Published in *Annals of Operations Research* 102:111-135 (2001).
© Kluwer, doi:10.1023/A:1010902015091

Project Scheduling with Multiple Modes: A Genetic Algorithm

Sönke Hartmann*

Christian-Albrechts-Universität zu Kiel, Lehrstuhl für Produktion und Logistik, D-24098 Kiel, Germany. E-mail: hartmann@bwl.uni-kiel.de

*supported by the *Studienstiftung des deutschen Volkes*

Abstract. In this paper we consider the resource-constrained project scheduling problem with multiple execution modes for each activity and makespan minimization as objective. We present a new genetic algorithm approach to solve this problem. The genetic encoding is based on a precedence feasible list of activities and a mode assignment. After defining the related crossover, mutation, and selection operators, we describe a local search extension which is employed to improve the schedules found by the basic genetic algorithm. Finally, we present the results of our thorough computational study. We determine the best among several different variants of our genetic algorithm and compare it to four other heuristics that have recently been proposed in the literature. The results that have been obtained using a standard set of instances show that the new genetic algorithm outperforms the other heuristic procedures with regard to a lower average deviation from the optimal makespan.

Keywords. Project Management and Scheduling, Multiple Modes, Genetic Algorithms, Local Search, Computational Results.

1 Introduction

Within the classical resource-constrained project scheduling problem (RCPSP), the activities of a project have to be scheduled such that the makespan of the project is minimized. Thereby, technological precedence constraints have to be observed as well as limitations of the renewable resources required to accomplish the activities. Once started, an activity may not be interrupted.

This problem has been extended to a more realistic model, the multi-mode resource-constrained project scheduling problem (MRCPSP). Here, each activity can be performed in one out of several modes (cf. Elmaghraby [8]). Each mode of an activity represents an alternative way of combining different levels of resource requirements with a related duration. Following Slowinski [24], renewable, nonrenewable and doubly constrained resources are dinstinguished. While renewable resources have a limited per-period availability such as manpower and machines, nonrenewable resources are limited for the entire project, allowing to model, e.g., a budget for the project. Doubly constrained resources are limited both for each period and for the whole project. However, since they can simply be incorporated by enlarging the sets of the renewable and nonrenewable resources, we do not consider them explicitly. The objective is to find a mode and a start time for each activity such that the schedule is makespan minimal and feasible with respect to the precedence and resource constraints.

The outlined problem arises within systems for production planning and scheduling as well as project management software. However, as shown by Sprecher and Drexl [27], even the currently

most powerful optimization procedures are unable to find optimal schedules for highly resource-constrained projects with more than 20 activities and three modes per activity within reasonable computation times. Hence, in practice heuristic algorithms to generate near-optimal schedules for larger projects are of special interest.

Several heuristic procedures for solving the MRCPSP have been proposed in the literature: Drexl and Grünewald [6] suggest a regret-based biased random sampling approach. Slowinski et al. [25] describe a single-pass approach, a multi-pass approach, and a simulated annealing algorithm. Kolisch and Drexl [17] present a local search procedure. Özdamar [23] proposes a genetic algorithm based on a priority rule encoding. Bouleimen and Lecocq [5] suggest a simulated annealing heuristic. Sprecher and Drexl [27] develop a branch-and-bound procedure which is, according to the results obtained by Hartmann and Drexl [11], the currently most powerful algorithm for exactly solving the MRCPSP. Sprecher and Drexl [27] suggested to use it as a heuristic by imposing a time limit. Finally, Boctor [2, 3, 4] as well as Mori and Tseng [22] present heuristics for multi-mode problems without nonrenewable resources.

This paper introduces a new genetic algorithm (GA) approach for solving the MRCPSP. The stepping stone is the concept of an activity list representation successfully employed by Hartmann [10] for the single-mode RCPSP. Extending this approach, we develop a genotype representation which consists of a precedence feasible activity list and a mode assignment. The phenotype, i.e., schedule, related to a genotype is generated using a serial scheduling scheme. The basic GA is extended by two local search components. The first one supports the process of finding nonrenewable resource feasible mode assignments while the second one systematically improves the feasible schedules found by the GA. We test the GA on the basis of a standard set of project instances. After analyzing the behaviour of our GA, we compare it to four heuristic approaches proposed in the literature.

2 Problem Description: The MRCPSP

We consider a project which consists of J activities (jobs) labeled $j = 1, \ldots, J$. Due to technological requirements the activities are partially ordered, that is, there are precedence relations between some of the jobs. These precedence relations are given by sets of immediate predecessors $\mathcal{P}_j$ indicating that an activity j may not be started before all of its predecessors are completed. The transitive closure of the precedence relations is given by sets of (not necessarily immediate) predecessors $\overline{\mathcal{P}}_j$. The precedence relations can be represented by an activity-on-node network which is assumed to be acyclic. We consider additional activities $j = 0$ representing the only source and $j = J + 1$ representing the unique sink activity of the network.

With the exception of the (dummy) source and (dummy) sink activity, each activity requires certain amounts of resources to be performed. The set of renewable resources is referred to as $\mathcal{K}^\rho$. For each renewable resource $k \in \mathcal{K}^\rho$ the per-period-availability is constant and given by R_k^ρ. The set of nonrenewable resources is denoted as $\mathcal{K}^\nu$. For each nonrenewable resource $k \in \mathcal{K}^\nu$ the overall availability for the entire project is given by R_k^ν.

