbol{m}, \boldsymbol{\pi}))$: The vertex set J is the set of activities, and arcs (j_1, j_2) in $A(\boldsymbol{m}, \boldsymbol{\pi})$ imply that j_1 is an obstacle to schedule j_2 before s_{j_2} in CONSTRUCT. In other words, if j_1 had not been scheduled before j_2, the start time of j_2 would be earlier than s_{j_2}. There are four possibilities for why j_2 could not be scheduled before s_{j_2};

a. there is a predecessor of j_2 whose completion time is s_{j_2},

b. if j_2 was scheduled at t' ($< s_{j_2}$), some hard renewable resource constraints would be violated,

c. if j_2 was scheduled at t' ($< s_{j_2}$), some immediate precedence constraint would be violated,

d. backtracking in Step 3 delayed the start time of j_2 to s_{j_2}.

The cases a–c correspond to the conditions 1–3 in Step 2 of CONSTRUCT, respectively. We define

$$A(\boldsymbol{m}, \boldsymbol{\pi}) = A_P(\boldsymbol{m}, \boldsymbol{\pi}) \cup A_R(\boldsymbol{m}, \boldsymbol{\pi}) \cup A_I(\boldsymbol{m}, \boldsymbol{\pi}) \cup A_B(\boldsymbol{m}, \boldsymbol{\pi}),$$

where A_P, A_R, A_I and A_B are given as follows, corresponding to the above four cases.

$$A_P(\boldsymbol{m}, \boldsymbol{\pi}) = \{(j_1, j_2) \mid j_1 \prec j_2 \text{ and } c_{j_1} = s_{j_2}\},$$

$$A_R(\boldsymbol{m}, \boldsymbol{\pi}) = \left\{ (j_1, j_2) \;\middle|\; \begin{array}{l} \text{For some } t' < s_{j_2},\ j_2 \text{ could not be started at } t' \text{ in CON-} \\ \text{STRUCT because of the lack of a resource which was also} \\ \text{consumed by } j_1. \end{array} \right\},$$

$$A_I(\boldsymbol{m}, \boldsymbol{\pi}) = \left\{ (j_1, j_2) \;\middle|\; \begin{array}{l} \text{For some } t' < s_{j_2} \text{ and } j \in J,\ j_2 \text{ could not be started at } t' \\ \text{in CONSTRUCT due to immediate precedence constraint} \\ j \prec\!\prec_r j_1 \text{ such that } c_j \le t' < s_{j_1}. \end{array} \right\},$$

and

$$A_B(\boldsymbol{m}, \boldsymbol{\pi}) = \left\{ (j_1, j_2) \;\middle|\; \begin{array}{l} \text{a backtracking was executed in Step 3 (a) of CONSTRUCT} \\ \text{with } \pi(i) = j_1 \text{ and } j' = j_2, \text{ or in Step 3 (b) with } j' = j_1 \text{ and} \\ \overline{j} = j_2. \end{array} \right\}.$$

In our code, for each solution $(\boldsymbol{m}, \boldsymbol{\pi})$, the digraph $G(\boldsymbol{m}, \boldsymbol{\pi})$ is constructed in the process of applying CONSTRUCT to $(\boldsymbol{m}, \boldsymbol{\pi})$.

In the tabu search algorithm developed for the standard RCPSP by Baar et al. [1], a digraph similar to our $G(\boldsymbol{m}, \boldsymbol{\pi})$ is used to define the neighborhood. But there are differences in the definition of the arc set, even if restricted to the standard RCPSP. The arc set of [1] is defined by $\{(j_1, j_2) \mid c_{j_1} = s_{j_2}\}$. Thus, even when $\pi^{-1}(j_1) > \pi^{-1}(j_2)$ holds (i.e., j_1 is scheduled later than j_2), the digraph of [1] has arc (j_1, j_2), if $c_{j_1} = s_{j_2}$ holds, while ours does not. As a result, our neighborhood appears to represent more directly the conflicts encountered in the process of scheduling activities.

Now, for a constraint C_ℓ currently violated by solution $(\boldsymbol{m}, \boldsymbol{\pi})$, we implement the following three types (i)(ii)(iii) of move sets $MV_\ell(\boldsymbol{m}, \boldsymbol{\pi})$ in our code, as these appear to be effective to handle the constraints of many existing problems. To handle other types of constraints, however, we may need to introduce different move sets; this is a topic of our future research.

(i) C_ℓ is a renewable resource constraint for $r \in R^{re}$ in period $(t-1, t]$. To reduce the penalty of C_ℓ, we try to change the modes or the start times of the activities j that consume r in $(t-1, t]$. Let J' denote the set of such activities j. For each of $j \in J'$, let $\boldsymbol{v}^-(j) = (v_1^-, v_2^-, \ldots, v_{n^-}^- = j)$ and $\boldsymbol{v}^+(j) = (j = v_1^+, v_2^+, \ldots, v_{n^+}^+)$ be two maximal simple directed paths to j and from j in $G(\boldsymbol{m}, \boldsymbol{\pi})$, respectively (if there is more than one candidate, choose arbitrarily), where we say that a simple directed path $\boldsymbol{v}$ is *maximal* if it is not contained in any other simple directed path to j and from j, respectively. Let m_v be the current mode of activity v. The move set is defined by

$$MV_\ell(\boldsymbol{m}, \boldsymbol{\pi}) = \bigcup_{j \in J'} CM(j) \cup SB(j) \cup SA(j), \tag{13}$$

where

$$CM(j) = \left\{ change\_mod(v, m'_v) \;\middle|\; \begin{array}{l} v \text{ is a non-setup activity in } \boldsymbol{v}^-(j), \\ m'_v \in M_v \text{ and } m'_v \neq m_v \end{array} \right\},$$

$$SB(j) = \{shift\_bef(v^-, j) \mid v^- \text{ is in } \boldsymbol{v}^-(j) \text{ and } (v^-, j) \text{ satisfies (12)}\},$$

and

$$SA(j) = \{shift\_aft(j, v^+) \mid v^+ \text{ is in } \boldsymbol{v}^+(j) \text{ and } (j, v^+) \text{ satisfies (12)}\}.$$

Here, for the activities in $\boldsymbol{v}^+(j)$, the moves *change_mod* are not considered, since they have less influence on the start time of j.

(ii) C_ℓ is a constraint related to the mode m_j of an activity j. C_ℓ is typically a non-renewable resource constraint. In general, it is described by a linear inequality (4) with $\alpha_{j,m_j} \neq 0$. In this case, we try to change the mode of j. If j is a setup activity, we try to change its mode by shifting its position in $\boldsymbol{\pi}$, as its mode is determined by its immediate predecessor (as described in Section 2.4). The move set is then given by

$$\begin{aligned} MV_\ell(\boldsymbol{m}, \boldsymbol{\pi}) \;&=\; \left\{ change\_mod(j, m'_j) \;\middle|\; \begin{array}{l} j \text{ is a non-setup activity,} \\ m'_j \in M_j \text{ and } \alpha_{j,m'_j} \neq \alpha_{j,m_j} \end{array} \right\} \\ &\cup \bigcup_{\sigma(j) \in J''} SB(\sigma(j)) \cup SA(\sigma(j)), \end{aligned} \tag{14}$$

where $J'' = \{j \mid j \text{ is a setup activity such that } \alpha_{j,m_j} \neq 0\}$, and move sets $SB(\sigma(j))$ and $SA(\sigma(j))$ are defined in the same manner as in case (i).

