Available online at www.sciencedirect.com
SCIENCE DIRECT®
European Journal of Operational Research 149 (2003) 282–301
EUROPEAN JOURNAL OF OPERATIONAL RESEARCH
www.elsevier.com/locate/dsw

Resource-constrained project scheduling: A critical activity reordering heuristic ☆

Vicente Valls [a,*], Sacramento Quintanilla [b], Francisco Ballestín [a]

[a] Dpto. de Estadística e Investigación Operativa, Facultad de Matemáticas, Universitat de Valencia, Dr. Moliner, 50, 46100 Burjassot, Valencia, Spain
[b] Dpto. de Economía Financiera y Matemática, Facultad de Económicas y Empresariales, Universitat de Valencia, Avda. de los Naranjos, s/n, Edificio Departamental Oriental, Valencia, Spain

Abstract

In this paper, we present a new metaheuristic algorithm for the resource-constrained project-scheduling problem. The procedure is a non-standard implementation of fundamental concepts of tabu search without explicitly using memory structures embedded in a population-based framework. The procedure makes use of a fan search strategy to intensify the search, whereas a strategic oscillation mechanism loosely related to the forward/backward technique provides the necessary diversification. Our implementation employs the topological order (TO) representation of schedules. To explore the TO vector space we introduce three types of moves, two of them based on the concept of relative criticality, and a third one based on multi-pass sampling ideas. The strategic utilisation of probabilities for move construction is another distinguishing feature of our approach. Extensive computational testing with more than 2000 problem instances shows the merit of the proposed solution method.

Keywords: Resource-constrained project scheduling; Tabu search; Heuristics

1. Introduction

The resource-constrained project-scheduling problem (RCPSP) may be stated as follows. A project consists of a set of n activities numbered 1 to n, where each activity has to be processed without interruption to complete the project. The dummy activities 1 and n represent the beginning and end of the project. The duration of an activity j is denoted by d_j where $d_1 = d_n = 0$. There are K renewable resource types. The availability of each resource type k in each time period is R_k units, $k = 1, \ldots, K$. Each activity j requires r_{jk} units of resource k during each period of its duration where $r_{1k} = r_{nk} = 0$, $k = 1, \ldots, K$. All parameters are assumed to be non-negative integer valued. There are precedence relations of the finish–start type with a

☆ This research was partially supported by the CICYT under contract TAP99-1123 and by the Generalitat Valenciana under contract FP 198-CB-12-303.
* Corresponding author. Tel.: +34-963544313; fax: +34-963544735.
E-mail addresses: vicente.valls@uv.es (V. Valls), maria.quintanilla@uv.es (S. Quintanilla), francisco.ballestin@uv.es (F. Ballestín).

0377-2217/03/$ - see front matter
doi:10.1016/S0377-2217(02)00768-3

zero parameter value (i.e., FS = 0) defined between the activities. In other words, activity i precedes activity j if j cannot start until i has been completed. The structure of a project can be represented by an activity-on-node network $G = (V, A)$, where V is the set of activities and A is the set of precedence relationships. S_j (P_j) is the set of successors (predecessors) of activity j. It is assumed that $1 \in P_j$, $j = 2, \ldots, n$, and $n \in S_j$, $j = 1, \ldots, n-1$. The objective of the RCPSP is to find a schedule S of the activities, i.e., a set of starting times $(s_1, s_2, \ldots, s_n)$ where $s_1 = 0$ and the precedence and resource constraints are satisfied, such that the schedule duration $T(S) = s_n$ is minimised. Let T^* be the minimum schedule duration or minimum makespan.

As a job shop generalisation, the RCPSP is NP-hard in the strong sense (see Blazewicz et al., 1983). In recent years, the applications and difficulties of the RCPSP and its extensions have attracted increasing interest from researchers and practitioners. We refer to the surveys provided by Icmeli et al. (1993), Özdamar and Ulusoy (1995), Kolisch and Padman (1997), Herroelen et al. (1998) and Brucker et al. (1999). For solving the RCPSP, the most competitive algorithms seem to be those of Demeulemeester and Herroelen (1992), Sprecher (1996), Brucker et al. (1998) and Mingozzi et al. (1998). However, only small-sized problem instances with up to 30 activities can be solved exactly in a satisfactory manner. Therefore, heuristic solution procedures remain the only feasible methods of handling practical RCPSPs. Many heuristic approaches have been proposed for the RCPSP (we refer to the aforementioned surveys). Kolisch and Hartmann (1999) have given an overview of heuristics focusing on X-pass methods (single pass methods, multi-pass methods, sampling procedures) and metaheuristics (simulated annealing, genetic algorithms, tabu search). Hartmann and Kolisch (2000) provide an investigation of the performance of those RCPSP heuristics. Their computational results indicate that the best metaheuristics outperform the best sampling approaches. As stated by the authors, this is mainly because sampling procedures generate each schedule anew without considering any information given by previously visited solutions as—metaheuristics typically do.

Li and Willis (1992) propose an iterative forward/backward scheduling technique for project scheduling under general resource constraints: renewable, non-renewable and doubly constrained resources with uneven availabilities over time. Basically, the procedure iteratively applies serial forward and backward scheduling until no further improvement in the project duration can be obtained. The activity finish (start) times of a forward (backward) schedule determine the activity priorities for the next backward (forward) schedule. Özdamar and Ulusoy (1996) take up again the basic idea of iterative forward/backward scheduling and construct a totally different algorithm. They employ a parallel schedule generation scheme (SGS) in which the selection of the next activity to be scheduled is made according to a conflict base procedure.

Nonobe and Ibaraki (1999) recently presented a tabu search based heuristic algorithm for a generalisation of the RCPSP. It is able to handle multi-mode processing, variable availability of renewable and non-renewable resources, and complex objective functions. They tested their code for a number of benchmarks of the jobshop and RCPSP, as well as some problems from real applications.

In this paper, we present a new metaheuristic algorithm, critical activity reordering algorithm (CARA), for the RCPSP. The procedure is a non-standard implementation of the fundamental concepts of tabu search without explicitly using memory structures embedded in a population-based framework. The procedure makes use of a fan search strategy to intensify the search whereas a strategic oscillation mechanism loosely related to the forward/backward technique provides the necessary diversification. Our implementation employs the topological order (TO) representation of schedules (Valls et al., 1999). Three types of moves are used to explore the TO vector space. Two of these are based on the concept of relative criticality and a third is based on multi-pass sampling ideas. Depending on the phase of the search, one or another type of move is used to explore the neighbourhood of a solution. Another distinguishing feature of our approach is the strategic utilisation of probabilities for move construction. Probabilities are designed to reflect evaluations of attractiveness in such a way that the highest evaluations receive probabilities that favour their selection without completely excluding other options. Probabilistic selection induces variety,

compensates for the imperfection of "best" decisions, and makes reliance on memory structures unnecessary.

2. Schedule representation

A TO of the activities of a project is an order that is compatible with the precedence relations (i.e., vector $\gamma = (\gamma_1, \gamma_2, \ldots, \gamma_n)$ of integer numbers between 1 and n, and that satisfies: $(i, j) \in A$ implies $\gamma_i < \gamma_j$ and that $\gamma_i \neq \gamma_j$, $i \neq j$). Given a schedule S, any vector $\gamma = (\gamma_1, \gamma_2, \ldots, \gamma_n)$ of integer numbers between 1 and n that satisfies: $s_i < s_j$ implies $\gamma_i < \gamma_j$ and that $\gamma_i \neq \gamma_j$, $i \neq j$—is called a *TO representation* of S. The TO representation is a special case of the random key representation of Bean (1994), the priority value representation of Lee and Kim (1996), and Cho and Kim (1997), and the problem-space based representation of Storer et al. (1992), Leon and Ramamoorthy (1995), and Naphade et al. (1997). In some sense, it can be considered as the dual of an activity list representation (Kolisch and Hartmann, 1999). The serial SGS can be used as a decoding procedure to obtain an active schedule $S(\gamma)$ from a TO vector γ by selecting at each stage the activity j with the lowest order γ_j. In other words, the orders play the role of priority values. A schedule S may be represented by several TO vectors that only differ in the orders assigned to those activities with the same start times. However, if S is active and $\gamma(S)$ is a representation of S then $S(\gamma(S)) = S$. In the rest of the paper, we assume that all considered schedules are active and use S and $\gamma(S)$ interchangeably.

