A random key based genetic algorithm for the resource constrained project scheduling problem

J.J.M. Mendes, J.F. Gonçalves, M.G.C. Resende

Abstract

This paper presents a genetic algorithm for the Resource Constrained Project Scheduling Problem (RCPSP). The chromosome representation of the problem is based on random keys. The schedule is constructed using a heuristic priority rule in which the priorities of the activities are defined by the genetic algorithm. The heuristic generates parameterized active schedules. The approach was tested on a set of standard problems taken from the literature and compared with other approaches. The computational results validate the effectiveness of the proposed algorithm.

Keywords

Project management; Scheduling; Genetic algorithms; Random keys; RCPSP

1. Introduction

The *Resource Constrained Project Scheduling Problem* (RCPSP) can be stated as follows. A project consists of $n+2$ activities where each activity has to be processed in order to complete the project. Let $J=\{0, 1, \ldots, n, n+1\}$ denote the set of activities to be scheduled and $K=\{1, \ldots, k\}$ the set of resources. Activities 0 and $n+1$ are dummies, have no duration, and represent the initial and final activities. The activities are interrelated by two kinds of constraints:

(1) *precedence constraints* force each activity j to be scheduled after all predecessor activities P_j are completed;
(2) activities require resources with *limited capacities*.

While being processed, activity j requires $r_{j,k}$ units of resource type $k \in K$ during every time instant of its non-preemptable duration d_j. Resource type k has a limited capacity of R_k at any point in time. The parameters d_j, $r_{j,k}$, and R_k are assumed to be integer, non-negative, and deterministic. For the project start and end activities, we have $d_0 = d_{n+1} = 0$ and $r_{0,k} = r_{n+1,k} = 0$, for all $k \in K$. The problem consists in finding a schedule of the activities, taking into account the resources and the precedence constraints, that minimizes the makespan $C_{\max}$.

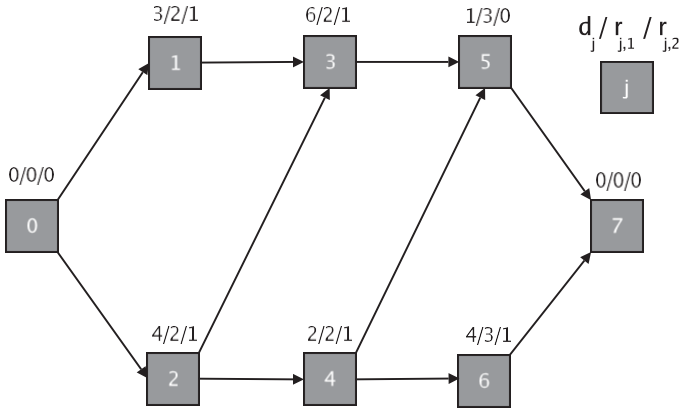

Fig. 1. Project network example. Activities are represented as boxes and precedences by directed arcs. Parameters $d_j/r_{j,1}/r_{j,2}$ are given for each activity j.

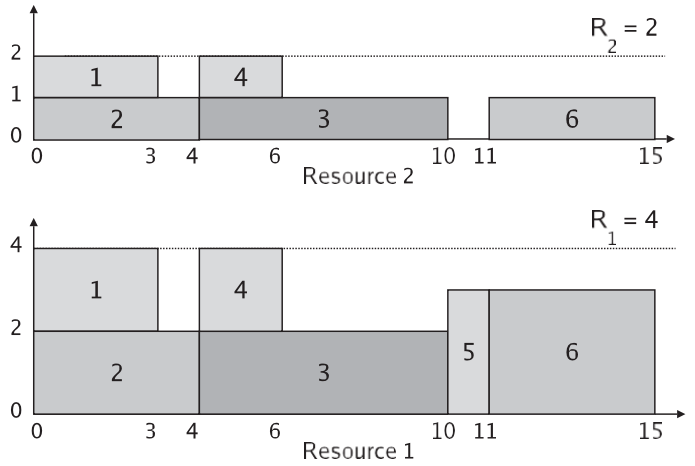

Fig. 2. Feasible schedule with an optimal makespan of 15 for the project network example of Fig. 1.

Let F_j represent the finish time of activity j. A schedule can be represented by a vector of finish times $(F_1, \ldots, F_m, \ldots, F_{n+1})$. Fig. 1 shows an example of a project comprising $n=6$ activities which have to be scheduled, subject to two renewable resource types with a capacity of four and two units, respectively. A feasible schedule with an optimal makespan of 15 time-periods is represented in Fig. 2.

Several exact methods to solve the RCPSP are proposed in the literature. Currently, the most competitive exact algorithms seem to be the ones of Demeulemeester and Herroelen [1], Brucker et al. [2], Klein and Scholl [3,4], Mingozzi et al. [5], and Sprecher [6]. Stork and Uetz [7] present several complexity results related to generation and counting of all circuits of an independence system, and study their relevance in the solution of RCPSP.

It has been shown by Blazewicz et al. [8] that the RCPSP, as a generalization of the classical job shop scheduling problem, belongs to the class of NP-hard optimization problems, therefore justifying the use of heuristics when solving large problem instances.

Several authors propose procedures for computing lower bounds on the makespan. Demassey et al. [9] propose a cooperation method between constraint programming and integer programming. Brucker and Knust [10] present a destructive lower bound for the multi-mode resource-constrained project scheduling problem with minimal and maximal time-lags. Brucker and Knust [11] developed a destructive lower bound for the RCPSP, where the lower bound calculations are based on two methods for proving infeasibility of a given threshold value for the makespan. The first uses constraint propagation techniques, while the second is based on a linear programming formulation.

Most of the heuristics methods used for solving resource-constrained project scheduling problems either belong to the class of priority rule based methods or to the class of metaheuristic based approaches [12]. The first class of methods starts with none of the jobs being scheduled. Subsequently, a single schedule is constructed by selecting a subset of jobs in each step and assigning starting times to these jobs until all jobs have been considered. This process is controlled by the scheduling scheme as well as priority rules with the latter being used for ranking the jobs. Several approaches of this class have been proposed in the literature, e.g. [12–22]. The second class of methods improves upon an initial solution. This is done by successively executing operations which transform one or several solutions into others. Several approaches of this class have been proposed in the literature, e.g. genetic algorithms [23–32], simulated annealing [33–35], tabu search [36–40], local search-oriented approaches [41,42], and population-based approaches [30,43].

Some surveys are provided by Icmeli et al. [44], Herroelen et al. [45], Brucker et al. [46], Klein [47], Kolisch and Hartmann [12], Hartmann and Kolisch [48], Kolisch and Padman [49], and Demeulemeester and Herroelen [50]. Kolisch and Hartmann [51] and Brucker and Knust [52] present models and algorithms for complex scheduling problems and discuss the RCPSP.

In this paper, we present a new genetic algorithm for finding cost-effective solutions for the RCPSP. The remainder of the paper is organized as follows. Section 2 presents the different classes of schedules. In Section 3, we present our hybrid approach to solve the RCPSP: genetic algorithm and schedule generation procedure. Section 4 reports computational results, and concluding remarks are made in Section 5.

2. Types of schedules

Schedules can be classified into one of the following three types of schedules:

(1) *Semi-active schedules.* These are feasible schedules obtained by sequencing activities as early as possible. In a semi-active schedule, no activity can be started earlier without altering the processing sequences.
(2) *Active schedules.* These are feasible schedules in which no activity could be started earlier without delaying some other activity or breaking a precedence constraint. Active schedules are also semi-active schedules. An optimal schedule is always active, so the search space can be safely limited to the set of all active schedules.
(3) *Non-delay schedules.* These are feasible schedules in which no resource is kept idle when it could start processing some activity. Non-delay schedules are active and hence are also semi-active.