(iii) C_ℓ is a constraint given by a linear inequality (4), for which $\beta_j \neq 0$ holds for some activity j. If $\beta_j > 0$ holds, for example, the penalty of C_ℓ can be decreased by making an activity j start earlier (see e.g., (6)). As in cases (i) and (ii), maximal simple paths $\boldsymbol{v}^-(j)$ and $\boldsymbol{v}^+(j)$ are also utilized. Let J^+ and J^- denote $\{j \mid \beta_j > 0\}$ and $\{j \mid \beta_j < 0\}$, respectively. Then, the move set is defined by

$$\begin{aligned} MV_\ell(\boldsymbol{m}, \boldsymbol{\pi}) \;&=\; \bigcup_{j \in J^+} \left\{ shift\_bef(v_i, v_{i+1}) \;\middle|\; \begin{array}{l} (v_i, v_{i+1}) \text{ is an arc in } \boldsymbol{v}^-(j) \\ \text{and satisfies (12)} \end{array} \right\} \\ &\cup \bigcup_{j \in J^-} SA(j) \;\cup \bigcup_{j \in J^+ \cup J^-} CM(j). \end{aligned} \tag{15}$$

The moves of type *shift_bef* in the first set is different from $SB(j)$ of case (i). Although other definitions are also possible, this is chosen in our implementation, because in case (iii), move set $SB(j)$ sometimes becomes quite small and is not effective; e.g., if C_ℓ is (6) for the minimization of makespan, then $SB(j = sink)$ becomes empty.

In passing, it is noted that, if the problem being solved is a standard RCPSP (in which constraint (6) is employed), a maximal simple path $\boldsymbol{v}^-(sink) = (v_1^-, v_2^-, \ldots, v_{n^-}^- = sink)$

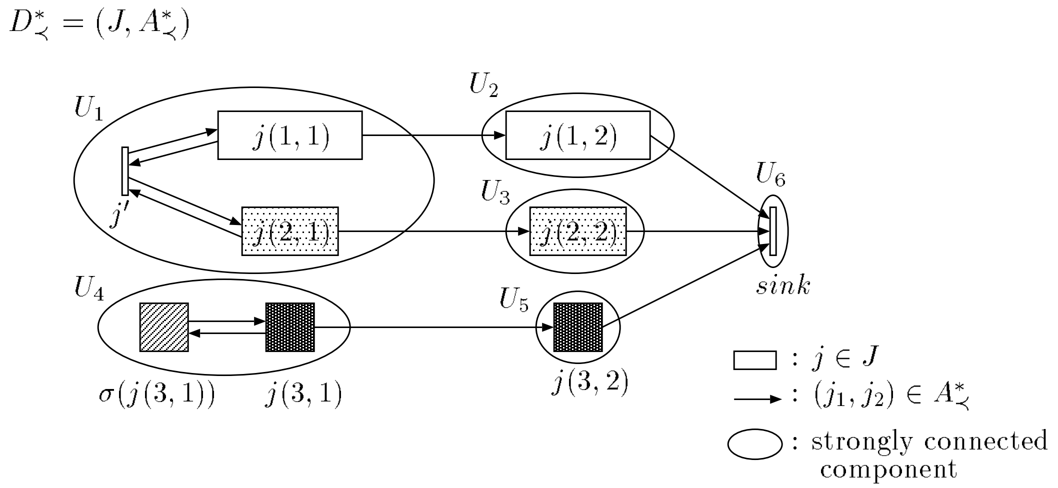


Figure 5: The digraph $D^*_\prec = (J, A^*_\prec)$ of the example.

chosen in (iii) to determine the move set is always a critical (i.e., longest) path, since it satisfies $s_{v_1^-} = 0$ and $c_{v_i^-} = s_{v_{i+1}^-}$ for all $i = 1, 2, \ldots, n^- - 1$. Since it is required to make such a critical path non-critical in order to decrease the makespan, the idea of eliminating a critical path is often used in algorithms for the jobshop scheduling problem [3], and also for the standard RCPSP (e.g., [1]). The moves *shift_bef* in (15) have similar effects, and, in such special cases, our neighborhood may be said to embody similar ideas.

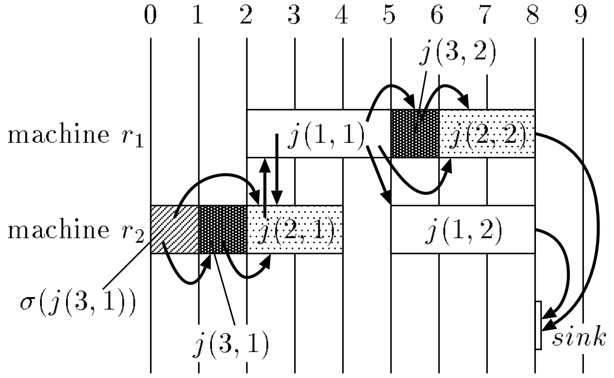
3.4 Example

We show an example to illustrate solutions $(\boldsymbol{m}, \boldsymbol{\pi})$, their schedules $(\boldsymbol{m}, \boldsymbol{s})$, digraph $G(\boldsymbol{m}, \boldsymbol{\pi})$ and how $MV_\ell(\boldsymbol{m}, \boldsymbol{\pi})$ are actually constructed.

Figure 5 gives an example consisting of two machines and three jobs. Each job i ($i = 1, 2, 3$) consists of two activities $j(i,1)$ and $j(i,2)$ with precedence constraint $j(i,1) \prec j(i,2)$. Activities $j(1,1)$, $j(2,2)$ and $j(3,2)$ are processed on machine 1, and the rest are processed on machine 2. The processing times of the activities in job i are defined to be $4 - i$. For simplicity, all activities have only one mode. In addition to the resource constraints of two machines, r_1 and r_2, and the above three precedence constraints, this problem has a constraint that $j(1,1)$ and $j(2,1)$ must start at the same time; hence a fictitious activity j' and a fictitious renewable resource r_0 are introduced, and two immediate precedence constraints $j' \prec\!\prec_{r_0} j(1,1)$ and $j' \prec\!\prec_{r_0} j(2,1)$ are added (see Section 2.3). Furthermore, to start the processing of $j(3,1)$, the setup operation of 1 period is required on machine r_2, unless $j(1,2)$ is processed just before $j(3,1)$ on machine r_2. As explained in Section 2.4, the setup activity $\sigma(j(3,1))$ is introduced together with its immediate precedence constraint $\sigma(j(3,1)) \prec\!\prec_{r_2} j(3,1)$. The mode set of $\sigma(j(3,1))$ is given by {dummy, setup}, where "dummy" has processing time 0, and "setup" is associated with the setup operation. Setup activity $\sigma(j(3,1))$ is in mode "dummy" (resp., "setup"), if $j(1,2)$ is (resp., is not) immediately preceding $\sigma(j(3,1))$ on machine r_2.