3. The Advance(S) set

Given a feasible schedule S, LF_i denotes the latest finish time of activity i as determined by backward recursion omitting resource constraints when $T(S)$ is used as the upper bound on the makespan. Then, $\mathrm{LF}_n = s_n = T(S)$ and the latest finish time of at least one of the immediate predecessors of activity n is equal to $T(S)$.

The *total float or total slack* of activity i *relative to* S can be defined as $\mathrm{TF}(i, S) = \mathrm{LF}_i - f_i$ where $f_i = s_i + d_i$ is the *finish time* of activity i. An activity i, $i \neq 1$, n, is defined as *critical relative* to S if $\mathrm{TF}(i, S) = 0$. Let us denote by $\mathrm{Cr}(S)$ the set of all critical activities relative to S. Note that $\mathrm{Cr}(S) \neq \emptyset$ Then, the following condition has to be fulfilled if the makespan of a feasible schedule S' is shorter than $T(S)$:

$$s'_i < s_i \quad \forall i \in \mathrm{Cr}(S) \tag{1}$$

To improve a given feasible schedule S our algorithm tries to advance the start times of all activities in $\mathrm{Cr}(S)$. Given a TO representation γ of S, the method used consists of diminishing the order γ_i associated with each activity with the objective of obtaining a new TO vector γ' whose $S(\gamma')$ satisfies the condition (1). To advance the start time of a critical activity, it may also be necessary to advance one or more non-critical predecessors of the activity.

As a way of illustrating the preceding ideas, consider the following example in Fig. 1.

A feasible schedule S of the above problem is depicted in Fig. 2.

The vector $\gamma = (1, 2, 3, 6, 8, 9, 4, 7, 10, 5, 11)$ is a TO representation of the schedule S. The other alternative representations are: $(1, 3, 2, 6, 8, 9, 4, 7, 10, 5, 11)$, $(1, 2, 3, 5, 8, 9, 4, 7, 10, 6, 11)$ and $(1, 3, 2, 5, 8, 9, 4, 7, 10, 6, 11)$. Table 1 shows the values of LF_j, f_j and $\mathrm{TF}(j, S)$ for each activity j of the illustrative example.

As we can observe $\mathrm{Cr}(S) = \{9\}$. As the activity 5 is immediately before activity 9 in γ and as $\gamma_5 = 8$, the maximum we can advance activity 9 in γ without advancing any other activity is by assigning the new order $\gamma'_9 = 9$; and in this way obtaining the new TO vector $\gamma' = (1, 2, 3, 6, 8, 10, 4, 7, 9, 5, 11)$. This advance does

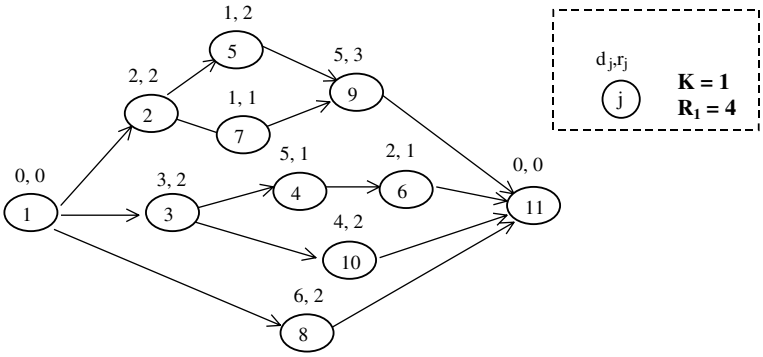

Fig. 1. Activity network for illustrative example.

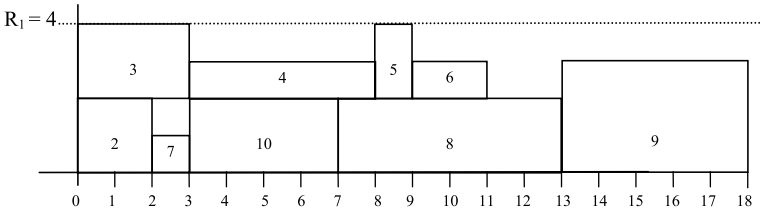

Fig. 2. Feasible schedule for the illustrative example.

Table 1
LF_j, f_j and $TF(j,S)$ for the activities of the illustrative example

Activity j	1	2	3	4	5	6	7	8	9	10	11
LF_j	8	12	11	16	13	18	13	18	18	18	18
f_j	0	2	3	8	9	11	3	13	18	7	18
$TF(j,S)$	8	10	8	8	4	7	10	5	0	11	0

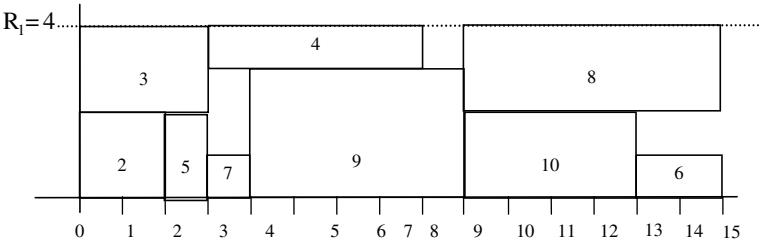

Fig. 3. The schedule $S(\gamma')$.

not shorten the makespan as $S(\gamma') = S$. Nevertheless, if we also advance activity 5 as much as possible then we obtain $\gamma' = (1, 2, 4, 8, 3, 10, 5, 9, 6, 7, 11)$. The associated schedule $S(\gamma')$ is depicted in Fig. 3.

There is also another motivation for advancing activities with non-zero total slack that precede critical activities. If the LF was calculated using a stronger lower bound than that obtained by backward recursion

omitting resource constraints (see Klein and Scholl, 1999), then possibly some of the activities with positive slack would become critical. Advancing a non-critical activity j, predecessor of a critical activity i, does not leave a "hole" to advance the activity i—unless those activities that are both successors of j and predecessors of i are also advanced. Furthermore, the slack of an activity is defined with regard to a schedule S. The information offered by the slack is useful to define movements that transform S, providing that these movements do not overly transform S. A strategy to limit the reach of these transformations is to maintain as fixed the order of approximately the first half of these activities (the head) and only reorder the other half (the tail). This strategy increases the intensifying effect because it limits the search region. The above results in the following definition of the set Advance(S):

$$\text{Advance}(S) = \left\{ j \in \text{Cr}(S,10)/\gamma_j > \frac{n}{2} \right\} \cup \left\{ j \in \text{Cr}(S,3)/\gamma_j \leqslant \frac{n}{2} \right\}$$

where

$$\text{Cr}(S,\delta) = \text{Cr}(S) \cup \{ j/0 < \text{TF}(j,S) \leqslant \delta \quad \text{and} \quad \exists i \in S_j \cap \text{Cr}(S) \} \quad \delta \geqslant 0$$

Therefore, to improve a given feasible schedule S our algorithm tries to advance all activities in Advance(S). To achieve this, a threshold was established above the order of activities in γ (mino), near to $n/2$. This action divides the activities into a head and a tail in such a way that the part of the sequence that changes (the tail) always contains the set Advance(S). The head is formed by those activities with lesser order than the mino threshold and the remaining activities form the tail. That is to say, that the head (tail) contains the activities sequenced first (last), when using the serial SGS as the decoding procedure:

$$\text{mino} = \min\{\min\{\gamma_j / j \in \text{Advance}(S)\} - 3, n/2\}$$

$$\text{Head}(S) = \text{Head}(\gamma) = \{ j/\gamma_j < \text{mino} \}$$

$$\text{Tail}(S) = \text{Tail}(\gamma) = \{ j/\gamma_j \geqslant \text{mino} \}$$

Note that the various values of δ (3 and 10, respectively) used in the definition of Advance(S) reflect the strategy of treating the first and the second half of a schedule differently.