Later in this paper we extend the use of parameterized active schedules as proposed in Gonçalves and Beirão [53] and Gonçalves et al. [54]. This type of schedule consists of schedules in which no resource is kept idle for more than a predefined period if it could start processing some activity. If the predefined period is set to zero, then we obtain a non-delay schedule. The basic concepts of this type of schedule are presented in the next section.

2.1. Parameterized active schedules

As mentioned above, the optimal schedule is in the set of all active schedules. However, the set of active schedules is usually very large and contains many schedules with relatively large delay times, hence with poor quality in terms of makespan. To reduce the solution space and to control the delay times, we use the concept of parameterized active schedules.

Fig. 3 illustrates where the set of parameterized active schedules is located relative to the class of semi-active, active, and non-delay schedules. By controlling the maximum delay time allowed, one can reduce or increase the size of the solution space. A maximum delay time equal to zero is equivalent to restricting the solution space to non-delay schedules. Section 3.2 presents the pseudo-code to generate parameterized active schedules.

3. New approach

The new approach combines a genetic algorithm and a schedule generator procedure that generates parameterized active schedules and a novel measure of merit that computes a *modified makespan* value which is used as fitness measure

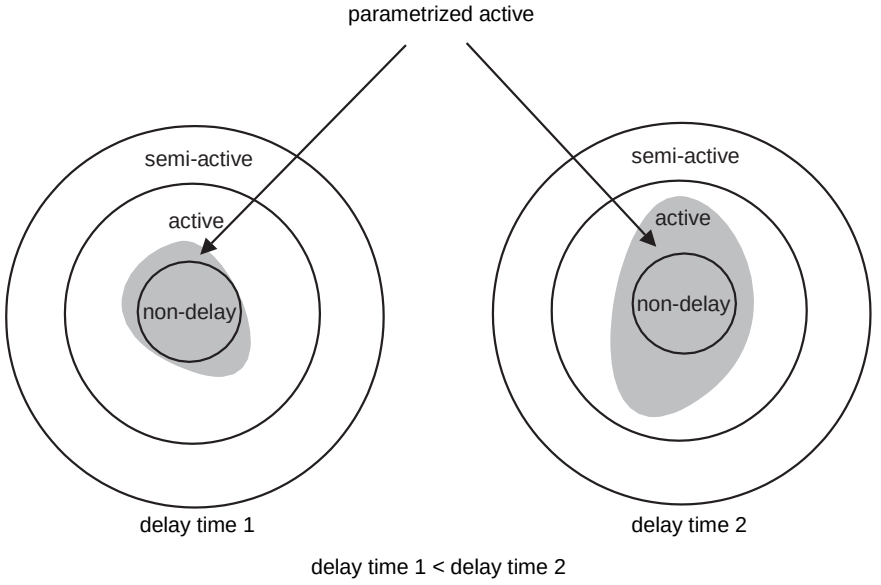

Fig. 3. Parameterized active schedules for different values of the delay time.

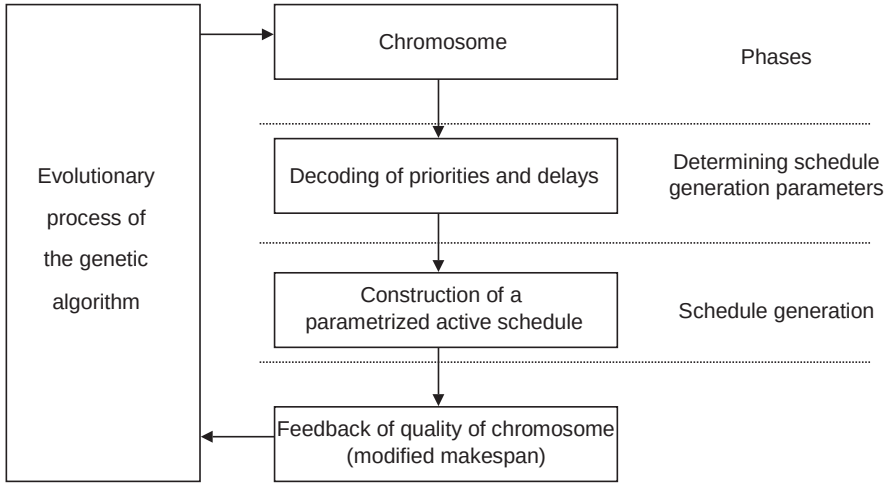

Fig. 4. Architecture of the new approach.

(quality measure) of feedback to the genetic algorithm. The genetic algorithm evolves the chromosomes which represent the priorities of the activities and delay times. For each chromosome the following two phases are applied:

(1) *Decoding of priorities, delay times.* This phase is responsible for transforming the chromosome supplied by the genetic algorithm into the priorities of the activities and delay times.
(2) *Schedule generation.* This phase makes use of the priorities and the delay times defined in the first phase and constructs parameterized active schedules.

After a schedule is obtained, the corresponding measure of quality (*modified makespan*) is feedback to the genetic algorithm. Fig. 4 illustrates the sequence of steps applied to each chromosome generated by the genetic algorithm. Details about each of these phases will be presented in the next sections.

3.1. Genetic algorithm

Genetic algorithms are adaptive methods, which may be used to solve search and optimization problems [55]. Before a genetic algorithm can be run, a suitable *encoding* (or representation) for the problem must be devised. A *fitness function* is also required, which assigns a figure of merit to each encoded solution. During the execution of the algorithm, parents must be *selected* for reproduction, and *recombined* to generate offspring.

It is assumed that a potential solution to a problem may be represented as a set of parameters. These parameters (known as *genes*) are joined together to form a string of values (*chromosome*). The set of parameters represented by a particular chromosome is referred to as an *individual*. The fitness of an individual depends on its chromosome and is evaluated by the fitness function.

The individuals, during the reproductive phase, are selected from the population and recombined, producing offspring, which comprise the next generation. Parents are randomly selected from the population using a scheme, which favors fitter individuals. Having selected two parents, their chromosomes are recombined, typically using mechanisms of *crossover* and *mutation*. Mutation is usually applied to some individuals to guarantee population diversity.

3.1.1. Chromosome representation

The genetic algorithm described in this paper uses a random-key alphabet which is comprised of real random numbers between 0 and 1. The evolutionary strategy used is similar to the one proposed by Bean [56], the main difference occurring in the crossover operator. The important feature of random keys is that all offspring formed by crossover are feasible solutions. This is accomplished by moving much of the feasibility issue into the objective function evaluation. If any random-key vector can be interpreted as a feasible solution, then any crossover vector is also feasible. Through the dynamics of the genetic algorithm, the system *learns* the relationship between random-key vectors and solutions with good objective function values.

A chromosome represents a solution to the problem and is encoded as a vector of random keys. In a direct representation, a chromosome represents a solution of the original problem, and is called *genotype*, while in an indirect representation it does not and special procedures are needed to derive a solution from it called *phenotype*.

In the present context, the direct use of schedules as chromosomes is too complicated to represent and manipulate. In particular, it is difficult to develop corresponding crossover and mutation operations. Instead, solutions are represented indirectly by parameters that are later used by a schedule generator to obtain a solution. To obtain the solution, we use the parameterized active schedule generator described in Section 3.2.

Each solution chromosome is made of $2n$ genes, where n is the number of activities.

$$Chromosome = (\underbrace{gene_1, \ldots, gene_n}_{\text{priorities}}, \underbrace{gene_{n+1}, \ldots, gene_{2n}}_{\text{delay times}}).$$

The first n genes are used to determine the priority of each of the n activities. The genes between $n+1$ and $2n$ are used to determine the delay times used at each of the n iterations of the scheduling procedure.