The digraph $D^*_\prec$ (defined in Section 3.1) of this problem is shown in Figure 5. Activi-

$$G(\boldsymbol{m}, \boldsymbol{\pi}_0 = (\sigma(j(3,1)), j(3,1), j', j(1,1), j(2,1), j(3,2), j(1,2), j(2,2), sink))$$



$$A_P = \left\{ \begin{array}{l} (\sigma(j(3,1)), j(3,1)), \\ (j(1,1), j(1,2)), \\ (j(1,2), sink), \\ (j(2,2), sink) \end{array} \right\}$$

$$A_R = \left\{ \begin{array}{l} (\sigma(j(3,1)), j(2,1)), \\ (j(3,1), j(2,1)), \\ (j(1,1), j(3,2)), \\ (j(1,1), j(2,2)), \\ (j(3,2), j(2,2)) \end{array} \right\}$$

$$A_I = \{(j(1,1), j(2,1))\}$$

$$A_B = \{(j(2,1), j(1,1))\}$$

Figure 6: The digraph $G(\boldsymbol{m}, \boldsymbol{\pi}_0) = (J, A(\boldsymbol{m}, \boldsymbol{\pi}_0))$.

ties $j', j(1,1)$ and $j(2,1)$, and activities $\sigma(j(3,1))$ and $j(3,1)$ belong to the same strongly connected components U_1 and U_4, respectively. Others belong to distinct components U_i, $i = 2, 3, 5, 6$, each consisting of a single activity.

For this problem, we now consider three soft constraints, respectively. For each of them, we explain how our algorithm determines the move sets, and reduces the penalties of these constraints.

Let us first assume that the current list is

$$\boldsymbol{\pi}_0 = (\sigma(j(3,1)), j(3,1), j', j(1,1), j(2,1), j(3,2), j(1,2), j(2,2), sink),$$

whose digraph $G(\boldsymbol{m}, \boldsymbol{\pi}_0)$ is given in Figure 6. Figure 6 may also be viewed as the Gantt chart of the schedule constructed from $\boldsymbol{\pi}_0$ by CONSTRUCT (in which fictitious activity j' is omitted). Notice that activities $\sigma(j(3,1))$ and $j(3,1)$, and activities $j', j(1,1)$ and $j(2,1)$ are contained in one strongly connected components, respectively, and hence consecutively ordered in $\boldsymbol{\pi}_0$. Hence $\boldsymbol{\pi}_0$ belongs to Π of (11). In the computation of CONSTRUCT, the mode of $\sigma(j(3,1))$ is set to "setup", as $j(1,2)$ is not processed just before $\sigma(j(3,1))$ on machine r_2. Activity $j(1,1)$ was first scheduled at time 0, but later delayed by the backtracking in Step 3 due to the violation of $j' \prec\!\prec_{r_0} j(2,1)$ (this fictitious activity j' is scheduled at time 0); thus $G(\boldsymbol{m}, \boldsymbol{\pi})$ has arc $(j(2,1), j(1,1)) \in A_B$.

First, we assume that there is a soft renewable resource constraint C_{ℓ_1} for $r' \in R^{re}$ and period $(5,6]$, where r' is consumed by only activity $j(3,2)$ and not available in period $(5,6]$ (but available in all other periods). As we can see from Figure 6, constraint C_{ℓ_1} is violated by the current solution $(\boldsymbol{m}, \boldsymbol{\pi}_0)$. In this case, move set (i) in Section 3.3 applies, and two simple paths $\boldsymbol{v}^-(j(3,2)) = (\sigma(j(3,1)), j(3,1), j(2,1), j(1,1), j(3,2))$ and $\boldsymbol{v}^+(j(3,2)) = (j(3,2), j(2,2), sink)$, which are respectively maximal, are chosen. Based on these, the move set (13) is determined as

$$\begin{array}{rcl} MV_{\ell_1}(\boldsymbol{m}, \boldsymbol{\pi}_0) & = & \{shift\_bef(j(2,1), j(3,2)),\ shift\_bef(j(1,1), j(3,2))\} \\ & \cup & \{shift\_aft(j(3,2), j(2,2))\}. \end{array}$$

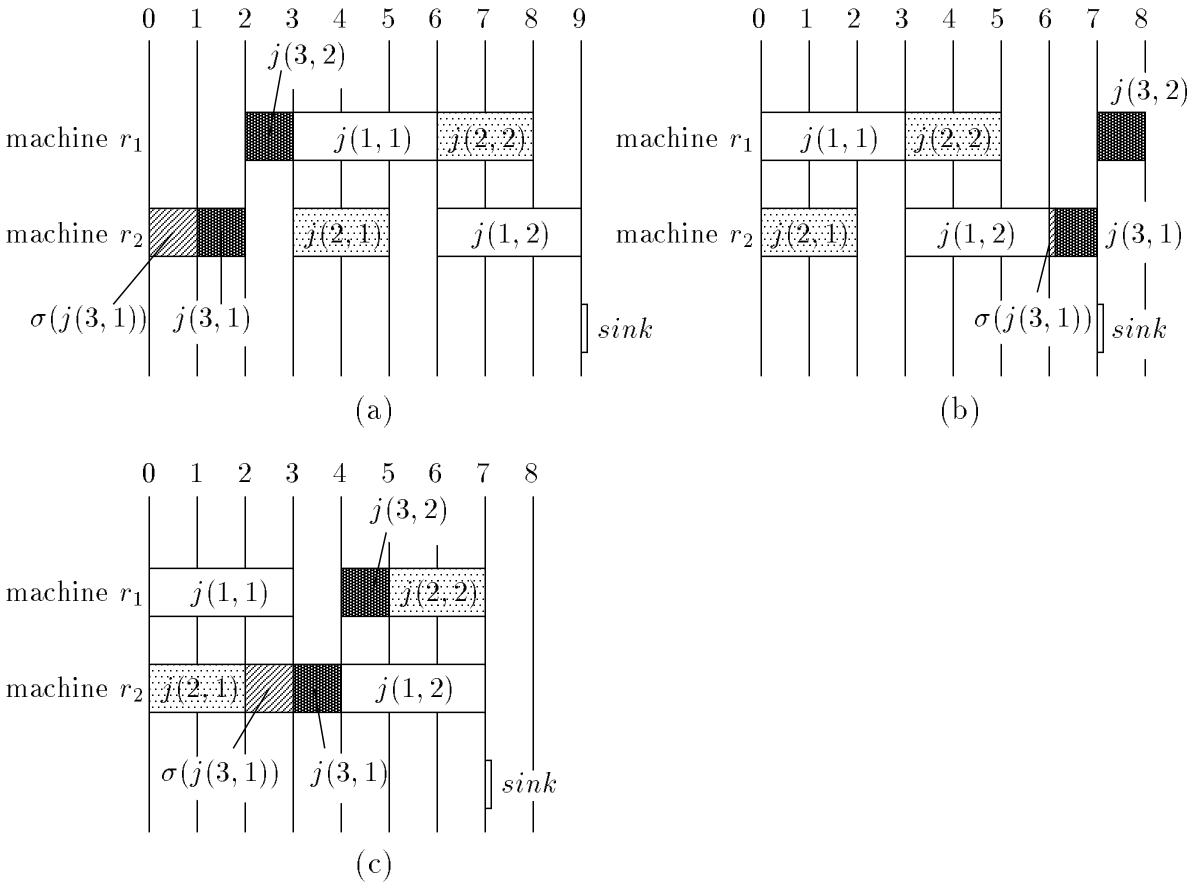


Figure 7: The schedules constructed from (a) $(\boldsymbol{m}, \boldsymbol{\pi}_1)$, (b) $(\boldsymbol{m}, \boldsymbol{\pi}_2)$ and (c) $(\boldsymbol{m}, \boldsymbol{\pi}_3)$ by CONSTRUCT.