Returning to schedule S in Fig. 2, $\text{Cr}(S) = \{9\}$; $\text{Cr}(S,10) = \{9\} \cup \{2,5,7\} = \{2,5,7,9\}$; $\text{Cr}(S,3) = \{9\} \cup \emptyset = \{9\}$; $\text{Advance}(S) = \{5,9\}$, $\text{mino} = \min\{\min\{8,10\} - 3, 11/2\} = 5$, $\text{Head}(S) = \text{Head}(\gamma) = \{1,2,3,7\}$ and $\text{Tail}(S) = \text{Tail}(\gamma) = \{4,5,6,8,9,10,11\}$.

4. Algorithm overview

The algorithm we present is based on the TO representation. It can be considered as a non-standard implementation of tabu search principles. Given a TO representation γ of a schedule S, we distinguish two parts in S: the head and the tail. The head is formed by approximately half of the activities—those with lower order. The remaining activities form the tail. Our procedure tries to alternatively improve the tail and the head by maintaining them fixed respectively. It can be seen as a type of *strategic oscillation approach* (Glover and Laguna, 1997).

We consider two phases in the application of the algorithm. The first phase begins with an initial population POP with TO vectors. To introduce diversity and quality in POP, some of the solutions are generated randomly and others are generated using priority rules. In this phase, two movements are probabilistically and fundamentally defined as a function of the information offered by the relative slack of the activities. To improve the tail (the head) of a TO representation γ of a schedule S a *fan search* (Glover and Laguna, 1997) is performed on the sub-set of TO vectors that assign the same order as γ to the activities

in the head (tail). The basic idea behind a fan search is to generate some p alternative moves at a given step and then to create a fan of solution streams, one for each alternative. The best available moves, one for each stream, provide the p new streams at the next step. A limit is placed on the number of iterations that the streams generate beyond the first iteration. This limit may depend on the state of the search and differ for each stream.

The second phase begins with the best solution obtained in the first phase. A modified biased random sampling method is applied to this solution and a new initial population is generated. The same first stage strategy of improving alternatively the head and the tail is then applied. In this phase, a movement is defined probabilistically by *window random sampling*.

5. Phase 1 of the algorithm

5.1. Move definition

Let γ be a TO representation of an active schedule S. The aim of the two types of movements shown below is to generate a new TO vector γ':

$$\gamma'_j < \gamma_j, \quad \forall j \in \text{Advance}(S) \tag{2}$$

$$\gamma'_j = \gamma_j, \quad \forall j \in \text{Head}(S) \tag{3}$$

$$\forall i, j \in \text{Tail}(S) \setminus \text{Advance}(S) : \gamma_i < \gamma_j \rightarrow \gamma'_i < \gamma'_j \tag{4}$$

The conditions (2) and (3) were discussed above. The condition (4) requires the maintenance of the order relative to those tail activities that we do not specifically wish to advance. This helps maintain part of the structure of S so that the information given by the slack does not lose its significance and reinforces the intensifier effect of the search. Condition (2) should be seen as a general objective that we cannot always strictly satisfy.

The first type of movement, LF_BACKWARD_MOVE(γ), iteratively orders the activities in Tail(γ) from back to front by first assigning the n order, then $n-1$, and so on successively. In each iteration, an order k is assigned to an activity j, starting with $\gamma'_n = n$. The assignment of orders is guided mostly by the LF priority rule. In each iteration, the selection of the activity j is undertaken in three steps. In the first step, the *Candidate* set—meaning the set of activities of Tail(γ) that has not been reordered but whose successors have been reordered—is divided into three sub-sets and then one of these is selected probabilistically. The first sub-set is C_NA, the candidate activities which are not in Advance(S); the second is C_E, the candidate activities in Advance(S) which are *eligible*—meaning that their order in γ is greater than k; and the third is C_NE, the remaining candidate activities. In the second step, the sub-set C_NE is selected only if the sub-sets C_NA and C_E are both empty. If this is the case, condition (2) is not met. If not, the probability of selecting any of the sub-sets C_NA or C_E is proportional to the maximum LF value of the activities in the sub-set. If one sub-set is empty then the other would be selected. In the third step, j is probabilistically selected from within the chosen sub-set. If either C_NA or C_NE is selected, then j is the activity in the sub-set with the highest order in γ. If C_E is selected, then j is chosen probabilistically in the following way: a 75% probability that it is the C_E activity with the highest LF, and a 25% probability that it is one of the other C_E activities randomly chosen. This three-step process and the way in which the probabilities are constructed facilitate the double objective of using the information offered by LF, and so avoiding taking the same, possibly erroneous, decisions; and satisfying the conditions (2)–(4). A detailed pseudocode of LF_BACKWARD_MOVE(γ) appears in Appendix A.

Returning to the example shown on Fig. 2, we can see which sequence produces LF_BACKWARD_MOVE(γ) supposing for the sake of simplicity that the random number generated between 0 and 1 (r) is always 0.5.

Step 0: Advance(S) $= \{5, 9\}$; mino $= 5$; Head(S) $= \{1, 2, 3, 7\}$; Tail(S) $= \{4, 5, 6, 8, 9, 10, 11\}$

$\gamma'_1 = 1$; $\gamma'_2 = 2$; $\gamma'_3 = 3$; $\gamma'_7 = 4$;
$\gamma'_4 = \infty$; $\gamma'_5 = \infty$; $\gamma'_6 = \infty$; $\gamma'_8 = \infty$; $\gamma'_9 = \infty$; $\gamma'_{10} = \infty$; $\gamma'_{11} = \infty$;
$\gamma'_{11} = 11$;
$k = 10$

As a result (Table 2), $\gamma = (1, 2, 3, 6, 8, 9, 4, 7, 10, 5, 11)$ will have become $\gamma' = (1, 2, 3, 7, 5, 10, 4, 8, 9, 6, 11)$. Therefore, conditions (2)–(4), are met—although the sequence $T((S(\gamma)) = T(S(\gamma'))$ is not shortened.

Table 2
LF_BACKWARD_MOVE(γ) for the illustrative example

Step 1	Step 2	Step 3	Step 4
Candidate $= \{6, 8, 9, 10\}$	$\alpha = 0$	$\gamma_6 = 9$; $\gamma_8 = 7$; $\gamma_{10} = 5 \rightarrow j = 6$	$\gamma'_6 = 10$
Eligible $= \{11\}$	$\beta = \max\{18, 18, 18\} = 18$		$k = 9$
C_NA $= \{6, 8, 9, 10\} \setminus \{5, 9\} = \{6, 8, 10\}$	$0.5 > 0/(0 + 18) \rightarrow$		
C_E $= \{6, 8, 9, 10\} \cap \{5, 9\} \cap \{11\} = \emptyset$	$\rightarrow$ SELECT $=$ C_NA		
C_NE $= \{6, 8, 9, 10\} \setminus (\{6, 8, 10\} \cup \emptyset) = \{9\}$			
Candidate $= \{4, 8, 9, 10\}$	$\alpha = \max\{18\} = 18$	$j = 9$	$\gamma'_9 = 9$
Eligible $= \{9, 11\}$	$\beta = \max\{16, 18, 18\} = 18$		$k = 8$
C_NA $= \{4, 8, 9, 10\} \setminus \{5, 9\} = \{4, 8, 10\}$	$0.5 \leqslant 18/(18 + 18) \rightarrow$		
C_E $= \{4, 8, 9, 10\} \cap \{5, 9\} \cap \{9, 11\} = \{9\}$	SELECT $=$ C_E		
C_NE $= \{4, 8, 9, 10\} \setminus (\{4, 8, 10\} \cup \{9\}) = \emptyset$			
Candidate $= \{4, 5, 8, 10\}$	$\alpha = 0$	$\gamma_4 = 6$; $\gamma_8 = 7$; $\gamma_{10} = 5 \rightarrow j = 8$	$\gamma'_8 = 8$
Eligible $= \{6, 9, 11\}$	$\beta = \max\{16, 18, 18\} = 18$		$k = 7$
C_NA $= \{4, 5, 8, 10\} \setminus \{5, 9\} = \{4, 8, 10\}$	$0.5 > 0/(0 + 18) \rightarrow$		
C_E $= \{4, 5, 8, 10\} \cap \{5, 9\} \cap \{6, 9, 11\} = \emptyset$	SELECT $=$ C_NA		
C_NE $= \{4, 5, 8, 10\} \setminus (\{4, 8, 10\} \cup \emptyset) = \{5\}$			
Candidate $= \{4, 5, 10\}$	$\alpha = \max\{13\} = 13$	$\gamma_4 = 6$; $\gamma_{10} = 5 \rightarrow j = 4$	$\gamma'_4 = 7$
Eligible $= \{5, 6, 9, 11\}$	$\beta = \max\{16, 18\} = 18$		$k = 6$
C_NA $= \{4, 5, 10\} \setminus \{5, 9\} = \{4, 10\}$	$0.5 > 13/(13 + 18) \rightarrow$		
C_E $= \{4, 5, 10\} \cap \{5, 9\} \cap \{5, 6, 9, 11\} = \{5\}$	SELECT $=$ C_NA		
C_NE $= \{4, 5, 10\} \setminus (\{4, 10\} \cup \{5\}) = \emptyset$			
Candidate $= \{5, 10\}$	$\alpha = \max\{13\} = 13$	$j = 10$	$\gamma'_{10} = 6$
Eligible $= \{5, 6, 8, 9, 11\}$	$\beta = \max\{18\} = 18$		$k = 5$
C_NA $= \{5, 10\} \setminus \{5, 9\} = \{10\}$	$0.5 > 13/(13 + 18) \rightarrow$		
C_E $= \{5, 10\} \cap \{5, 9\} \cap \{5, 6, 8, 9, 11\} = \{5\}$	SELECT $=$ C_NA		
C_NE $= \{5, 10\} \setminus (\{10\} \cup \{5\}) = \emptyset$			
Candidate $= \{5\}$	$\alpha = \max\{13\} = 13$	$j = 5$	$\gamma'_5 = 5$
Eligible $= \{4, 5, 6, 8, 9, 11\}$	$\beta = 0$		$k = 4$
C_NA $= \{5\} \setminus \{5, 9\} = \emptyset$	$0.5 \leqslant 13/(13 + 0) \rightarrow$		$k = 4 < 5 =$ mino $\rightarrow$ stop
C_E $= \{5\} \cap \{5, 9\} \cap \{4, 5, 6, 8, 9, 11\} = \{5\}$	SELECT $=$ C_E		
C_NE $= \{5\} \setminus (\{10\} \cup \{5\}) = \emptyset$			