3.1.2. Decoding the priorities of the activities

In an earlier version of the algorithm, the priorities of the activities were given directly by the genetic algorithm, i.e.

$$PRIORITY_j = gene_j \quad \text{for all } j = 1, \ldots, n.$$

Though this approach worked quite well, we conjectured that it could be improved if we could somehow provide the genetic algorithm with information about the structure of the problem. The priority decoding expression used in the current version of the algorithm combines what we consider the *ideal* priority under infinite capacity and a factor that corrects this value to account for the real resource load and capacity availability. For the ideal priority under infinite capacity we use the term LLP_j/LCP, where LLP_j is the longest-length path from the beginning of activity j to the end of the project and LCP is length along the critical path of the project. It is clear that $0 \leq LLP_j/LCP \leq 1$.

The factor that adjusts the priority to account for capacity is given by $(1 + gene_j)/2$. Therefore, the final decoding expression is given by

$$PRIORITY_j = \frac{LLP_j}{LCP} \times \left[\frac{1 + gene_j}{2}\right], \quad j = 1, \ldots, n.$$

3.1.3. Decoding the delay times

The genes between positions $n + 1$ and $2n$ are used to determine the delay times used when scheduling an activity. The delay time $delay_g$ used by each scheduling iteration g is given by

$$delay_g = gene_{n+g} \times 1.5 \times maxDur,$$

where *maxDur* is the maximum duration of all activities. The factor 1.5 was obtained after some experimental tuning.

3.1.4. Fitness function

Different schedules can have the same value of the makespan. However, the potential for improvement of schedules with the same makespan value might not be the same. To differentiate the potential for improvement between schedules having the same makespan we developed a new measure of fitness, that we call *modified makespan*.

The modified makespan combines the makespan of the schedule with a measure of the potential for improvement of the schedule. This measure of potential for improvement of the schedule will have values in the interval $]0, 1[$. The rationale for this new measure is that if we have two schedules with the same makespan value, then the one with a smaller number of activities ending close to the makespan will have more potential for improvement.

To define the modified makespan, we introduce the concept of the distance of one activity to the final activity (activity $n+1$). The distance $dist(j)$ of activity j to activity $n+1$ is equal to the number of activities in the path connecting them that has the smallest number of activities, including activity j and excluding activity $n+1$.

The *modified makespan* of distance L is given by

$$F_{n+1} + \frac{\sum_{d=1}^{L} \sum_{i|dist(i)=d} F_i}{\sum_{d=1}^{L} \sum_{i|dist(i)=d} F_{n+1}}. \tag{1}$$

Note that when $L \to \infty$, we get a modified makespan equal to the usual makespan. We will use the project network example given in Fig. 1 and the schedule presented in Fig. 2 to illustrate the calculation of the modified makespan. The distances of activities 1–6 are

activity i	1	2	3	4	5	6
$dist(i)$	3	3	2	2	1	1

The makespan of the project is 15 (see Fig. 2). The modified makespan of distance $L = 1$ is

$$15 + \frac{(15 + 11)}{(15 + 15)} = 15.867.$$

The modified makespan of distance $L = 2$ is

$$15 + \frac{(15 + 11) + (6 + 10)}{(15 + 15) + (15 + 15)} = 15.7.$$

3.1.5. Evolutionary strategy

To breed good solutions, the random-key vector population is operated upon by a genetic algorithm. Many variations of genetic algorithms can be obtained by varying the reproduction, crossover, and mutation operators. The reproduction and crossover operators determine which parents will have offspring, and how genetic material is exchanged between the parents to create those offspring. Reproduction and crossover operators tend to increase the quality of the populations and force convergence. Mutation opposes convergence since it allows for random alteration of genetic material.

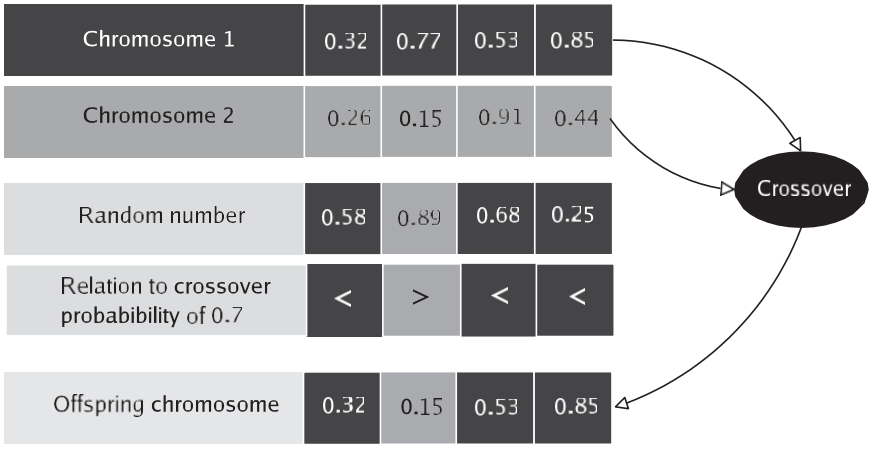

Fig. 5. Example of parameterized uniform crossover with crossover probability equal to 0.7. For each gene, a random number in the interval $[0, 1]$ is generated. With probability 0.7 the offspring inherits the gene of Chromosome 1 and with probability 0.3, it inherits the one of Chromosome 2.

Given a current population, we perform the following three steps to obtain the next generation:

(1) *Reproduction.* Some of the best individuals are copied from the current generation into the next (see *TOP* in Fig. 6). This strategy is called *elitist* [57] and its main advantage is that the best solution is monotonically improving from one generation to the next. However, it can lead to a rapid population convergence to a local minimum. Nevertheless, this can be overcome by using high mutation rates as described below.

(2) *Crossover.* Regarding the crossover operator, parameterized uniform crossovers [58] are used as opposed to the traditional one-point or two-point crossover. Two individuals are randomly chosen to act as parents. One of the parents is chosen amongst the best individuals in the population (*TOP* in Fig. 6), while the other is randomly chosen from the whole current population (including *TOP*). For each gene, a real random number in the interval $[0, 1]$ is generated. If the random number obtained is smaller than a threshold value, called *crossover probability* (*CProb*), for example 0.7, then the allele of the first parent is used. Otherwise, the allele used is that of the second parent. An example of a crossover outcome is given in Fig. 5.

(3) *Mutation.* In this scheme, mutation is used in a broader sense than usual. The operator we define acts like a mutation operator and its purpose is to prevent premature convergence of the population. Instead of performing gene-by-gene mutation, with very small probability at each generation, we introduce some new individuals into the next generation (see *BOT* in Fig. 6). These new individuals (mutants) are randomly generated from the same distribution as the original population and thus, no genetic material of the current population is brought in. This process prevents premature convergence of the population, like in a mutation operator, and leads to a simple statement of convergence, i.e. if a sufficiently large number of generations is carried out, then the entire solution space will be sampled.

Fig. 6 depicts the transitional process between two consecutive generations.

The initial population is randomly generated. However, to ensure that some non-delay solutions are included in the initial population, we changed the corresponding delay genes of some chromosomes to zero to make the delay equal to zero, i.e. non-delay. The percentage of non-delay chromosomes in the initial population was set to 25% after some experimentation on a small set of problems.