Three moves $shift\_bef(\sigma(j(3,1)), j(3,2))$, $shift\_bef(j(3,1), j(3,2))$ and $shift\_aft(j(3,2), sink)$ are excluded because precedence constraints $\sigma(j(3,1)) \prec j(3,2)$, $j(3,1) \prec j(3,2)$ and $j(3,2) \prec sink$ violate (12).

As an example of move, let us apply $shift\_bef(j(2,1), j(3,2))$ to $\boldsymbol{\pi}_0$. The resulting list becomes

$$\boldsymbol{\pi}_1 = (\sigma(j(3,1)), j(3,1), j(3,2), j', j(1,1), j(2,1), j(1,2), j(2,2), sink).$$

Recall that activities j', $j(1,1)$ and $j(2,1)$ belong to the same strongly connected component U_1, and were moved after $j(3,2)$ by condition (8). Figure 7 (a) illustrates the schedule constructed from $(\boldsymbol{m}, \boldsymbol{\pi}_1)$ by CONSTRUCT, in which C_{ℓ_1} is satisfied.

As the second example, we consider the minimization of the cost of the setup operation $\sigma(j(3,1))$ (which is positive if this operation is executed, but is 0 otherwise). As explained in Section 2.5, this is described by a soft nonrenewable resource constraint

$$C_{\ell_2} : x_{\sigma(j(3,1)),\mathrm{setup}} \leq 0.$$

Assume again that the current solution is $(\boldsymbol{m}, \boldsymbol{\pi}_0)$ (note that we do not consider constraint C_{ℓ_1} in this example). C_{ℓ_2} is violated by $(\boldsymbol{m}, \boldsymbol{\pi}_0)$, and move set (ii) of Section 3.3 is generated. Here, $J'' = \{\sigma(j(3,1))\}$ holds, and we choose two maximal simple directed paths $\boldsymbol{v}^-(\sigma(j(3,1))) = (\sigma(j(3,1)))$ and $\boldsymbol{v}^+(\sigma(j(3,1))) = (\sigma(j(3,1)), j(3,1), j(2,1), j(1,1), j(1,2), sink)$. The move set $MV_{\ell_2}(\boldsymbol{m}, \boldsymbol{\pi}_0)$ of (14) is given by

$$MV_{\ell_2}(\boldsymbol{m}, \boldsymbol{\pi}_0) = \left\{ \begin{array}{l} shift\_aft(\sigma(j(3,1)), j(2,1)), shift\_aft(\sigma(j(3,1)), j(1,1)), \\ shift\_aft(\sigma(j(3,1)), j(1,2)) \end{array} \right\},$$

as $\sigma(j(3,1)) \prec j(3,1) \prec sink$ holds. If we apply a move $shift\_aft(\sigma(j(3,1)), j(1,2))$ to $\boldsymbol{\pi}_0$, then the new list becomes

$$\boldsymbol{\pi}_2 = (j', j(1,1), j(2,1), j(1,2), \sigma(j(3,1)), j(3,1), j(3,2), j(2,2), sink),$$

since $\sigma(j(3,1))$ and $j(3,1)$ are contained in the same strongly connected component, and $\sigma(j(3,1)) \prec j(3,2)$ holds. From this list, the schedule that requires no setup operation is obtained by CONSTRUCT, which is illustrated in Figure 7 (b).

Finally, we consider the minimization of makespan by introducing a soft constraint C_{ℓ_3} of type (6). C_{ℓ_3} is violated by the solution $(\boldsymbol{m}, \boldsymbol{\pi}_0)$, and move set of type (iii) is generated, where $J^+ = \{sink\}$ and $J^- = \emptyset$. Let a maximal simple directed path $\boldsymbol{v}^-(sink) = (\sigma(j(3,1)), j(3,1), j(2,1), j(1,1), j(1,2), sink)$ be chosen for our purpose. The move set $MV_{\ell_3}(\boldsymbol{m}, \boldsymbol{\pi}_0)$ of (15) is then given by

$$MV_{\ell_3}(\boldsymbol{m}, \boldsymbol{\pi}_0) = \{shift\_bef(j(3,1), j(2,1)),\ shift\_bef(j(2,1), j(1,1))\}.$$

The moves $shift\_bef(\sigma(j(3,1)), j(3,1))$ and $shift\_bef(j(1,1), j(1,2))$ are not considered because $\sigma(j(3,1)) \prec j(3,1)$ and $j(1,1) \prec j(1,2)$ hold. By applying a move $shift\_bef(j(3,1), j(2,1))$ to $\boldsymbol{\pi}_0$, for example, we obtain the list

$$\boldsymbol{\pi}_3 = (j', j(1,1), j(2,1), \sigma(j(3,1)), j(3,1), j(3,2), j(1,2), j(2,2), sink).$$

Recall again that activities j', $j(1,1)$ and $j(2,1)$, and activities $\sigma(j(3,1))$ and $j(3,1)$, respectively, belong to the same strongly connected components. Using this, the makespan is reduced from 8 to 7. The resulting schedule is illustrated in Figure 7 (c).

3.5 Implementation of tabu search

In our tabu search, an initial solution is randomly selected from the search space Λ of (10), and then repeatedly replaced by its best non-tabu neighbor until a stopping criterion is achieved; e.g., the predetermined number of iterations or computational time is reached, or a satisfactory schedule is found.

The tabu list in our algorithm is implemented as follows. For each move, we define its *attribute* by the pair of move type ("change_mod" or "shift") and the corresponding activity; e.g., (change_mod, j), (shift, j_1) and (shift, j_2) are the attributes of $change\_mod(j, m'_j)$, $shift\_aft(j_1, j_2)$ and $shift\_bef(j_1, j_2)$, respectively. The tabu list T keeps the attributes of all the moves executed in τ recent iterations, where τ is a program parameter called *tabu tenure*. As stated in Step k (3) below, it prohibits all moves that possess an attribute in the tabu list.

The value of τ has a large influence on the performance of tabu search. In our algorithm, τ is adaptively controlled by the method introduced in the CSP code of [17]. Although we do not give detailed computational results here, this control mechanism of τ performs well also in our RCPSP code.

The outline of our algorithm for RCPSP is described as follows.

Tabu search algorithm for RCPSP

Step 0 (initialization): Generate an initial solution $(\boldsymbol{m}^{(1)}, \boldsymbol{\pi}^{(1)})$ randomly. Initialize the incumbent schedule $(\boldsymbol{m}^*, \boldsymbol{s}^*)$ to the schedule obtained from $(\boldsymbol{m}^{(1)}, \boldsymbol{\pi}^{(1)})$ by applying CONSTRUCT. Set $k := 1$.

Step k (k-th iteration):

(1) Calculate the move set $MV(\boldsymbol{m}^{(k)}, \boldsymbol{\pi}^{(k)})$ and the neighborhood $\tilde{N}(\boldsymbol{m}^{(k)}, \boldsymbol{\pi}^{(k)})$ for the current solution $(\boldsymbol{m}^{(k)}, \boldsymbol{\pi}^{(k)})$.

(2) Apply CONSTRUCT to each solution $(\boldsymbol{m}, \boldsymbol{\pi}) \in \tilde{N}(\boldsymbol{m}^{(k)}, \boldsymbol{\pi}^{(k)})$, and evaluate the resulting schedule $(\boldsymbol{m}, \boldsymbol{s})$ by its penalty $p(\boldsymbol{m}, \boldsymbol{s})$.