The second type of movement, DSLACK_FORWARD_MOVE(γ), first constructs a schedule S' and then defines $\gamma' = \gamma(S')$. S' is constructed by iteratively completing the partial schedule formed by sequencing the head activities using a serial SGS. To select the next activity j to be sequenced a three-step procedure was implemented that is similar to that in LF_BACKWARD_MOVE(γ) and shares the same objectives. Nevertheless, the dynamic slack (DSLACK) now partially establishes the priority. Additionally, the dynamic slack, by definition, is updated in each iteration according to the current state of the partial schedule. In the first step, the set of candidate activities, *Candidate*, is the unscheduled activities in Tail(γ) whose predecessors have been scheduled, are partitioned into two sub-sets, depending on whether they belong to the set Advance(S) (C_A) or not (C_NA). Then we select one of the sub-sets in the second step with a probability proportional to the minimum DSLACK of their elements, except when only one of the sub-sets contains an activity with negative slack, in which case, this sub-set is directly selected. In the third step, if C_A (C_NA) has been selected then j is the activity with least dynamic slack (least order in γ) in C_A (C_NA). Note that if some iteration of the algorithm meets $\gamma_{i_0} = \min\{\gamma_i / i \in \text{C\_A}\} < k + 1$, then condition (2) is not met. A detailed pseudocode of DSLACK_FORWARD_MOVE(γ) appears in Appendix A.

5.2. Tail improving procedure

Fig. 4 outlines the procedure to improve the tail of a TO vector γ. The procedure starts by generating (PopSize + nbs) neighbours of γ and then selecting the best PopSize to form the population POP. The neighbours of γ are generated by the repeated application of X_MOVE(γ), where X_MOVE represents the procedure LF_BACKWARD_MOVE half the time and the procedure DSLACK_FORWARD_MOVE the other half. Then a fan search is sequentially applied to each TO vector in POP. If this set of fan searches provides an improved solution then the whole procedure is restarted from it.

FAN_SEARCH(γ) is an iterative procedure that, in the first iteration, generates "width" neighbours of γ by the repeated application of X_MOVE(γ). Subsequent iterations generate width neighbours of the best TO obtained in the previous iteration. The procedure stops after "depth" iterations without improvements.

5.3. Head improving procedure

TAIL_IMPROVING(γ) is a procedure that searches the sub-space formed by the TOs that share the same head with γ and looks for the "best" tail that can be appended to the head of γ. Once this search is exhausted, it seems natural to change the search strategy in the sense of fixing the tail and looking for the "best" head to be appended to it.

Before explaining the new procedure HEAD_IMPROVING(γ) some definitions have to be introduced. An *active* (or *left active*) schedule can be defined as a schedule where no activity can be started earlier

1. $POP = \phi$
2. **For** [i = 1,..., $PopSize + nbs$]{
   2.1 σ_i = X_MOVE(γ)
   2.2 $POP = POP \cup \{\sigma_i\}$
   }
3. Select the $PopSize$ best TO vectors $\in POP$: $\sigma_1,...,\sigma_{PopSize}$
4. **For** [i = 1,...,$PopSize$]{
   4.1 σ = FAN_SEARCH(σ_i)
   4.2 **If** [T(σ) < T(γ)] γ =A ; Go to 2
   }
5. **Return**γ

Fig. 4. TAIL_IMPROVING(γ) outline.

without delaying some other activity or violating the constraints. A formal definition can be found in Sprecher et al. (1995). Analogously, a *right active schedule* is a schedule where no activity can be finished later without advancing some other activity, or violating the constraints, or increasing the makespan. Given a schedule S, to *justify* an activity $j \neq n$ (1) to the right (left) consists of obtaining a schedule S' such that $s'_i = s_i$, $i \neq j$, $s'_j \geqslant s_j$ ($s'_j \leqslant s_j$) and s'_j is as big (small) as possible. Given a schedule S, the right (left) justification of the activities j in decreasing (increasing) order of f_j (s_j) provides a right active (left active) schedule S^{R} (S^{L}). S^{R} (S^{L}) is not unique, and depends on the tie-breaking rule used. If $s_1^{\mathrm{R}} > 0$ ($s_n^{\mathrm{L}} < T$), then S^{R} (S^{L}) is shorter than S. It is not difficult to see that $T(S^{\mathrm{R}}) \leqslant T(S)$ $(T(S^{\mathrm{L}}) \leqslant T(S))$.

The project obtained by simply reversing the precedence relations is called the *reverse* of the original project. Given a schedule S, the following transformation derives a schedule S^{r} of the reverse project: $s_j^{\mathrm{r}} = T(S) - f_j$. Clearly, if S is a schedule of a project (reverse project), then $(S^{\mathrm{R}})^{\mathrm{r}}$ is a left active schedule in the reverse project (original project). The tail (head) of S is transformed, roughly speaking, into the the head (tail) of $(S^{\mathrm{R}})^{\mathrm{r}}$. This allows the implementation of the new procedure HEAD_IMPROVING making use of the TAIL_IMPROVING(γ) procedure.

Fig. 5 shows a procedure for reordering the head of a γ where RIGHT_JUSTIFICATION(γ) is the procedure that right justifies the activities j in decreasing order of f_j (ties randomly broken).

5.4. Strategic oscillation

The oscillatory mechanism described in Fig. 6 provides an effective interplay between intensification and diversification. The search is alternatively driven to two different regions, the region with "fixed head" and that of "fixed tail", whereas the fan search is a means of intensifying the search in both regions.

5.5. Initial population

The initial population of solutions is generated with the goal of introducing quality and diversity. The procedure INITIAL_SET_1($N1$) randomly generates 10 TO vectors. Two schedules are generated from each TO vector γ by applying both serial and parallel SGS. We also use nine well-known priority rules. The rules are defined and described in Appendix B. Each priority rule generates one or two schedules depending on whether both or only one SGS can be applied. For each generated schedule S, a TO representation with evaluation $T(S)$ is obtained which becomes a candidate for inclusion in the initial population. The initial population consists of the $N1$ candidates with the best evaluation.