3.2. Schedule generation procedure

The procedure used to construct parameterized active schedules is based on a scheduling generation scheme that does time incrementing. For each iteration g, there is a scheduling time t_g. The active set comprises all activities which

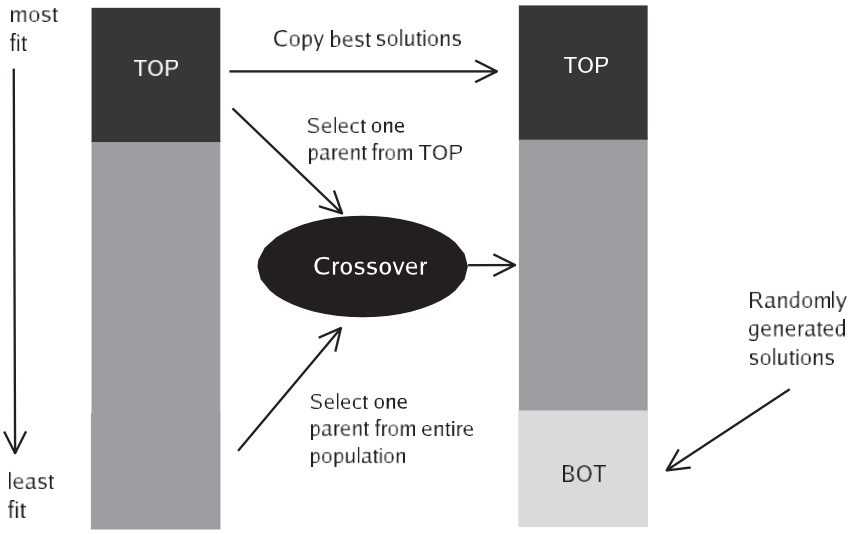

Fig. 6. Transitional process between consecutive generations. Current population is sorted from best to worst fitness. Top (*TOP*) individuals from current population are copied unchanged to next population. Bottom (*BOT*) individuals in next population are randomly generated. The remaining individuals are generated by applying crossover operator to randomly selected individual from *TOP* individuals of current population and randomly selected individual from the entire current population.

are active at t_g, i.e

$$A_g = \{j \in J | F_j - d_j \leqslant t_g < F_j\}.$$

The remaining resource capacity of resource k at instant time t_g is given by

$$RD_k(t_g) = R_k(t_g) - \sum_{j \in A_g} r_{j,k}.$$

S_g comprises all activities which have been scheduled up to iteration g, and F_g comprises the finish times of the activities in S_g. Let $delay_g$ be the delay time associated with iteration g, and let E_g comprise all activities which are precedence feasible in the interval $[t_g, t_g + delay_g]$, i.e.

$$E_g = \{j \in J \backslash S_{g-1} | F_i \leqslant t_g + delay_g, \text{ for } i \in P_j\}.$$

The algorithmic description of the scheduling generation scheme used to generate parameterized active schedules is shown in the pseudo-code in Fig. 7. The makespan of the solution is given by the maximum of all predecessor activities of activity $n + 1$, i.e. $F_{n+1} = \max\{F_l | l \in P_{n+1}\}$.

The basic idea of parameterized active schedules is incorporated in the selection step of the procedure,

$$j^* = \underset{j \in E_g}{\operatorname{argmax}}\{PRIORITY_j\}.$$

The set E_g is responsible for forcing the selection to be made only amongst activities which will have a delay smaller or equal to the maximum allowed delay. The parameters $PRIORITY_j$ and $delay_g$ (priority of activity j and delays used at each g) are supplied by the genetic algorithm.

For *nGen* generations this procedure generates

$$nCrom + [nCrom \times (1 - TOP)] \times (nGen - 1)$$

parameterized active schedules, where *nCrom* is the number of chromosomes and *TOP* is the percentage of previous population copied to the next generation.

```
procedure CONSTRUCT-SCHEDULES(E_1)
1    Initialize g ← 1; t_1 ← 0; A_0 ← 0 ; Γ_o ← 0 ; S_0 ← 0 ;
2    for k ∈ K do initialize RD_k(0) ← R_k;
3    while S_g < n+2 do
4        while E_g ≠ ∅ do    // update E_g
5            // select activity with highest priority
·            j* ← argmax PRIORITY_j  j ∈ E_g ;
6            // compute earliest finish time (in terms of precedence only)
·            EF_j* ← max F_i  i ∈ P_j  + d_j*;
7            // compute earliest finish time (in terms of precedence and capaciy)
·            F_j* ← min  t ∈ [EF_j* − d_j*, ∞] ∩ Γ_g  r_j*,k ≤ RD_k(τ),
·                                  k ∈ K  r_j*,k > 0, τ ∈ [t, t + d_j*]  + d_j*;
8            Update S_g ← S_g−1 ∪ j* ; Γ_g ← Γ_g−1 ∪ F_j* ;
9            g ← g + 1;    // increment iteration counter
10           Update A_g and E_g;
11           Update RD_k(t)  t ∈ [F_j* − d_j* , F_j*] , k ∈ K  r_j*,k > 0 ;
12       end while
13       // determine time associated with activity g
·        t_g ← min t ∈ Γ_g−1  t > t_g−1 ;
14   end while
end CONSTRUCT-SCHEDULES;
```

Fig. 7. Pseudo-code of parametrized active schedule construction procedure.

4. Computational experiments

This section presents results of the computational experiments done with the algorithm proposed in this paper. We call this algorithm *GAPS* (*G*enetic *A*lgorithm for *P*roject *S*cheduling). The algorithm was implemented in Visual Basic 6.0 and the tests were carried out on a computer with a 1.333 GHz AMD Thunderbird CPU on the MS Windows Me operating system.

To illustrate its effectiveness, we consider a total of 1560 instances from three classes of standard RCPSP test problems: J30 (480 instances each with 30 activities), J60 (480 instances each with 60 activities), and J120 (600 instances each with 120 activities). All problem instances require four resource types. Instance details are described by Kolisch et al. [59]. The modified makespan with distance $L = 2$ was used because it gave the best results in a small pilot test.

The proposed algorithm is compared with the following algorithms:

(1) Local search-oriented approaches:
   - Fleszar and Hindi [41]
   - Palpant et al. [42]

(2) Population-based approaches:
   - Debels et al. [43]
   - Valls et al. [60]

(3) Problem and heuristic space:
   - Leon and Ramamoorthy [23]

(4) Priority-rule based sampling methods:
   - Tormos and Lova [22]—sampling LFT, forward–backward improvement (FBI)
   - Schirmer and Riesenberg [61]
   - Kolisch and Drexl [62]
   - Kolisch [20]—single pass LFT (serial)
   - Kolisch [20]—single pass LFT (parallel)

- Kolisch [19,20]—single pass WCS
- Kolisch [63]—random (serial)
- Kolisch [63]—random (parallel)

(5) Genetic algorithms:
- Valls et al. [31]—GA—forward–backward improvement (FBI)
- Debels and Vanhoucke [32]—GA—DBH
- Valls et al. [30]—GA—hybrid, forward–backward improvement (FBI)
- Kochetov and Stolyar [28]—GA, tabu search, path-relinking
- Hartmann [27]—GA self-adapting
- Hartmann [25]—GA activity list
- Hartmann [25]—GA random key
- Hartmann [25]—GA priority rule

(6) Simulated annealing:
- Bouleimen and Lecocq [35]

(7) Tabu search:
- Nonobe and Ibaraki [39]
- Baar et al. [37]

(8) Other type heuristics:
- Möhring et al. [64]—Lagrangian Relaxation Heuristic.

4.1. GA configuration

The present state-of-the-art theory on genetic algorithms provides little insight on how to configure them. In our past experience with genetic algorithms based on the same evolutionary strategy (see e.g. [54,65–68]), we obtained good results with values of *TOP*, *BOT*, and crossover probability (*CProb*) in the following intervals:

Parameter	Interval
TOP	0.10–0.20
BOT	0.15–0.30
CProb	0.70–0.80

For the population size, we obtained good results by indexing it to the size of the problem, i.e. use small size populations for small problems and larger populations for larger problems. Having this past experience in mind and to obtain a reasonable configuration, we conducted a small pilot experiment with combinations of the following values $TOP \in \{0.10, 0.15, 0.20\}$, $BOT \in \{0.15, 0.20, 0.25, 0.30\}$, and $CProb \in \{0.70, 0.75, 0.80\}$. We tried population sizes with 1, 2, 5, and 10 times the number of activities in the project.