(3) For the non-tabu neighbor $(\boldsymbol{m}', \boldsymbol{\pi}')$ having the smallest penalty, let $(\boldsymbol{m}^{(k+1)}, \boldsymbol{\pi}^{(k+1)}) := (\boldsymbol{m}', \boldsymbol{\pi}')$. If $p(\boldsymbol{m}', \boldsymbol{s}') < p(\boldsymbol{m}^*, \boldsymbol{s}^*)$, then update $(\boldsymbol{m}^*, \boldsymbol{s}^*) := (\boldsymbol{m}', \boldsymbol{s}')$.

(4) If the stopping criterion is met, then output $(\boldsymbol{m}^*, \boldsymbol{s}^*)$ and halt; otherwise, let $k := k + 1$ and go to Step k.

4 Computational experiments

Our algorithm was coded in C, and tested on many benchmarks of RCPSP, and some practical problems. In this section, we report some of such computational results. All experiments was conducted on a workstation Sun Ultra 2 (300MHz, 1G bytes memory).

4.1 Benchmarks from PSPLIB

Kolisch et al. [11] generated a large number of RCPSP instances systematically, which are available in the Project Scheduling Problem LIBrary (PSPLIB) together with their optimum or best known values. All instances are from the standard RCPSP in the sense that the available amounts of resources do not change with periods, and that no immediate precedence constraints are considered. We solved four types of instances j60.sm, j90.sm, j120.sm and j30.mm, which are 2110 instances in total. Statistics of these instances such as the number of activities, the number of modes in which each activity can be processed, and the number of resources are shown in Table 1. All instances have precedence constraints as well as resource constraints, and the objective is to minimize the makespan.

For each instance in j60.sm, j90.sm, j120.sm and j30.mm, we ran our algorithm until 5000 iterations are reached, respectively. Computational results are summarized in Table 2. As listed in Table 2, the best known schedules (and sometimes even better ones) were found for many instances. (By running our algorithm repeatedly with different initial solutions, we could increase the number of instances with improved best known values. Table 3 shows the current status, as of August 2000, of the number of instances whose best values were improved by our codes, and the number of instances whose optimal solutions were found; i.e., these makespans are equal to their lower bounds.)

In the last three columns in Table 2, the minimum, average and maximum computational times required for each run (i.e., 5000 iterations) are listed. It seems that the deviation of the computational time is quite large. This is because the size of neighborhood varies from instance to instance. As mentioned in Section 3.3, the neighborhood $\tilde{N}(\boldsymbol{m}, \boldsymbol{\pi})$ is based on the type of constraints violated, and in the case of the standard RCPSP, its size highly depends on the number of activities in a critical path (since the move set is given by (15)). Figure 8 shows the relationship between the computational time and the makespan of the best schedule obtained by our code for 600 instances in j120.sm. From this, we observe a tendency that more computational time is required for larger makespan (since the critical path tends to contain more activities).

Next, we compare our results for j120.sm with those obtained by three existing RCPSP algorithms; genetic algorithm by Hartmann (H) [8], Lagrangian heuristic by Möhring et al. (MSSU) [14] and ant colony optimization algorithm by Merkle et al. (MMS) [13]. Table 4 gives the average deviation of the best obtained values from the critical path based lower bounds. For each instance, H, MMS and our algorithm have been executed for 5000 iterations, respectively. Each run of MSSU and our algorithm consumed the average and maximum computational times of 65 and 711 seconds (on a Sun Ultra 2 with 200MHz clock pulse), and 109.4 and 575.8 seconds (on a Sun Ultra 2 with 300MHz clock pulse), respectively. According to [8, 13], algorithms H and MMS required 3.07 seconds (on a Pentium 133MHz) and 25 seconds (on a Pentium III 500MHz) for 1000 iterations, respectively. These results indicate that our tabu search is competitive with other powerful codes in its solution quality, although our algorithm is slower. In closing, we emphasize that our code is designed for the generalized RCPSP. In fact, among the tested four algorithms, only our code can be applied to instances in j30.mm, which are multi-mode RCPSPs.

4.2 Problems from real-world applications

The generalized RCPSP can encompass a wide variety of problems encountered in real life applications. In this subsection, we consider two such examples, which cannot be formulated as the standard RCPSP. We explain below how they are formulated as extended RCPSP instances and how we apply our code, together with their computational results. The first problem can be found in spinning mills in textile industry, for example, while the second problem can be found in some types of chemical plants.

4.2.1 Production scheduling with setup activities

We are given a set of activities J, a set of machines R, and the duration of the schedule T (days). J is partitioned into $|R|$ sets J_r such that $J = \cup_{r \in R} J_r$, and all activities $j \in J_r$

Table 1: Properties of four instance types in PSPLIB.

instance type	#activities	#modes	#renewable resources	#nonrenewable resources
j60.sm	60	1	4	0
j90.sm	90	1	4	0
j120.sm	120	1	4	0
j30.mm	30	3	2	2

Table 2: Computational results for benchmarks in PSPLIB.

instance type	#instances	#instances whose best known values are found[1]	CPU time per run (in seconds)		
			min	average	max
j60.sm	480	370 (1)	3.2	13.6	78.5
j90.sm	480	367 (9)	5.5	29.3	212.3
j120.sm	600	202 (40)	8.4	109.4	575.8
j30.mm	550	360 (50)	7.2	16.8	41.3

1. The best known values are as of August 2000. Figures in parenthesis are the numbers of instances whose best known values were improved by our code.

Table 3: The number of instances whose best known values were improved by our code (as of August 2000).

instance type	#instances whose best values are improved by our code	#instances with proven optimality
j60.sm	4	0
j90.sm	14	4
j120.sm	79	33
j30.mm	126	0

must be processed on machine r, each requiring consecutive p_j days (thus, each activity has only one mode). For each activity j, its due date d_j is also specified, and its cost function $cost\_due(c_j, d_j)$ is given, which can take any form (in our experiment, it contains both earliness and tardiness penalties). To change the activity on machine r from j_1 to j_2 (where $j_1, j_2 \in J_r$), the setup operation $setup(j_1, j_2)$ of q consecutive days on the same machine r as well as the setup cost are required. The q is common to all setup operations, but the setup cost, $cost\_set(j_1, j_2)$, depends on j_1 and j_2. Since setup operations consume the setup resource, at most K_t machines can process setup operations in each day $t = 1, 2, \dots, T$. The K_t takes different values, depending on whether t is a weekday or not. Each machine must process exactly one activity (which may be a setup operation) everyday (i.e., there is no idle time). To make this possible, we assume that the data satisfy the constraints

$$\sum_{j \in J_r} p_j + q(|J_r| - 1) = T, \qquad r \in R.$$

The objective is to minimize the total cost

$$\sum_{j \in J} cost\_due(c_j, d_j) + \sum_{r \in R} \sum_{i=1}^{|J_r|-1} cost\_set(a(r,i), a(r,i+1)), \tag{16}$$

where $a(r,i)$ denotes the i-th activity processed on machine r.