1. $\gamma_{\mathrm{best}} = \gamma$
2. $\sigma =$ RIGHT_JUSTIFICATION(γ)
3. $\varphi =$ TAIL_IMPROVING (σ^{r})
4. $\theta =$ RIGHT_JUSTIFICATION(φ)
5. If $[T(\theta^{\mathrm{r}}) < T(\gamma_{\mathrm{best}})]$ $\gamma_{\mathrm{best}} = \theta^{\mathrm{r}}$
6. Return γ_{best}

Fig. 5. HEAD_IMPROVING(γ) outline.

1. $\sigma =$ TAIL_IMPROVING (γ)
2. $\varphi =$ HEAD_IMPROVING (σ)
3. If $[T(\varphi) < T(\gamma)]$ $\gamma = \varphi$; go to 1
4. Return γ

Fig. 6. HEAD_TAIL_IMPROVING(γ) outline.

1. POP = INITIAL_SET_1(N1)
2. Apply HEAD_TAIL_IMPROVING (γ) to each $\gamma \in POP$
3. Return γ_{best} where γ_{best} is the best TO vector obtained

Fig. 7. PHASE_1 outline.

5.6. Phase 1 outline

Having described the main components of phase 1 in the previous sections, we now provide its outline (Fig. 7).

6. Phase 2 of the algorithm

Phase 2 is a search procedure similar to phase 1 but differing in the generation of the initial population, the move definition, and the tail improving procedure.

Phase 1 starts from a population of relatively low quality solutions and drives the search to a local optimum γ_0, and hopefully a high quality TO vector. As phase 2 starts the search within a high quality region, it seems appropriate to make controlled moves to avoid going to regions of much lesser quality. Multi-pass sampling methods can be adequately adapted to define such controlled moves.

6.1. Initial population

Given γ, a new TO vector γ' can be obtained by the application of what we call a β *biased random sampling method* ($0 \leqslant \beta \leqslant 1$). This method makes use of the serial SGS. The following strategy is employed to select in each iteration the next activity to be scheduled. A number $p \in (0,1)$ is randomly generated. If $p < \beta$, then j is the eligible activity with least order. Otherwise, j is one of the other eligible activities selected by biased random sampling, employing the orders as priority values in order to obtain the selection probabilities. The parameter β has the following interpretation. On average, $n\beta$ is the number of iterations in which γ' and γ would select the same activity. So if we define $\beta = 1 - k/n$, then $k = n(1-\beta)$ is on average the number of iterations in which the decisions according to γ' and γ would be different. Therefore, the parameter β controls how different the new TO vector is from the original.

The procedure INITIAL_SET_2($\gamma_0, N2$) generates 400 TO vectors with $\beta = 1 - 20/n$ and 200 TO vectors with $\beta = 1 - 10/n$ by applying the β biased random sampling method to γ_0 and then selects the best $N2$ for the initial population POP for phase 2.

6.2. Move definition

The procedure WINDOW_SAMPLING_MOVE(γ) first constructs a schedule S' and then defines $\gamma' = \gamma(S')$. S' is constructed by completing the partial schedule formed by the head activities while making use of the serial SGS and a modified random sampling method for selecting the next activity j to be scheduled. The modification works as follows. In each iteration an activity belongs to the *Candidate* set if its predecessors have been scheduled. Then an activity is eligible only if the difference between its order and the minimal order $w_0 = \min\{\gamma_i / i \in \text{Candidate}\}$ is less or equal to a given non-negative parameter *window*. The activity i is selected from the eligible activities by means of a random sampling method. The closeness of γ' to γ can be controlled by selecting the appropriate parameter *window*. However, the effect of the window is schedule dependent. A given value for the window will produce, in general, fewer changes in a schedule where many activities begin simultaneously, than in a schedule where many activities are scheduled one

after the other. A robust strategy is to use two values, one with a small window and the other with a larger window. A detailed pseudocode of the procedure WINDOW_SAMPLING_MOVE(γ) appears in Appendix A.

6.3. Tail improving procedure

In this phase, TAIL_IMPROVING(γ) is a multi-pass window random sampling method. It applies WINDOW_SAMPLING_MOVE(γ) one hundred times with window = 2 and another one hundred times with window = 5 and returns the best TO vector obtained. The execution time of phase 2 is largely determined by the number of times WINDOW_SAMPLING_MOVE(γ) is applied. Initial computational tests show that 200 passes require a reasonable computational time and that for this number of passes, small *window* values offer better results than large values. It is also shown that two windows values offer more robustness than one value. The minimum window value that enables γ' to be different from γ is one. Window = 2 will produce schedules quite similar to γ. Window = 5 allows results that are different to those obtained with window = 2, without excessive deteriation in the average quality of the sequences generated.

6.4. Phase 2 outline

Phase 2 is very similar to phase 1; but the new definition of TAIL_IMPROVING(γ) (see Fig. 8) must be borne in mind. It is important to note that each time an improved TO vector is generated the whole procedure restarts from it.

The outline of the complete CARA appears in Fig. 9.

7. Computational experiments

This section reports on the results of the computational studies of CARA. The algorithm has been coded in C and the experiments have been performed on a personal computer AMD at 400 MHz.

7.1. Test problems

For test instances, we have used the standard sets *j*30, *j*60, *j*90 and *j*120 for the RCPSP. These were generated using ProGen (Kolish et al., 1995). The sets *j*30, *j*60 and *j*90 consist of 480 projects with four

1. POP = INITIAL_SET_2 (γ_0, $N2$)
2. Apply HEAD_TAIL_IMPROVING_2 (γ) to each $\gamma \in PO$ in increasing order of makespan.
3. Return γ_{best} where γ_{best} is the best TO vector obtained

Fig. 8. PHASE_2(γ_0) outline.

1. γ_0 = PHASE_1
2. γ_1 = PHASE_2(γ_0)
3. Return S(γ_1)

Fig. 9. CARA outline.

resource types and 30, 60 and 90 non_dummy activities, respectively. The set $j120$ consists of 600 projects with four resource types and 120 activities. There are 2040 instances in total. These instances were generated under a full factorial experimental design with the following three independent problem parameters: network complexity, resource factor and resource strength. Details of these problem instances are given in Kolish et al. (1995) and Kolish and Sprecher (1997). They are available in the Project Scheduling Problem Library (PSPLIB) along with their optimum or best-known values.

7.2. Parameter setting

To fix the values of the parameters of the algorithm we have performed some preliminary experiments with half the problems of instance set $j120$—the first five seeds. Some of these values were established when describing the algorithm, and values of the remaining parameters are: PopSize = depth = width = 4, nbs = 20, and $N1 = N2 = 20$. Subsequent preliminary trials indicated the usefulness of increasing the size of the tail for instance set $j30$ problems. To achieve this, and only when solving problems of 30 activities, the head contains approximately a quarter of the total number of activities. Specifically, for problems of 30 activities, we have changed $n/2$ for $n/4$ in the definitions of Advance(S) and of mino. The remaining parameters could be slightly improved by customising them to each instance set. However, we decided, for the sake of clarity, to maintain for all instance sets those values fixed in the preliminary study that used half of the $j120$ set problems.

7.3. Computational results

Table 3 summarises the results of our first set of experiments. The first column indicates the instance set referred to in the results shown in each row. The second column, labelled $\sum$CARA, consists of the sum of the values obtained by CARA whereas the third column, $\sum$PSPLIB, shows the sum of the best values in PSPLIB as of June 1, 2000. The fourth (fifth) column, av_dev (max _dev), consists of the average (maximal) percentage deviations from the best solutions in PSPLIB. The number of instances for which CARA obtains the best-known values in PSPLIB is reported in the sixth column (best_sol). In the same column the cardinality of the instance set referred to appears between brackets. It is worth noting that these known values include those found by CARA. The average (maximal) computation time in seconds is shown in column seven (eight), labelled av_CPU (max _CPU). Finally, the average percentage deviation from the critical path makespan is reported in the ninth column, labelled CPM_dev.