The following configuration was held constant for all experiments and all problem instances:

Population size	$5\times$ number of activities in the problem
CProb	0.7
TOP	The top 15% from the previous population chromosomes are copied to the next generation
BOT	The bottom 20% of the population chromosomes are replaced with randomly generated chromosomes
Fitness	Modified makespan ($L = 2$) (to minimize)
Stopping criterion	250 generations

The experimental results demonstrate that this configuration provides high-quality solutions and that it is robust.

4.2. Experimental results

Table 1 shows the CPU time (in s) spent for 250 generations of the genetic algorithm. Tables 2 and 3 (for algorithms reporting the number of schedules generated) and Table 4 (for algorithms not reporting the number of schedules

Table 1
CPU time for 250 generations

Instance class	J30	J60	J120
Average CPU time (s)	5.02	20.11	112.46

Table 2
Average percent deviations from optimal makespan—ProGen set $J = 30$

Algorithm	SGS	Reference	Maximum number of schedules Number of generations of GAPS (Number of schedules of GAPS)		
			1000 7 (915)	5000 39 (4976)	50,000 250 (31,773)
GA, TS—path relinking	Serial	Kochetov and Stolyar [28]	0.10	0.04	**0.00**
GAPS	Param. active	This paper	**0.06**	**0.02**	0.01
Scatter Search—FBI	Serial	Debels et al. [43]	0.27	0.11	0.01
GA—DBH	Serial	Debels and Vanhoucke [32]	0.15	0.04	0.02
GA—hybrid, FBI	Serial	Valls et al. [30]	0.27	0.06	0.02
GA—FBI	Serial	Valls et al. [31]	0.34	0.20	0.02
Sampling—LFT, FBI	Both	Tormos and Lova [22]	0.25	0.13	0.05
TS—activity list	Serial	Nonobe and Ibaraki [39]	0.46	0.16	0.05
GA—self-adapting	Serial	Hartmann [27]	0.38	0.22	0.08
GA—activity list	Serial	Hartmann [25]	0.54	0.25	0.08
SA—activity list	Serial	Bouleimen and Lecocq [35]	0.38	0.23	–

For each algorithm, the table lists its schedule generation scheme (SGS), the reference in which the results are published, and the average percent deviations for 1000, 5000, and 50,000 schedules. The table also lists the number of GAPS generations as well as the number of schedules generated by GAPS. Continues on Table 3.
The values in bold correspond to best known solutions.

Table 3
Average percent deviations from optimal makespan—ProGen set $J = 30$

Algorithm	SGS	Reference	Maximum number of schedules Number of generations of GAPS (Number of schedules of GAPS)		
			1000 7 (915)	5000 39 (4976)	50,000 250 (31,773)
Sampling—adaptative	Both	Schirmer and Riesenberg [61]	0.65	0.44	–
TS—schedule scheme	Related	Baar et al. [37]	0.86	0.44	–
Sampling—adaptative	Both	Kolisch and Drexl [62]	0.74	0.53	–
Sampling—LFT	Serial	Kolisch [20]	0.83	0.53	0.27
GA—random key	Serial	Hartmann [25]	1.03	0.56	0.23
Sampling—random	Serial	Kolisch [63]	1.44	1.00	0.51
GA—priority rule	Serial	Hartmann [25]	1.38	1.12	0.88
Sampling—WCS	Parallel	Kolisch [19,20]	1.40	1.28	–
Sampling—LFT	Parallel	Kolisch [20]	1.40	1.29	1.13
Sampling—random	Parallel	Kolisch [63]	1.77	1.48	1.22
GA—problem space	Mod. Par.	Leon and Ramamoorthy [23]	2.08	1.59	–

For each algorithm, the table lists its schedule generation scheme (SGS), the reference in which the results are published, and the average percent deviations for 1000, 5000, and 50,000 schedules. The table also lists the number of GAPS generations as well as the number of schedules generated by GAPS. Continued from Table 2.

Table 4
Average percent deviations from optimal makespan—ProGen set $J = 30$

Algorithm	SGS	Reference	CPU time			
			Avg. % deviation	Avg.	Max.	CPU freq.
Decomp. & local opt.	Serial	Palpant et al. [42]	0.00	10.26 s	123.0 s	2.3 GHz
VNS—activity list	Serial	Fleszar and Hindi [41]	0.01	0.64 s	5.9 s	1.0 GHz
Local search—critical	Serial	Valls et al. [30]	0.06	1.61 s	6.2 s	400 MHz
Population-based	Serial	Valls et al. [60]	0.10	1.16 s	5.5 s	400 MHz

For each algorithm, the table lists its schedule generation scheme (SGS), the reference in which the results are published, the average percent deviation, average and maximum CPU times, and the frequency of the CPU used in the experiments.

Table 5
Average percent deviations from critical path lower bound—ProGen set $J = 60$

Algorithm	SGS	Reference	Maximum number of schedules Number of generations of GAPS (Number of schedules of GAPS)			
			1000 3 (808)	5000 19 (4872)	50,000 196 (49,830)	63,546 250 (63,546)
GAPS	Param. active	This paper	11.72	11.04	**10.67**	**10.67**
GA—DBH	Serial	Debels and Vanhoucke [32]	**11.45**	**10.95**	10.68	–
Scatter search—FBI	Serial	Debels et al. [43]	11.73	11.10	10.71	–
GA—hybrid, FBI	Serial	Valls et al. [30]	11.56	11.10	10.73	–
GA, TS—path relinking	Both	Kochetov and Stolyar [28]	11.71	11.17	10.74	–
GA—FBI	Serial	Valls et al. [31]	12.21	11.27	10.74	–
GA—self-adapting	Both	Hartmann [27]	12.21	11.70	11.21	–
GA—activity list	Serial	Hartmann [25]	12.68	11.89	11.23	–
Sampling—LFT, FBI	Both	Tormos and Lova [22]	11.88	11.62	11.36	–
SA—activity list	Serial	Bouleimen and Lecocq [35]	12.75	11.90	–	–
TS—activity list	Serial	Nonobe and Ibaraki [39]	12.97	12.18	11.58	–

For each algorithm, the table lists its schedule generation scheme (SGS), the reference in which the results are published, and the average percent deviations for 1000, 5000, 50,000, and 63,546 schedules. The table also lists the number of GAPS generations as well as the number of schedules generated by GAPS. Continues on Table 6.
The values in bold correspond to best known solutions.

generated) summarize the average percentage deviation from the optimal makespan D_{OPT} for instance set J30. GAPS obtained D_{OPT} 0.01. The number of instances for which our algorithm obtains the optimal solution is 477, i.e. for 99.74% of the instances. The number of generated schedules was 31,773 ($= 5 \times 30 + 249 \times INT(0.85 \times 5 \times 30)$). GAPS ranks first for 1000 and 5000 schedules and ranks second for 50,000 schedules. We use only 31,773 schedules which is the number that corresponds to 250 generations.

Tables 5 and 6 (for algorithms reporting the number of schedules generated) and Table 7 (for algorithms not reporting the number of schedules generated) summarize the average percentage deviation from the well-known critical path-based lower bound (D_{LB}) for instance set J60. These lower bound values are reported by Stinson et al. [69]. For the J60 instances, GAPS obtained $D_{\mathrm{LB}} = 10.67$. The number of generated schedules was 63,546 ($= 5 \times 60 + 249 \times INT(0.85 \times 5 \times 60)$). GAPS outperformed all other heuristics for 50,000 schedules, ranks second for 5000, and ranks fourth for 1000 schedules.