To formulate this problem as an extended RCPSP, we introduce

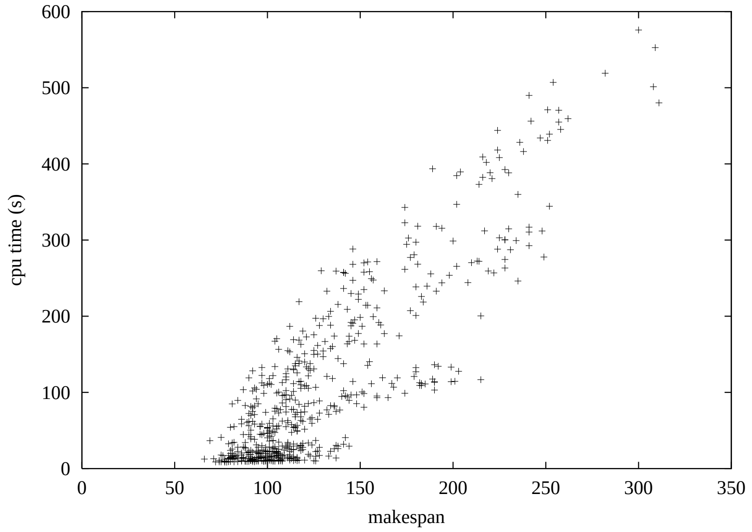


Figure 8: The relationship between computational time and makespan.

Table 4: Average deviations from the critical path based lower bounds over all 600 instances in j120.sm.

algorithm	deviation (%)
Genetic algorithm by Hartmann [8]	35.60
Our algorithm	35.86
Lagrangian heuristic by Möhing et al. [14]	36.2
Ant colony optimization by Merkle et al. [13]	36.65

- the renewable resource r for each machine r,
- the mode set with only one mode m_j for each $j \in J$, except that the setup activity $\sigma(j)$ for j ($\in J_r$) has the mode set,

$$\{setup(j', j) \mid j' \neq j,\ j' \in J_r\} \cup \{\text{dummy}\},$$

  and
- the renewable resource r_{set} common to all setup activities.

As explained in Section 2.4, each setup activity $\sigma(j)$ ($j \in J_r$) has the immediate precedence constraint $\sigma(j) \prec\!\prec_r j$, and $\sigma(j)$ must be processed in mode "$setup(j', j)$" (resp., "dummy") if j' is immediately preceding $\sigma(j)$ on r (i.e., $j' = a(r, i)$ and $j = a(r, i+1)$ for some i) (resp., if j is the first activity on r). Finally, $|J|$ constraints on completion times c_j,

$$c_j \leq T, \qquad j \in J, \tag{17}$$

are added so that no machine has idle time.

As noted in Section 2.5, the minimization of total cost (16) is attained via soft constraints. The minimization of the earliness and tardiness cost $\sum_j cost\_due(c_j, d_j)$ is realized by soft renewable resource constraints, as described at the end of Section 2.5. The cost of setup activities, where setup activity $\sigma(j)$ in mode $setup(j', j)$ consumes $cost\_set(j', j)$ of the resource, is described by a nonrenewable resource constraint,

$$\sum_{r \in R} \sum_{j \in J_r} \sum_{\substack{j' \neq j \\ j' \in J_r}} cost\_set(j', j) x_{\sigma(j), setup(j', j)} \leq 0.$$

The left hand side represents the total cost of setup activities (i.e., the penalty of this constraint).

There may exist no feasible schedule to this problem. To obtain a practical schedule in such a case, we regard each setup resource constraint of amount K_t as a soft one with a large weight. In practice, its violation (i.e., too many setups in a day) may be resolved by allocating more manpower.

The instance we solved has $|J| = 77$, $|R| = 19$, $T = 30$ and $q = 1$, and each machine $r \in R$ is assigned 2–7 activities, and the processing times p_j vary between 1 and 25 days. As to setup activities, at most three machines are allowed on each weekday, but no machine is allowed on weekend or holiday. The cost $cost\_set(j_1, j_2)$ of each setup activity in mode $setup(j_1, j_2)$ takes one of the two values $\{0, 1\}$, and the cost function $cost\_due(c_j, d_j)$ for j ($\in J$) is defined by

$$cost\_due(c_j, d_j) = \begin{cases} 10, & c_j \leq d_j - 5, \\ 2k, & c_j = d_j - k, \quad 1 \leq k \leq 4, \\ 0, & c_j = d_j, \\ 10, & c_j = d_j + 1, \\ 16, & c_j = d_j + 2, \\ 18, & c_j = d_j + 3, \\ 20, & c_j \geq d_j + 4. \end{cases}$$

Since this problem has many soft constraints, the size of neighborhood $\tilde{N}(\boldsymbol{m}, \boldsymbol{\pi})$ (see Section 3.3) is rather large. To demonstrate the effect of the random neighborhood reduction

Table 5: Effect of the neighborhood reductions.

	the sample rate ρ [%]					
	5%	10%	30%	50%	80%	100%
the number of runs in which the optimum schedule could be found	30/30	30/30	27/30	22/30	14/30	11/30
average CPU time (sec) required to find the optimum schedule	17.5	9.1	—	—	—	—

stated in Section 3.3, we executed six tabu search algorithms in which $\rho = 5$, 10, 30, 50, 80 and 100% of solutions in $\tilde{N}(\boldsymbol{m}, \boldsymbol{\pi})$ are randomly sampled in each iteration, respectively. We ran our code 30 times for each value of ρ, setting the computational time to 180 seconds for each run. Table 5 shows the number of runs (out of 30) for each ρ in which we could find the optimum schedule (the optimality was proved by integer programming, as explained later), as well as the average computational time required to find the optimum schedule if we could find it in all 30 runs. From this result, we can see that the neighborhood reduction is quite effective, and the best value of ρ is 10% for this instance. However, at the current stage, some preliminary experiments are needed to find an appropriate value of ρ.

This problem can be formulated as an integer program (IP) by introducing $|J_r|!$ 0-1 variables for each machine r, corresponding to all permutations of activities in J_r. By solving the resulting integer program by a commercial package SOPT, we found the optimum solution with its optimality in one second (not including the computational time for generating and storing the data, which takes about ten seconds). Although this integer programming approach seems more effective for this small instance, our approach will become more useful for larger instances, since the number of 0-1 variables in IP formulation increases exponentially with the number of activities. To show this, we solved three more instances I_8, I_9 and I_{10}, in which the maximum numbers of activities assigned to one machine (i.e., $\max_r |J_r|$) are 8, 9 and 10, respectively. The IP approach could find optimal solutions for instances I_8 and I_9 in 2 and 21 seconds (requiring about 1 and 14 minute(s) for generating and storing the data), respectively, but could not solve instance I_{10} because the data size exceeds the memory bound of the computer. On the other hand, our approach could handle all instances I_8, I_9 and I_{10}. We ran our code 30 times for instances I_8, I_9 and I_{10}, consuming 50.6, 85.7 and 90.8 seconds, respectively, for each of the 30 runs. For instances I_8 and I_9, we could obtain the optimal solutions (which were obtained by IP approach). Optimal solutions for instances I_{10} are not known.

4.2.2 Parallel machine shop scheduling with setup activities

The next example is the parallel shop scheduling problem treated in [16]. This problem consists of a set of jobs J, which are classified into G groups, and a set of parallel machines R. For each job $j \in J$, the set of machines R_j on which j can be processed and the total amount D_j of demanded days for processing j are specified. Each job can be processed on more than one machine with preemption, and be also processed on several machines in parallel. Jobs are scheduled on a day basis, and the schedule horizon is T days. This problem has the following constraints:

- Each machine can process only one job at a time, and runs even on weekends and holidays.
- Once the processing of a job is initiated, it must be continued for at least three days.
- Switching jobs within the same group on a machine takes no time, but changing a job into another job in a different group requires the setup construction of q days.
- There is no precedence constraint between jobs.
- The switching operations and setup constructions are possible only on weekdays, and their numbers which can be carried out in a day are at most K_{sw} and K_{set}, respectively.