The deviations and the times recorded in Table 3 are practically identical to those obtained in the preliminary trials using only one half of the instance problems. This seems to indicate that the performance of CARA is stable. Predictably, av_dev and av_CPU increased with the size of the problem. Nevertheless, this increase is not linear. It is worth noting that if we had not changed $n/2$ to $n/4$ in $j30$ then the number of optimal solutions obtained would have been 445.

The second set of experiments analyses the repercussions that variations of the values of the parameters used would produce on the performance of the algorithm. The tests undertaken indicate that the values of

Table 3
Computational results for $j30$, $j60$, $j90$ and $j120$

	$\sum$CARA	$\sum$PSPLIB	av_dev	max _dev	Best_sol	Av_CPU	Max_CPU	CPM_dev
$j120$	76356	74609	1.9222	8.1081	199(600)	17.00	43.94	34.5330
$j90$	46247	45879	0.6158	5.4054	368(480)	4.63	25.49	11.1234
$j60$	38671	38422	0.5034	6.3158	375(480)	2.76	14.61	11.4546
$j30$	28335	28316	0.0562	3.4483	463(480)	1.61	6.15	13.4573

Table 4
Computational results for several values of (p, q)

p, q	$\sum$CARA	av_dev	av_CPU
600, 0	76373	1.9379	17.71
400, 200	76356	1.9222	17.00
0, 600	76329	1.9053	16.71
300, 300	76380	1.9571	16.39
200, 400	76365	1.9436	16.44
200, 100	76393	1.9692	16.68
600, 300	76386	1.9490	16.82
800, 400	76366	1.9460	17.35

the majority of the parameters (all except $N1$ and $N2$) could be varied widely without significant changes in the performance of the algorithm. In order to be brief, the following tables only show the results for the instance set $j120$. The results, except where otherwise stated, are similar for the other sets. To illustrate a case of a stable parameter, Table 4 shows the results obtained when varying the number of times (p, q) that a β biased random sampling method was used in procedure INITIAL_SET_2$(\gamma_0, N2)$ with $\beta = 1 - 20/n$, and with $\beta = 1 - 10/n$, respectively. We can see in Table 4 that the major difference between the two entries of the column av_dev is 0.0639 and it appears that the choice of the parameter values (p, q) inside the studied range does not noticeably influence the performance of the algorithm—even if it is true that the best value of av_dev was obtained with the combination $(0, 600)$. Nevertheless, as this combination produced worse results than $(400, 200)$ for the problems in $j30$, $j60$ and $j90$ we decided to fix $(p, q) = (400, 200)$ in CARA as a compromise.

Table 5 gives the results for different values of the phase 1 initial population size ($N1$). Not surprisingly, av_CPU increases with the size of the initial population, yet nevertheless, the best value of av_dev is obtained with $N1 = 20$. This would seem to indicate that to obtain the best results it is not enough to simply start with quality initial solutions—the initial population must also be diverse. If time is of concern it is worth noting that CARA with $N1 = 5$ produces also good quality solutions in half the time required by CARA with $N1 = 20$.

The next parameter studied was $N2$—which is the size of the initial population in phase 2. The results shown in Table 6 show that increasing $N2$ will increase the quality of the solutions obtained, though at the price of increased computing time. Of the four instance sets, $N2 = 20$ is the best compromise between solution quality and computing time.

Worth noting is the fact that with $N2 = 80$ some 468 optimal solutions for $j30$ were obtained with av_CPU = 3.38 and av_dev = 0.038.

With the third set of experiments, we wanted to know if a two-phase strategy is necessary or if phase one alone could produce results similar to CARA. The results of Table 7 show that the phase 1 of CARA alone with $N1 = 20$ offers a av_dev = 2.6648, a value far from the av_dev = 1.9222 offered by CARA. We can also see that phase 1 alone with $N1 = 30$ takes longer than CARA, and offers a worse av_dev. The two-phase

Table 5
Computational results for $N1 = 5$, 12, 20, 30

$N1$	$\sum$CARA	av_dev	av_CPU
5	76503	2.0973	8.13
12	76384	1.9670	12.10
20	76356	1.9222	17.00
30	76379	1.9353	23.55

Table 6
Computational results for $N2 = 10, 20, 30, 40$

$N2$	$\sum$ CARA	av_dev	av_CPU
10	76546	2.1467	14.08
20	76356	1.9222	17.00
30	76308	1.8652	19.26
40	76206	1.7642	22.55

Table 7
Computational results for phase 1 alone

	$\sum$ CARA	av_dev	av_CPU
Phase 1 alone, $N1 = 20$	77026	2.6648	11.30
Phase 1 alone, $N1 = 30$	76965	2.5777	18.17

Table 8
Best values in the initial population

	$\sum$ best_ini_value	ini_av_dev	ini_max_dv	$\sum$ best_pr_value	pr_av_dev	pr_max_dv
*j*120	79720	6.0401	22.6994	79732	6.0544	22.6994
*j*90	47285	2.5035	16.1290	47306	2.5380	16.2602
*j*60	39660	2.6381	16.3636	39692	2.7148	16.3636
*j*30	28874	1.6885	13.1579	28962	1.9365	16.6667

strategy is effective and cannot be substituted by applying just phase 1—even if the size of the initial population is increased.

It is also interesting to establish by how much CARA improves the best schedule obtained in the initial population. Table 8 contains data regarding the initial population. Column 2 shows the sum of the best makespans in each initial population; and columns 3 (4) show the average (maximal) percentage deviations from the best solutions in PSPLIB. Columns 5–7 show the same concepts but just referring to the best initial population schedule obtained using a priority rule. We can see that applying CARA noticeably reduces the values of av_dev and max_dev in all the instance sets. We can also see that the difference pr_av_dev − ini_av_dev reduces when the problem size increases. This appears to indicate that as the size of the problem increases it is more difficult to obtain randomly generated quality schedules.

7.4. Comparison with other heuristics

Comparing CARA with other heuristic algorithms is difficult because the data published by various authors refer to different computers, use different stopping rules, and express the quality of the solutions obtained in different ways. Despite this, and with the aim of giving an impression of the performance of other state-of-the-art heuristics, we show the computational results for three metaheuristic algorithms (Hartmann, 1998; Bouleimen and Lecocq, 1998; Nonobe and Ibaraki, 1999). The first two performed best among the 16 heuristics tested in the study of Hartmann and Kolisch (2000). This study does not take into consideration the Nonobe and Ibaraki algorithm. Table 9 summarises the percentage deviations from the optimal makespan for the instance set *j*30 and from the lower bound critical path for instance sets *j*60 and *j*120 when the number of generated and evaluated schedules was limited to 5000. To give an impression of the computation times, the Hartmann algorithm requires, on average, approximately 15 s to compute 5000

Table 9
Percentages deviation for the two best heuristics in Hartmann and Kolisch (2000)

	*j*30	*j*60	*j*120
Hartmann	0.25	11.89	36.74
Bouleimen and Lecocq	0.23	11.90	37.68

schedules for the *j*120 set on a Pentium-based computer at 133 MHz under the Linux operating system. We can observe that CARA yields better results than these two algorithms. Of course, such a comparison is not fair because CARA requires more computational time.

The Hartman algorithm offers better results than Bouleimen and Lecocq for the instance set *j*120 and similar results for the instance sets *j*30 and *j*60, and for this reason we have selected it for a detailed comparison with CARA. The comparison of both algorithms cannot be undertaken by limiting the number of sequences generated and evaluated because this limitation is meaningless with CARA. In the first phase, CARA does not generate complete sequences but 'half' sequences, as either the head or the tail of the sequence is already sequenced. Additionally, phases 1 and 2 require different computational efforts to generate a sequence.

To provide a basis for the comparison, we will impose a bound on the computation time and work with the Linux operating system. The program was compiled and run under Linux and the results are shown in Table 10. The data shown in Table 11 was produced using the code supplied by Dr. Hartmann and described in Hartmann (1998). Using this code, the maximum number of sequences can be fixed. To establish this maximum we increased the number of sequences in units of one thousand until the time taken passed the time taken by CARA.

The results show that both algorithms are similar for *j*30 and *j*60. The behaviour of CARA is a little better for *j*90 and even better for the *j*120 instances.