Tables 8 and 9 (for algorithms reporting the number of schedules generated) and Table 10 (for algorithms not reporting the number of schedules generated) summarize the average percentage deviation from the well-known critical path-based lower bound (D_{LB}) for instance set J120. This lower bound values are reported by Stinson et al. [69]. For the J120 instances, GAPS obtained $D_{\mathrm{LB}} = 31.20$. The number of generated schedules was 127,341 ($= 5 \times 120 + 249 \times INT(0.85 \times 5 \times 60)$). GAPS ranks seventh for 1000 schedules and ranks third for 5000 and 50,000 schedules (we use only 49,464 which is the number that corresponds to 250 generations).

Table 6
Average percent deviations from critical path lower bound—ProGen set $J = 60$

Algorithm	SGS	Reference	Maximum number of schedules Number of generations of GAPS (Number of schedules of GAPS)			
			1000 3 (808)	5000 19 (4872)	50,000 196 (49,830)	63,546 250 (63,546)
Sampling—adaptative	Both	Schirmer and Riesenberg [61]	12.94	12.58	–	–
Sampling—adaptative	Both	Kolisch and Drexl [62]	13.51	13.06	–	–
TS—schedule scheme	Related	Baar et al. [37]	13.80	13.48	–	–
GA—random key	Serial	Hartmann [25]	14.68	13.32	12.25	–
GA—priority rule	Serial	Hartmann [25]	13.30	12.74	12.26	–
Sampling—LFT	Parallel	Kolisch [20]	13.59	13.23	12.85	–
Sampling—LFT	Serial	Kolisch [20]	13.96	13.53	12.97	–
Sampling—random	Parallel	Kolisch [63]	14.89	14.30	13.66	–
Sampling—WCS	Parallel	Kolisch [19,20]	13.66	13.21	–	–
Sampling—random	Serial	Kolisch [63]	15.94	15.17	14.22	–
GA—problem space	Mod. Par.	Leon and Ramamoorthy [23]	14.33	13.49	–	–

For each algorithm, the table lists its schedule generation scheme (SGS), the reference in which the results are published, and the average percent deviations for 1000, 5000, 50,000, and 63,546 schedules. The table also lists the number of GAPS generations as well as the number of schedules generated by GAPS. Continued from Table 5.

Table 7
Average percent deviations from critical path lower bound—ProGen set $J = 60$

JJ Algorithm	SGS	Reference	CPU time			
			Avg. % deviation	Avg.	Max.	CPU freq.
Decomp. & local opt.	Serial	Palpant et al. [42]	10.81	38.8 s	223.0 s	2.3 GHz
Population-based	Serial	Valls et al. [60]	10.89	3.7 s	22.6 s	400 MHz
Local search—critical	Serial	Valls et al. [30]	11.45	2.8 s	14.6 s	400 MHz
Lagrangian Relax. heuristic	Both, Mod. parallel	Möhring et al. [64]	15.60	6.9 s	57 s	200 MHz

For each algorithm, the table lists its schedule generation scheme (SGS), the reference in which the results are published, the average percent deviation, average and maximum CPU times, and the frequency of the CPU used in the experiments.

From the above results it is clear that no algorithm dominates GAPS. The approach of Debels et al. [43] is the one that seems to have similar performance. When 50,000 or more schedules are allowed, GAPS seems to have the best performance. Given that GAPS uses an evolutionary strategy that depends on the number of generations, it is not surprising that, for problems with large number of activities, it does not perform so well when only a small number of schedules generated is allowed.

To demonstrate the contribution of the parameterized active generation scheme to the overall performance of GAPS we also run GAPS using the parallel and serial schedule generation schemes (SGSs). Table 11 presents the results. It is clear that in all cases the parameterized active generation scheme performs considerably better.

Additionally, to demonstrate the contribution of the parameter L in the overall performance of GAPS, we ran GAPS using $L = 0, 1, 2, 3$. Table 12 presents the results. It is clear that the results obtained by using the modified makespan outperform, in all cases, the ones obtained by using the makespan $(L = 0)$. Additionally, the results confirm that GAPS obtains the best results using the modified makespan when $L = 2$.

5. Conclusions

This paper presents a genetic algorithm for the resource constrained project scheduling problem. The chromosome representation of the problem is based on random keys. The schedules are constructed using a priority rule in which the

Table 8
Average percent deviations from critical path lower bound—ProGen set $J = 120$

Algorithm	SGS	Reference	Maximum number of schedules Number of generations of GAPS (Number of schedules of GAPS)				
			1000 2 (1109)	5000 9 (4672)	50,000 97 (49,464)	100,000 196 (99,855)	127,341 250 (127,341)
GA—DBH	Serial	Debels and Vanhoucke [32]	34.19	**32.34**	**30.82**	–	–
GA—hybrid, FBI	Serial	Valls et al. [30]	**34.07**	32.54	31.24	–	–
GAPS	Param. active	This paper	35.87	33.03	31.44	31.32	31.20
Scatter search—FBI	Serial	Debels et al. [43]	35.22	33.10	31.57	–	–
GA—FBI	Serial	Valls et al. [31]	35.39	33.24	31.58	–	–
GA, TS—path relinking	Both	Kochetov and Stolyar [28]	34.74	33.36	32.06	–	–
GA—self-adapting	Both	Hartmann [27]	37.19	35.39	33.21	–	–
Sampling—LFT, FBI	Both	Tormos and Lova [22]	35.01	34.41	33.71	–	–
GA—activity list	Serial	Hartmann [25]	39.37	36.74	34.03	–	–
SA—activity list	Serial	Bouleimen and Lecocq [35]	42.81	37.68	–	–	–
TS—activity list	Serial	Nonobe and Ibaraki [39]	40.86	37.88	35.85	–	–

For each algorithm, the table lists its schedule generation scheme (SGS), the reference in which the results are published, and the average percent deviations for 1000, 5000, 50,000, 100,000, and 127,341 schedules. The table also lists the number of GAPS generations as well as the number of schedules generated by GAPS. Continues on Table 9.
The values in bold correspond to best known solutions.

Table 9
Average percent deviations from critical path lower bound—ProGen set $J = 120$

Algorithm	SGS	Reference	Maximum number of schedules Number of generations of GAPS (Number of schedules of GAPS)				
			1000 2 (1109)	5000 9 (4672)	50,000 97 (49,464)	100,000 196 (99,855)	127,341 250 (127,341)
GA—priority rule	Serial	Hartmann [25]	39.93	38.49	36.51	–	–
Sampling—adaptative	Both	Schirmer and Riesenberg [61]	39.85	38.70	–	–	–
Sampling—LFT	Parallel	Kolisch [20]	39.60	38.75	37.74	–	–
Sampling—WCS	Parallel	Kolisch [19,20]	39.65	38.77	–	–	–
Sampling—adaptative	Both	Kolisch and Drexl [62]	41.37	40.45	–	–	–
GA—problem space	Mod. Par.	Leon and Ramamoorthy [23]	42.91	40.69	–	–	–
GA—random key	Serial	Hartmann [25]	45.82	42.25	38.83	–	–
Sampling—LFT	Serial	Kolisch [20]	42.84	41.84	40.63	–	–
Sampling—random	Parallel	Kolisch [63]	44.46	43.05	41.44	–	–
Sampling—random	Serial	Kolisch [63]	49.25	47.61	45.60	–	–

For each algorithm, the table lists its schedule generation scheme (SGS), the reference in which the results are published, and the average percent deviations for 1000, 5000, 50,000, 100,000, and 127,341 schedules. The table also lists the number of GAPS generations as well as the number of schedules generated by GAPS. Continued from Table 8.

priorities are defined by the genetic algorithm. Schedules are constructed using a procedure that generates parameterized active schedules.