The objectives are to minimize the weighted sum of the total amount (days) of unsatisfied demands and the number of switchings.

Since a job may be processed on more than one machine, for each pair of job j and machine $r \in R_j$, we prepare activity $a(j,r)$ with mode set

$$\{t \mid 3 \leq t \leq T\} \cup \{\text{dummy}\},$$

where we consider that job j is processed on machine r for t days (resp., is not processed on r) if activity $a(j,r)$ takes mode "t" (resp., "dummy"). We denote by $t_{a(j,r)}$ the number of days given to activity $a(j,r)$ on machine r. Furthermore, for each activity $a(j,r)$, we introduce setup activity $\sigma(a(j,r))$ whose mode set is $\{\text{setup}, \text{dummy}\}$. Hence, the number of activities is $(\sum_j |R_j|) \times 2$ in total. In addition to $|R|$ machine resources, two renewable resources r_{set} and r_{sw} are introduced to keep the numbers of setup constructions and switchings in a day within K_{set} and K_{sw} (or 0 in a holiday or weekend), respectively. All of these renewable resource constraints are considered hard. To obtain schedules whose horizon is T (days) and in which no machine has idle time, we impose the following constraints:

$$\begin{aligned} c_{a(j,r)} &\leq T, && \text{for all pairs } j \text{ and } r \in R_j, && (18)\\ \sum_{j \in \{j \mid r \in R_j\}} t_{a(j,r)} + t_{\sigma(a(j,r))} &\geq T, && \text{for all } r. && (19) \end{aligned}$$

As explained in Section 2.5, these additional constraints (18) and (19) are treated as soft constraints C_ℓ with large weights w_ℓ in our code (w_ℓ is set to 100 in our experiments).

In general, a penalty function for a soft inequality constraint $C_\ell : f_\ell(\boldsymbol{m}, \boldsymbol{s}) \leq 0$ (including (18) and (19)) may be naturally defined by

$$p_\ell(\boldsymbol{m}, \boldsymbol{s}) = \max\{0, f_\ell(\boldsymbol{m}, \boldsymbol{s})\}. \tag{20}$$

However, it is observed in our experiment that allowing the search into infeasible region of soft constraints is useful to find many different feasible schedules. Thus, to allow some amount of violations (occasionally), the following penalty function appears more appropriate.

$$\begin{aligned} p_\ell(\boldsymbol{m}, \boldsymbol{s}) &= \max\left\{0, \frac{f_\ell(\boldsymbol{m}, \boldsymbol{s})}{|f_\ell(\boldsymbol{m}, \boldsymbol{s})|}\right\}, \text{ or} && (21)\\ p_\ell(\boldsymbol{m}, \boldsymbol{s}) &= \max\left\{0, \frac{f_\ell(\boldsymbol{m}, \boldsymbol{s})}{|f_\ell(\boldsymbol{m}, \boldsymbol{s})|} + \delta \cdot f_\ell(\boldsymbol{m}, \boldsymbol{s})\right\}, && (22) \end{aligned}$$

Table 6: Computational results with different penalty functions.

penalty functions for constraints (18) and (19)	penalty		
	min	average	max
$\max\{0, f_\ell(\boldsymbol{m}, \boldsymbol{s})\}$	24.0	30.1[1]	—[2]
$\max\left\{0, \dfrac{f_\ell(\boldsymbol{m}, \boldsymbol{s})}{\lvert f_\ell(\boldsymbol{m}, \boldsymbol{s})\rvert}\right\}$	16.0	19.3	23.0
$\max\left\{0, \dfrac{f_\ell(\boldsymbol{m}, \boldsymbol{s})}{\lvert f_\ell(\boldsymbol{m}, \boldsymbol{s})\rvert} + \delta \cdot f_\ell(\boldsymbol{m}, \boldsymbol{s})\right\}$	15.0	16.5	20.0

1. the average of those runs in which feasible schedules could be found.
2. In two runs (out of 30), we could not find any feasible schedule.

where $\delta > 0$ is a small value. The penalty function defined by (21) depends only on whether $f_\ell(\boldsymbol{m}, \boldsymbol{s})$ is positive (i.e., the constraint is violated) or not, and (22) is a compromise between (20) and (21).

The soft constraints associated to the objective function can be written by

$$\sum_{r \in R_j} t_{a(j,r)} \geq D_j, \qquad \text{for all } j, \tag{23}$$

corresponding to minimizing the total amount of unsatisfied demands, and

$$m_{a(j,r)} = m_{\sigma(a(j,r))} = \text{“dummy”}, \qquad \text{for all } a(j,r), \tag{24}$$

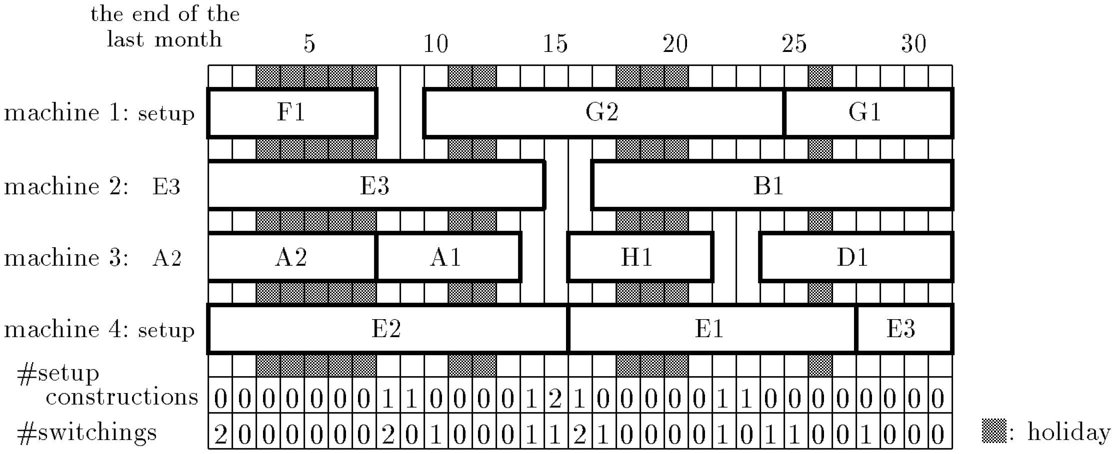
corresponding to minimizing the number of switchings (a switching operation is required if an activity is processed in non-dummy mode), respectively.

We solved an instance with $|J| = 11$, $|R| = 4$, $G = 7$, $T = 31$, $K_{\text{set}} = 2$, $K_{\text{sw}} = 3, q = 2$ and the objective function of

$$0.5 \times (\text{the total amount of unsatisfied demands}) + (\text{the number of switchings}).$$

Thus, we set the weights of soft constraints (23) and (24) to 0.5 and 1, respectively. We tried penalty functions (20), (21) and (22) with $\delta = 10^{-5}$ for constraints (18) (19), respectively. For each case, we ran our tabu search 30 times consuming 300 seconds in each run. The size of neighborhood is randomly reduced to $\rho = 50\%$ as in Section 4.2.1. The computational results are shown in Table 6, where the minimum, average and maximum penalty values are indicated. "—" in the last column means that no feasible schedule could be found in some runs, and such runs are excluded in the calculation of the average penalty. Among these three, penalty function (22) brought the best results. A best schedule (whose penalty value is 15.0) obtained by our experiments is illustrated in Figure 9, in which all hard constraints as well as (18) and (19) are satisfied. Note that job E3 is processed on two machines 2 and 4, in this schedule.