The algorithm of Nonobe and Ibaraki (1999) was not included in the aforementioned computational study. Table 12 has been built from the data given by Dr. Nonobe. Note that av_CPU refers to seconds on a Sun Ultra 2 workstation (300 MHz, 1 Gbyte memory). By comparing the columns av_dev in the Tables 3 and 10 we can observe that the differences are small—but CARA is better in all instance sets. With respect

Table 10
CARA results in LINUX

	$\sum$CARA	av_dev	av_CPU	CPM_dev
*j*120	76367	1.9246	48.50	34.55
*j*90	46247	0.6118	12.88	11.1186
*j*60	38662	0.4805	7.74	11.4317
*j*30	28337	0.0560	3.95	13.4597

Table 11
Results of the Hartman algorithm with limited execution time

	$\sum$	av_dev	av_CPU	CPM_dev
*j*120	76924	2.6509	48.86	35.5498
*j*90	46295	0.7207	13.10	11.2482
*j*60	38660	0.4965	8.06	11.4270
*j*30	28349	0.0908	3.95	13.5127

Table 12
Computational results for the Nonobe and Ibaraki algorithm

	$\sum$	av_dev	av_CPU	best_sol	CPM_dev
*j*120	76600	2.2858	645.33	217	34.99
*j*90	46294	0.7057	181.41	368	11.25
*j*60	38697	0.5587	26.49	370	11.55
*j*30	28337	0.0578	9.07	463	13.46

to the two columns av_CPU, the times refer to different computers, yet we can conclude that CARA is faster than the Nonobe and Ibaraki algorithm—especially in resolving the *j*90 and *j*120 sets. If we compare the two best_sol columns we can see that CARA obtains the same number of best solutions as Nonobe and Ibaraki for the instance sets *j*30 and *j*90; and more for *j*60 and less for *j*120.

8. Concluding remarks

We have presented a metaheuristic algorithm, CARA, for solving the RCPSP. It can be considered as a non-standard implementation of tabu search principles. The algorithm makes use of the TO representation of schedules. We consider two phases in the application of the algorithm. Phase 1 makes use of temporal information, such as: the total slack of an activity relative to a given schedule S; the set Advance(S); and the latest finish time or the dynamic slack. Phase 2 is based on two new multi-pass sampling methods (the β biased random sampling method and the window random sampling method) adequately defined to make controlled moves. As far as we know, the utilisation of sampling methods as a probabilistic local search mechanism is an innovation.

Computational experiments show that the quality of CARA is quite good and that computational times are short. They also show that CARA can compete with state-of-the-art heuristics for the RCPSP. Furthermore, as a by-product, it has been shown that CARA outperforms a pool of nine of the best priority rules and uses a reasonable computation time.

Several strategies implemented in CARA contribute to these good results. The strategic utilisation of probabilities induces variety, compensates for the lack of complete information, and makes reliance on memory structures unnecessary. By using, whenever possible, two different values for the same parameter, or two different procedures to perform the same task, the algorithm is converted into a robust procedure that can be successfully applied to diversely characterised instance problems without the need for parameter adjustment.

The strategy of alternatively improving the head and tail by maintaining them fixed respectively induces two main effects. Firstly, it intensifies the search by restricting it to the sub-space formed by the TOs that share the same head (tail). Secondly, it contributes to the reduction of computational time because only half the activities have to be scheduled to generate a new schedule.

Finally, combining the forward/backward strategy with the head and tail strategy has been shown to be an effective mechanism for diversifying the search (strategic oscillation).

Acknowledgements

The authors would like to thank Dr. Nonobe for making the computational results of his heuristic available and Dr. Hartmann for providing his run code. We are also very grateful for the constructive comments of the referees who helped make this article more readable.

Appendix A. Move pseudocodes

A.1. LF_BACKWARD_MOVE(γ)

Step 0. Initialisation.
Construct Advance(S). Calculate mino. Construct Head(S) and Tail(S)
$\gamma'_j = \gamma_j \ \forall j \in \text{Head}(\gamma)$; $\gamma'_j = \infty \ \forall j \in \text{Tail}(\gamma)$; $\gamma'_n = n$; $k = n - 1$.
Step 1. Partition of the candidate set.
Define Candidate $= \{j/j \in \text{Tail}(\gamma);\ \gamma'_j = \infty$ and $\gamma'_i < \infty \ \forall i \in S_j\}$
Eligible $= \{j/\gamma_j > k\}$
Divide Candidate into
C_NA = Candidate $\setminus$ Advance(S)
C_E = Candidate $\cap$ Advance(S) $\cap$ Eligible
C_NE = Candidate $\setminus$ (C_NA $\cup$ C_E).
Step 2. Probabilistic set selection.
If C_NA $\cup$ C_E $= \phi$, then SELECT = C_NE. Go to Step 3.
If no, calculate:
$\alpha = \max\{\text{LF}_i/i \in \text{C\_E}\}$ (If C_E $= \phi$, then $\alpha = 0$).
$\beta = \max\{\text{LF}_i/i \in \text{C\_NA}\}$ (If C_NA $= \phi$, then $\beta = 0$).
Select with uniform probability a $r \in (0, 1)$.
If $r \leqslant \alpha/(\alpha + \beta)$ then SELECT = C_E. If no, SELECT = C_NA.
Step 3. Probabilistic activity selection.
If SELECT = C_NE, then j is the activity with the highest order in C_NE.
If SELECT = C_NA, then j is the activity with the highest order in C_NA.
If SELECT = C_E, then j is chosen probabilistically in the following way: a 75% probability that it is the C_E activity with the highest LF, and a 25% probability that it is one of the other C_E activities randomly chosen.
Step 4.
Perform $\gamma'_j = k$; $k = k - 1$.
If $k <$ mino, Return γ'. Stop. If no, go to Step 1.

A.2. DSLACK_FORWARD_MOVE(γ)

Step 0. Initialisation.
Construct Advance(S). Calculate mino. Construct Head(S) and Tail(S)
Schedule in series the activities Head(S) *in increasing order of* γ_j.
Label(j) $= 1 \ \forall j \in \text{Head}(S)$; label($j$) $= 0 \ \forall j \in \text{Tail}(S)$, $k =$ mino.
Step 1. Partition of the candidate set.
Define:
Candidate $= \{j/j \in \text{Tail}(S)\}$; label($j$) $= 0$ and label(i) $= 1 \ \forall i \in P_j\}$
Partition Candidate in
C_A = Candidate $\cap$ Advance(S)
C_NA = Candidate $\setminus$ Advance(S).
Step 2. Probabilistic set selection.
If C_A $= \emptyset$, then SELECT = C_NA. Go to Step 3
Let $i_0 \in$ C_A such that $\gamma_{i_0} = \min\{\gamma_i/i \in \text{C\_A}\}$
If $\gamma_{i_0} \leqslant k + 1$ then $\gamma'_{i_0} = k$. Go to Step 4.
If C_NA $= \emptyset$, then SELECT = C_A. Go to Step 3.

If no, calculate:

$\alpha = \min\{\text{DSLACK}_i;\ i \in \text{C\_A}\}$

$\beta = \min\{\text{DSLACK}_i;\ i \in \text{C\_NA}\}$

2.1. If $\alpha < 0$ and $\beta \geqslant 0$, then SELECT = C_A

2.2. If $\alpha \geqslant 0$ and $\beta < 0$, then SELECT = C_NA

2.3. Otherwise, r to be random in $(0, 1)$:

2.3.1. If $\alpha > 0$ and $\beta > 0$:

If $r \geqslant \alpha/(\alpha + \beta)$, then SELECT = C_A.

Otherwise: SELECT = C_NA

2.3.2. If $\alpha < 0$ and $\beta < 0$:

If $r \leqslant \alpha/(\alpha + \beta)$, then SELECT = C_A.

Otherwise: SELECT = C_NA

2.3.3. If $\alpha = 0$ and $\beta = 0$: Select at random between C_A and C_NA.

Step 3. Probabilistic activity selection.

If SELECT = C_A, then j is the activity with least dynamic slack in C_A.

If SELECT = C_NA, then j is the C_NA activity with least order.

Step 4. Activity sequencing.