The approach was tested on a set of 1560 standard instances taken from the literature and compared with 14 other approaches. The algorithm produced good results when compared with other approaches, therefore validating the effectiveness of the proposed algorithm.

Table 10
Average percent deviations from critical path lower bound—ProGen set $J = 120$

Algorithm	SGS	Reference	CPU time			
			Avg. % deviation	Avg.	Max.	CPU freq.
Population-based	Serial	Valls et al. [60]	31.58	59.4 s	264.0 s	400 MHz
Decomp. & local opt.	Serial	Palpant et al. [42]	32.41	207.9 s	501.0 s	2.3 GHz
Local search—critical	Serial	Valls et al. [30]	34.53	17.0 s	43.9 s	400 MHz
Lagrangian Relax. heuristic	Both, Mod. parallel	Möhring et al. [64]	36.00	72.9 s	654 s	200 MHz

For each algorithm, the table lists its schedule generation scheme (SGS), the reference in which the results are published, the average percent deviation, average and maximum CPU times, and the frequency of the CPU used in the experiments.

Table 11
Average percent deviations using the parallel, serial, and parameterized active schedule generation schemes (SGS)

Prob. set	SGS	Maximum number of schedules						
		1000	5000	31,773	50,000	63,546	100,000	127,341
J30	Parallel	1.23	1.13	1.11	–	–	–	–
	Serial	0.58	0.20	0.14	–	–	–	–
	Param. active	**0.06**	**0.02**	**0.01**	–	–	–	–
J60	Parallel	13.71	12.58	–	12.15	12.12	–	–
	Serial	13.68	12.05	–	11.38	11.35	–	–
	Param. active	**11.72**	**11.04**	–	**10.67**	**10.67**	–	–
J120	Parallel	40.20	37.93	–	34.09	–	33.57	33.45
	Serial	43.22	39.93	–	34.31	–	33.99	33.92
	Param. active	**35.87**	**33.03**	–	**31.44**	–	**31.32**	**31.20**

For problem set J30 the deviation is from the optimal makespan while for sets J60 and J120 it is from the critical path lower bound. The fitness function used is the modified makespan with $L = 2$.
The values in bold correspond to best known solutions.

Table 12
Average percent deviation of GAPS using the modified makespan fitness function with different values of L for increasing number of schedules generated

Prob. set	L	Maximum number of schedules						
		1000	5000	31,773	500,00	63,546	100,000	127,341
J30	0	0.82	0.38	0.29	–	–	–	–
	1	0.76	0.34	0.25	–	–	–	–
	2	**0.06**	**0.02**	**0.01**	–	–	–	–
	3	0.21	0.15	0.07	–	–	–	–
J60	0	13.76	12.63	–	12.45	12.41	–	–
	1	12.89	12.10	–	11.92	11.44	–	–
	2	**11.72**	**11.04**	–	**10.67**	**10.67**	–	–
	3	11.87	11.25	–	10.89	10.88	–	–
J120	0	41.94	40.45	–	38.81	–	38.29	38.11
	1	39.56	37.45	–	35.41	–	34.95	34.67
	2	**35.87**	**34.53**	–	**31.44**	–	**31.32**	**31.20**
	3	37.71	35.99	–	33.71	–	33.22	32.96

For problem set J30 the deviation is from the optimal makespan while for sets J60 and J120 it is from the critical path lower bound.
The values in bold correspond to best known solutions.

Acknowledgments

We would to thank Prof. Rainer Kolisch of the Technical University of Munich, Germany, for supplying the problem sets data.

References

[1] Demeulemeester E, Herroelen W. New benchmark results for the resource-constrained project scheduling problem. Management Science 1997;43:1485–92.
[2] Brucker P, Knust S, Schoo A, Thiele O. A branch and bound algorithm for the resource-constrained project scheduling problem. European Journal of Operational Research 1998;107:272–88.
[3] Klein R, Scholl A. Progress: optimally solving the generalized resource-constrained project scheduling problem. Technical Report, University of Technology, Darmstadt, 1998.
[4] Klein R, Scholl A. Scattered branch and bound: an adaptative search strategy applied to resource-constrained project scheduling problem. Technical Report, University of Technology, Darmstadt; 1998.
[5] Mingozzi A, Maniezzo V, Ricciardelli S, Bianco L. An exact algorithm for project scheduling with resource constraints based on a new mathematical formulation. Management Science 1998;44:714–29.
[6] Sprecher A. Scheduling resource-constrained projects competitively at modest memory requirements. Management Science 2000;46:710–23.
[7] Stork F, Uetz M. On the generation of circuits and minimal forbidden sets. Mathematical Programming 2005;102:185–203.
[8] Blazewicz J, Lenstra JK, Rinnooy Kan AHG. Scheduling subject to resource constraints: classification and complexity. Discrete Applied Mathematics 1983;5:11–24.
[9] Demassey S, Artigues C, Michelon P. Constraint-propagation-base cutting planes: an application to the resource-constrained project scheduling problem. INFORMS Journal of Computing 2005;17:52–65.
[10] Brucker P, Knust S. Lower bounds for resource-constrained project scheduling problems. European Journal of Operational Research 2003;149:302–13.
[11] Brucker P, Knust S. A linear programming and constraint propagation-based lower bound for the RCPSP. European Journal of Operational Research 2000;127:355–62.
[12] Kolisch R, Hartmann S. Heuristic algorithms for solving the resource-constrained project scheduling problem: classification and computational analysis. In: Weglarz J, editor. Handbook on recent advances in project scheduling. Dordrecht: Kluwer Academic Publishers; 1999. p. 147–78.
[13] Alvarez-Valdez R, Tamarit JM. Heuristic algorithms for resource-constrained project scheduling: a review and empirical analysis. In: Slowinski R, Weglarz J, editors. Advances in project scheduling. Amsterdam: Elsevier; 1989. p. 113–34.
[14] Boctor FF. Some efficient multi-heuristic procedures for resource-constrained project scheduling. European Journal of Operational Research 1990;49:3–13.
[15] Cooper DF. Heuristics for scheduling resource-constrained projects: an experimental investigation. Management Science 1976;22:1186–94.
[16] Cooper DF. A note on serial and parallel heuristics for resource-constrained project scheduling. Foundations of Control Engineering 1977;2: 131–3.
[17] Davis EW, Patterson JH. A comparison of heuristic and optimum solutions in resource-constrained project scheduling. Management Science 1975;21:944–55.
[18] Lawrence SR. Resource constrained project scheduling—a computational comparison of heuristic scheduling techniques. Technical Report, Graduate School of Industrial Administration, Carnegie-Mellon University, Pittsburgh; 1985.
[19] Kolisch R. Efficient priority rules for the resource-constrained project scheduling problem. Journal of Operations Management 1996;14: 179–92.
[20] Kolisch R. Serial and parallel resource-constrained project scheduling methods revisite: theory and computation. European Journal of Operational Research 1996;90:320–33.
[21] Tormos P, Lova A. A competitive heuristic solution technique for resource-constrained project scheduling. Annals of Operations Research 2001;102:65–81.
[22] Tormos P, Lova A. Integrating heuristics for resource constrained project scheduling: one step forward. Technical Report, Department of Statistics and Operations Research, Universidad Politecnica de Valencia; 2003.
[23] Leon VJ, Ramamoorthy B. Strength and adaptability of problem-space based neighborhoods for resource constrained scheduling. Operations Research Spektrum 1995;17:173–82.
[24] Lee J-K, Kim Y-D. Search heuristics for resource constrained project scheduling. Journal of the Operational Research Society 1996;47: 678–89.
[25] Hartmann S. A competitive genetic algorithm for resource-constrained project scheduling. Naval Research Logistics 1998;45:279–302.
[26] Kohlmorgen U, Schmeck H, Haase K. Experiences with fine-grained parallel genetic algorithms. Annals of Operations Research 1999;90: 203–19.
[27] Hartmann S. A self-adapting genetic algorithm for project scheduling under resource constraints. Naval Research Logistics 2002;49:433–48.
[28] Kochetov Y, Stolyar A. Evolutionary local search with variable neighborhood for the resource constrained project scheduling problem. In: Proceedings of the 3rd international workshop of computer science and information technologies, 2003.
[29] Mendes JJM. Sistema de apoio à decisão para planeamento de sistemas de produção do tipo projecto. PhD thesis, Departamento de Engenharia Mecânica e Gestão Industrial, Faculdade de Engenharia da Universidade do Porto, Porto, Portugal; 2003 (in Portuguese).