To evaluate our results, we formulated this problem as a mixed integer program, and solved it by using a commercial package CPLEX. We could find feasible schedules with



#groups = 7 (group names: A, B, D, E, F, G, H)
#jobs = 11 (job names: A1, A2, B1, D1, E1, E2, E3, F1, G1, G2, H1)

job j:	set of machines R_j	demanded days D_j
A1:	{1,3}	6
A2:	{1,2,3}	1
B1:	{2}	14
D1:	{1,2,3,4}	8
E1:	{1,2,3,4}	11
E2:	{1,2,3,4}	14
E3:	{1,2,3,4}	18
F1:	{1,2,3,4}	6
G1:	{1,2}	9
G2:	{1,2}	15
H1:	{1,2,3,4}	6

Figure 9: An example of schedules obtained by our RCPSP code.

penalties 16.5 and 16.0 in about 50 minutes and 9 hours, respectively, but no better schedule could be found in 10 hours of computational time. Thus, the schedules obtained by our code appear to be quite good.

5 Conclusion

In this paper, we first gave a generalization of RCPSP as a framework for handling complicated practical scheduling problems. Then, we proposed a tabu search algorithm, in which a list of activities is used to represent a solution so that only feasible schedules are searched efficiently. Our code was tested on a number of benchmarks of RCPSP and performed well. In addition to these benchmarks, we solved problems from real applications, in which setup activities are required, available amounts of resources may vary with periods, and objective functions may not be regular. For these problems, we succeeded to find practically usable schedules. We also demonstrated the effectiveness of the random neighborhood reduction. However, its performance highly depends on the sampling rate ρ, which is currently preset to an appropriate value by preliminary experiment; it would be a topic of future research to devise an automatic control mechanism of ρ.

Since our algorithm was developed as a general problem solver, for specific problems, it may not be able to attain as good performance as those algorithms specially designed for them (as observed in the computational results of Section 4.1). However, this would not deny the importance of our code, because it is easy to try, and can find satisfactory schedules in many practical problems, which cannot be formulated as the standard RCPSP.

As a possible avenue of future research, we point out that it may be possible to improve our code by customizing some of its components to the given problem; the following changes may be worth considering.

- As mentioned in Section 3.1, in general, optimal schedules may not be constructible from the solution space (10) by CONSTRUCT. Applying constructive methods other than CONSTRUCT may be able to overcome this limitation.

- In our code, the neighborhood is defined as a set of solutions obtained by applying only those moves which are effective to remove some violations of constraints. We discussed in Section 3.3 a few simple rules to select such moves for some typical constraints. To increase applicability, we can use other move sets specialized to the given problems.

Acknowledgments

We appreciate Prof. S. Morito of Waseda University for the data in Section 4.2.2. We are also grateful to Prof. T. Uno of Tokyo Institute of Technology, Prof. M. Yagiura of Kyoto University, and three anonymous referees for their valuable comments and suggestions. This research was partially supported by the Grant-in-Aid for Scientific Research of Priority Areas "Algorithm Engineering as a New Paradigm: A Challenge to Hard Computation Problems" of the Ministry of Education, Science, Sports and Culture of Japan. The first author was supported by Research Fellowship of the Japan Society for the Promotion of Science for Young Scientists.

References

[1] T. Baar, P. Brucker and S. Knust, "Tabu search algorithms and lower bounds for the resource-constrained project scheduling problem", in: S. Voss, S. Martello, I. Osman and C. Roucairol (eds.): *Meta-heuristics: Advances and Trends in Local Search Paradigms for Optimization*, Kluwer Academic Publishers (1998) 1–18.

[2] C.E. Bell and J. Han, "A new heuristic solution method in resource-constrained project scheduling", *Naval Research Logistics* 38 (1991) 315–331.

[3] J. Błażewicz, W. Domschke and E. Pesch, "The job shop scheduling problem: Conventional and new solution techniques", *European Journal of Operational Research* 93 (1996) 1–33.

[4] P. Brucker, A. Drexl, R. Möhring, K. Neumann and E. Pesch, "Resource-constrained project scheduling: Notation, classification, models, and methods", *European Journal of Operational Research* 112 (1999) 3–41.

[5] J-H. Cho and Y-D. Kim, "A simulated annealing algorithm for resource constrained project scheduling problems", *Journal of the Operational Research Society* 48 (1997) 736–744.

[6] F. Glover, "Tabu search — Part I", *ORSA Journal on Computing* 1 (1989) 190–206.

[7] S. Hartmann, "A competitive genetic algorithm for resource-constrained project scheduling", *Naval Research Logistics* 45 (1998) 733–750.

[8] S. Hartmannn, "Self-adapting genetic algorithms with an application to project scheduling", Manuskripte aus den Instituten für Betriebswirtschaftslehre der Universität Kiel #506 (1999).

[9] W. Herroelen, B. De Reyck and E. Demeulemeester, "Resource-constrained project scheduling: A survey of recent developments", *Computers and Operations Research* 25 (1998) 279–302.

[10] R. Kolisch, "Serial and parallel resource-constrained project scheduling methods revisited: Theory and computation", *European Journal of Operational Research* 90 (1996) 320–333.

[11] R. Kolisch and A. Sprecher, "PSPLIB — A project scheduling library", *European Journal of Operational Research* 96 (1997) 205–216.

[12] J-K. Lee and Y-D. Kim, "Search Heuristics for Resource Constrained Project Scheduling", *Journal of the Operational Research Society* 47 (1996) 678–689.

[13] D. Merkle, M. Middendorf and H. Schmeck, "Ant colony optimization for resource-constrained project scheduling", *Proceedings of the Genetic and Evolutionary Computation Conference* (GECCO) (2000) 893–900.

[14] R.H. Möhring, A.S. Schulz, F. Stork and M. Uetz, "Solving project scheduling problems by minimum cut computations", Technical Report #680, Fachbereich Mathematik, Technische Universität Berlin (2000).

[15] M. Mori and C.C. Tseng, "A genetic algorithm for multi-mode resource constrained project scheduling problem", *European Journal of Operational Research* 100 (1997) 134–141.

[16] S. Morito, J. Imaizumi and J.W. Park, "A mathematical programming approach to a tightly constrained scheduling problem", *Proceeding of the Production Scheduling Symposium '96* (1996) 85–90 (in Japanese).

[17] K. Nonobe and T. Ibaraki, "A tabu search approach to the constraint satisfaction problem as a general problem solver," *European Journal of Operational Research* 106 (1998) 599–623.

[18] J.H. Patterson, "A comparison of exact approaches for solving the multiple constrained resource, project scheduling problem", *Management Science* 30 (1984) 854–867.

[19] S.E. Sampson and E.N. Weiss, "Local search techniques for generalized resource constrained project scheduling problem", *Naval Research Logistics* 40 (1993) 665–675.

[20] A. Sprecher, R. Kolisch and A. Drexl, "Semi-active, active, non-delay schedules for the resource-constrained project scheduling problem", *European Journal of Operational Research* 80 (1995) 94–102.

[21] P.R. Thomas and S. Salhi, "A tabu search approach for the resource constrained project scheduling problem", *Journal of Heuristics* 4 (1998) 123–139.

[22] M.G.A. Verhoeven, "Tabu search for resource-constrained scheduling", *European Journal of Operational Research* 106 (1998) 266–276.