Schedule j as early as possible taking into account the precedence relations and the resources used by the activities already scheduled. Label$(j) = 1$ and $k = k + 1$.

Step 5.

If $j = n$, then S will be the constructed schedule. Return $\gamma' = \gamma(S)$. Stop.

Otherwise, go to Step 1.

A.3. WINDOW_SAMPLING_MOVE(γ)

Step 0. Initialisation.

Apply the serial SGS to the activities in Head$(S) = \{j/\gamma_j < n/2\}$ in increasing order of γ_j.

Label$(j) = 1\ \forall j \in$ Head(S); label$(j) = 0\ \forall j \in$ Tail(S)

Step 1. Probabilistic activity selection.

Define:

Candidate $= \{j/j \in \text{Tail}(S);\ \text{label}(j) = 0\ \text{ and }\ \text{label}(i) = 1\ \forall i \in P_j\}$

$w_0 = \min\{\gamma_j/j \in \text{Candidate}\}$

Eligible $= \{j/j \in \text{Candidate and } \gamma_j - w_0 \leqslant \text{window}\}$

Randomly select j in Eligible.

Step 2. Activity sequencing.

Schedule j as soon as possible while bearing in mind the precedence relations and the resources used by the activities already scheduled. Label$(j) = 1$.

Step 3.

If $j = n$, then S' will be the constructed schedule. Return $\gamma' = \gamma(S')$. Stop.

In the contrary case, go to Step 1.

Appendix B. Priority rules for initial population

1. Most total successors (MTS)	(max) $v(j) = \lvert S_j \rvert$
2. Minimum latest start time (LST)	(min) $v(j) = \text{LF}_j - d_j$
3. Minimum latest finish time (LFT)	(min) $v(j) = \text{LF}_j$

4. Greatest rank positional weight (GRPW)	(max) $v(j) = d_j + \sum_{i \in \widetilde{S}_j} d_i$
5. Weighted resource utilisation ratio and precedence (WRUP)	(max) $v(j) = w_1 \lvert\widetilde{S}_j\rvert + w_2 \sum_{k=1}^{K} r_{j,k}/R_k$
6. Minimum slack (MSLK)	(min) $v(j) = \mathrm{LF}_j - \mathrm{EF}_j$
7. Worst case slack (WCS)	(min) $v(j) = \mathrm{LF}_j - d_j - \max_{(i,j)\in \mathrm{AP}}\{E(i,j)\}$
8. Resource scheduling method (RSM)	(min) $v(j) = \max_{(i,j)\in \mathrm{AP}}\{0, t_g + d_j - (\mathrm{LF}_i - d_i)\}$
9. Improved resource scheduling method (IRSM)	(min) $v(j) = \max\{0, E(j,i) - (\mathrm{LF}_i - d_i)\}$

Definitions:

- $\widetilde{S}_j$ is the immediate successors of activity j.
- EF_j is the earliest finish time of activity j.
- t_g is the schedule time at stage g of the parallel SGS.
- $\mathrm{AP} = \{(i,j) \in D_g x D_g / i \neq j\}$, where D_g is the eligible set at stage g of the parallel SGS.
- $E(i,j)$ is the earliest time to schedule activity j if activity i is started at the schedule time t_g.
- w_1 and $w_2 \geqslant 0$, $w_1 + w_2 = 1$; specifically, $w_1 = 0.3$ and $w_2 = 0.7$ have been fixed.

References

Blazewicz, J., Lenstra, J.K., Rinooy Kan, A.H.G, 1983. Scheduling subject to resource constraints: Classification and complexity. Discrete Applied Mathematics 5, 11–24.

Bean, J.C., 1994. Genetic algorithms and random keys for sequencing and optimization. ORSA Journal on Computing 6 (2), 154–160.

Bouleimen, K., Lecocq, H., 1998. A new efficient simulated annealing algorithm for the resource-constrained project scheduling problem, Technical Report, Service de Robotique et Automatisation, Université de Liège.

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

Brucker, P., Knust, S., Schoo, A., Thiele, O., 1998. A branch & bound algorithm for the resource-constrained project scheduling problem. European Journal of Operational Research 107, 272–288.

Cho, J.H., Kim, Y.D., 1997. A simulated annealing algorithm for resource constrained project scheduling problems. Journal of Operational Research Society 48, 736–744.

Demeulemeester, E., Herroelen, W., 1992. A branch-and-bound procedure for the multiple resource-constrained project scheduling problem. Management Science 38, 1803–1818.

Glover, F., Laguna, M., 1997. Tabu Search. Kluwer Academic Publishers.

Hartmann, S., 1998. A competitive genetic algorithm for resource-constrained project scheduling. Naval Research Logistics 45, 733–750.

Hartmann, S., Kolisch, R., 2000. Experimental evaluation of state-of-the-art heuristics for the resource-constrained project scheduling problem. European Journal of Operational Research 127 (2), 394, 407.

Herroelen, W., De Reyck, B., Demeulemeester, E., 1998. Resource-constrained project scheduling: A survey of recent developments. Computers and Operations Research 25 (4), 279–302.

Icmeli, O., Erenguc, S.S., Zappe, C.J., 1993. Project scheduling problems: A survey. International Journal of Operations & Production Management 13 (11), 80–91.

Klein, R., Scholl, A., 1999. Computing lower bounds by destructive improvement: An application to resource-constrained project scheduling. European Journal of Operational Research 112, 322–346.

Kolisch, R., Hartmann, S., 1999. Heuristic algorithms for solving the resource-constrained project scheduling problem: Classification and computational analysis. In: Weglarz, J. (Ed.), Project Scheduling. Recent Models, Algorithms and Applications. Kluwer Academic Publishers, Boston, pp. 147–178.

Kolisch, R., Padman, R., 1997. An integrated survey of project scheduling, Technical report 463, Manuskripte aus den Instituten für Betriebswirtschaftslehre der Universität Kiel.

Kolish, R., Sprecher, A., 1997. PSPLIB—A project scheduling library. European Journal of Operational Research 96, 205–216. Available from <http://www.bwl.uni-kiel.de/Prod/psplib/index.html>.

Kolish, R., Sprecher, A., Drexl, A., 1995. Characterization and generation of a general class of resource-constrained project scheduling problems. Management Science 41 (10), 1693–1703.

Lee, J.K., Kim, Y.D., 1996. Search heuristics for resource constrained project scheduling. Journal of the Operational Research Society 47, 678–689.

Leon, V.J., Ramamoorthy, B., 1995. Strength and adaptability of problem-space based neighborhoods for resource-constrained scheduling. OR Spektrum 17, 173–182.

Li, R.K.-Y., Willis, J., 1992. An iterative scheduling technique for resource-constrained project scheduling. European Journal of Operational Research 56, 370–379.

Mingozzi, A., Maniezzo, V., Ricciardelli, S., Bianco, L., 1998. An exact algorithm for project scheduling with resource constraints based on a new mathematical formulation. Management Science 44, 714–729.

Naphade, K.S., Wu, S.D., Storer, R.H., 1997. Problem space search algorithms for resource-constrained project scheduling. Annals of Operations Research 70, 307–326.

Nonobe, K., Ibaraki, T., 1999. Formulation and Tabu search algorithm for the resource constrained project scheduling problem (RCPSP), Technical report 99010.

Özdamar, L., Ulusoy, G., 1995. A survey on the resource-constrained project scheduling problem. AIIE Transactions 27, 574–586.

Özdamar, L., Ulusoy, G., 1996. An iterative local constraint based analysis for solving the resource constrained project scheduling problem. Journal of Operations Management 14, 193–208.

Sprecher, A., 1996. Solving the RCPSP efficiently at modest memory requirements, Technical report 425, Manuskripte aus den Instituten für Betriebswirtschaftslehre der Universität Kiel.

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

Storer, R.H., Wu, S.D., Vaccari, R., 1992. New search paces for sequencing problems with application to job shop scheduling. Management Science 38, 1495–1509.

Valls, V., Laguna, M., Lino, P., Pérez, A., Quintanilla, S, 1999. Project Scheduling with Stochastic Activity Interruptions. In: Weglarz, J. (Ed.), Project Scheduling. Recent Models, Algorithms and Applications. Kluwer Academic Publishers, Boston, pp. 333–354.