[30] Valls V, Ballestin J, Quintanilla MS. A hybrid genetic algorithm for the RCPSP. Technical Report, Department of Statistics and Operations Research, University of Valencia; 2003.
[31] Valls V, Ballestin F, Quintanilla MS. Justification and RCPSP: a technique that pays. European Journal of Operational Research 2005;165: 375–86.
[32] Debels D, Vanhoucke M. A decomposition-based heuristic for the resource-constrained project scheduling problem. Technical Report 2005/293, Faculty of Economics and Business Administration, University of Ghent, Ghent, Belgium; 2005.
[33] Slowinski R, Soniewicki B, Weglarz J. DSS for multiobjective project scheduling. European Journal of Operational Research 1994;79:220–9.
[34] Boctor FF. An adaptation of the simulated annealing algorithm for solving resource-constrained project scheduling problems. International Journal of Production Research 1996;34:2335–51.
[35] Bouleimen K, Lecocq H. A new efficient simulated annealing algorithm for the resource-constrained project scheduling problem and its multiple mode version. European Journal of Operational Research 2003;149:268–81.
[36] Pinson E, Prins C, Rullier F. Using tabu search for solving the resource constrained project scheduling problem. Technical Report, Universite Catholique de l'Ouest, Angers, 1994.
[37] Baar T, Brucker P, Knust S. Tabu-search algorithms and lower bounds for the resource-constrained project scheduling problem. In: Voss S, Martello S, Osman I, Roucairol C, editors. Meta-heurisitics: advances and trends in local search paradigms for optimization. Dordrecht: Kluwer; 1998. p. 1–8.
[38] Thomas PR, Salhi S. A tabu search approach for the resource constrained project scheduling problem. Journal of Heuristics 1998;4:123–39.
[39] Nonobe K, Ibaraki T. Formulation and tabu search algorithm for the resource constrained project scheduling problem. In: Ribeiro CC, Hansen P, editors. Essays and surveys in metaheuristics. Dordrecht: Kluwer Academic Publishers; 2002. p. 557–88.
[40] Gagnon M, Boctor FF, d'Avignon G. A tabu search algorithm for the resource-constrained project scheduling problem. In: Proceedings of administrative sciences association of Canada annual conference (ASAC 2004); 2004.
[41] Fleszar K, Hindi KS. Solving the resource-constrained project scheduling problem by a variable neighbourhood search. European Journal of Operational Research 2004;155:402–13.
[42] Palpant M, Artigues C, Michelon P. LSSPER: solving the resource-constrained project scheduling problem with large neighbourhood search. Annals of Operations Research 2004;131:237–57.
[43] Debels D, De Reyck B, Leus R, Vanhoucke M. A hybrid scatter search/electromagnetism meta-heuristic for project scheduling. European Journal of Operational Research 2006;169:638–53.
[44] Icmeli O, Erenguc SS, Zappe CJ. Project scheduling problems: a survey. International Journal of Operations & Production Management 1993;13:80–91.
[45] Herroelen W, De Reyck B, Demeulemeeste E. Resource-constrained project scheduling: a survey of recent developments. Computers & Operations Research 1998;25:279–302.
[46] Brucker P, Drexl A, Möhring R, Neumann K, Pesch E. Resource-constrained project scheduling: notation, classification, models, and methods. European Journal of Operational Research 1999;112:3–41.
[47] Klein R. Scheduling of resource-constrained projects. Dordrecht: Kluwer Academic Publishers; 1999.
[48] Hartmann S, Kolisch R. Experimental evaluation of state-of-the-art heuristics for the resource-constrained project scheduling problem. European Journal of Operational Research 2000;127:394–407.
[49] Kolisch R, Padman R. An integrated survey of deterministic project scheduling. The International Journal of Management Science, Omega 2001;29:249–72.
[50] Demeulemeester E, Herroelen W. Project scheduling—a research handbook. Dordrecht: Kluwer Academic Publishers; 2002.
[51] Kolisch R, Hartmann S. Experimental investigation of heuristics for resource-constrained project scheduling: an update. European Journal of Operational Research 2006;174(1):23–37.
[52] Brucker P, Knust S. Complex scheduling. Berlin: Springer; 2006.
[53] Gonçalves JF, Beirão NC. Um algoritmo genético baseado em chaves aleatórias para sequenciamento de operações. Revista Associação Portuguesa Investigação Operacional 1999;19:123–37 (in Portuguese).
[54] Gonçalves JF, Mendes JJM, Resende MGC. A hybrid genetic algorithm for the job shop scheduling problem. European Journal of Operational Research 2005;167:77–95.
[55] Beasley D, Bull DR, Martin RR. An overview of genetic algorithms: Part 1, fundamentals. University Computing 1993;15(2):58–69 Department of Computing Mathematics, University of Cardiff, UK.
[56] Bean JC. Genetics and random keys for sequencing and optimization. ORSA Journal on Computing 1994;6:154–60.
[57] Goldberg DE. Genetic algorithms in search optimization and machine learning. Reading, MA: Addison-Wesley; 1989.
[58] Spears WM, Dejong KA. On the virtues of parameterized uniform crossover. In: Proceedings of the fourth international conference on genetic algorithms, 1991, p. 230–6.
[59] Kolisch R, Sprecher A, Drexl A. Characterization and generation of a general class of resource-constrained project scheduling problems. Management Science 1995;41:1693–703.
[60] Valls V, Ballestin F, Quintanilla MS. A population-based approach to the resource-constrained project scheduling problem. Annals of Operations Research 2004;131:305–24.
[61] Schirmer A, Riesenberg S. Case-based reasoning and parameterized random sampling for project scheduling. Technical Report, University of Kiel, Germany, 1998.
[62] Kolisch R, Drexl A. Adaptative search for solving hard project scheduling problems. Naval Research Logistics 1996;43:23–43.
[63] Kolisch R. Project scheduling under resource constraints: efficient heuristics for several problem classes. Wurzburg: Physica-Verlag; 1995.
[64] Möhring RH, Schulz AS, Stork F, Uetz M. Solving project scheduling problems by minimum cut computations. Management Science 2003;49:330–50.

[65] Gonçalves JF, Almeida JR. A hybrid genetic algorithm for assembly line balancing. Journal of Heuristics 2002;8:629–42.
[66] Gonçalves JF, Resende MGC. An evolutionary algorithm for manufacturing cell formation. Computers & Industrial Engineering 2004;47: 247–73.
[67] Buriol LS, Resende MGC, Ribeiro CC, Thorup M. A hybrid genetic algorithm for the weight setting problem in OSPF/IS-IS routing. Networks 2005;46(1):36–56.
[68] Gonçalves JF. A hybrid genetic algorithm-heuristic for a two-dimensional orthogonal packing problem. European Journal of Operational Research 2007;183(3):1212–29.
[69] Stinson JP, Davis EW, Khumawala BM. Multiple resource-constrained scheduling using branch and bound. AIIE Transactions 1978;10:23–40.