Each activity can be performed in one of several different modes of accomplishment. A mode represents a combination of different resources and/or levels of resource requests with a related duration. Once an activity is started in one of its modes, it is not allowed to be interrupted, and its mode may not be changed. Activity j may be executed in M_j modes given by the set $\mathcal{M}_j = \{1, \ldots, M_j\}$. The processing time or duration of job j being performed in mode $m \in \mathcal{M}_j$ is given by p_{jm}. Furthermore, activity j executed in mode m uses r_{jmk}^ρ units of renewable resource k each period it is in process, where we assume without loss of generality $r_{jmk}^\rho \le R_k^\rho$ for each renewable resource $k \in \mathcal{K}^\rho$. Note, otherwise activity j could not be performed in mode m. Moreover, it consumes r_{jmk}^ν units of nonrenewable resource $r \in \mathcal{K}^\nu$. We assume that the dummy source and the dummy sink activity have only one mode each with a duration of zero periods and no request for any resource.

The objective is to minimize the makespan of the project. We assume the parameters to be

nonnegative and integer valued. A mathematical programming formulation of this problem has been given by Talbot [29].

3 A Genetic Algorithm

In this section we present a new genetic algorithm (GA) approach for the MRCPSP. Introduced by Holland [13], GAs serve as a heuristic meta strategy to solve hard optimization problems. Following the basic principles of biological evolution, they essentially recombine existing solutions to obtain new ones. The goal is to successively produce better solutions by selecting the better ones of the existing solutions more frequently for recombination. For an introduction into GAs, we refer to Goldberg [9].

3.1 Basic Scheme

Before the GA itself is executed, we apply a preprocessing procedure which adapts the project data in order to reduce the search space. After preprocessing, the GA starts by computing an initial population, i.e., the first generation. The number of individuals in the population is denoted as POP. We assume POP to be an even integer. Then it determines the fitness values of the individuals of the initial population. After that, the population is randomly partitioned into pairs of individuals. To each resulting pair of (parent) individuals, we apply the crossover operator to produce two new (children) individuals. Subsequently, we apply the mutation operator to the genotypes of the newly produced children. After computing the fitness of each child individual, we add the children to the current population, leading to a population size of $2 \cdot POP$. Then we apply the selection operator to reduce the population to its former size POP and obtain the next generation to which we again apply the crossover operator. This process is repeated for a prespecified number of generations which is denoted as GEN or, alternatively, until a given CPU time limit is reached.

More formally, the GA scheme can be summarized as follows. $\mathcal{POP}$ denotes the current population (i.e., a set of individuals), and $\mathcal{CHI}$ is the set of children. G is the current generation number.

execute preprocessing procedure;
$G := 1$;
generate initial population $\mathcal{POP}$;
compute fitness for individuals $I \in \mathcal{POP}$;
WHILE $G < GEN$ **AND** time limit is not reached **DO**
BEGIN
    $G := G + 1$;
    produce children $\mathcal{CHI}$ from $\mathcal{POP}$ by crossover;
    apply mutation to children $I \in \mathcal{CHI}$;
    compute fitness for children $I \in \mathcal{CHI}$;
    $\mathcal{POP} := \mathcal{POP} \cup \mathcal{CHI}$;
    reduce population $\mathcal{POP}$ by means of selection;
END.

Observe that exactly $POP \cdot GEN$ individuals are computed (if a time limit is not given). Hence, it is possible to specify in advance the number of schedules to be computed (because one individual corresponds to one or—in a variant to be described later on—two schedules). Therefore, this general GA scheme allows an easy comparison with other heuristics in computational experiments, given that the computational effort for constructing one schedule is similar in the heuristics under consideration.

It should be mentioned that the overall scheme which adds children to the current population and then reduces the population size by removing individuals has been used by Eiben et al. [7]. The parent selection mechanism which implies that each individual is selected exactly once for mating and which leads to a doubled population size after offspring production has been used by Hartmann [10].

In the following subsections, the components of the GA are described. Throughout this section, we illustrate the definitions using the project example displayed in Figure 1.

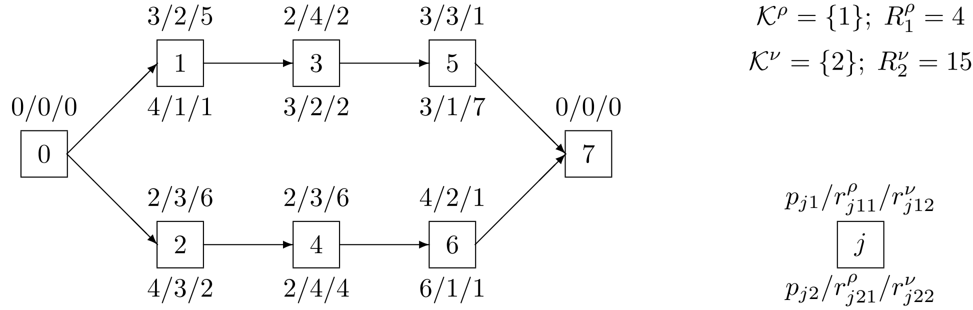

Figure 1: Project instance

3.2 Preprocessing

Before the execution of the GA itself, the project data is adapted by preprocessing in order to reduce the search space. The reduction procedure has been introduced by Sprecher et al. [28] in order to accelerate a branch-and-bound algorithm for the MRCPSP. We briefly summarize the definitions and results: Sprecher et al. [28] define a mode to be non-executable if its execution would violate the renewable or nonrenewable resource constraints in any schedule. A mode is called inefficient if its duration is not shorter and its resource requests are not less than those of another mode of the same activity. A nonrenewable resource is called redundant if the sum of the maximal requests of the activities for this resource does not exceed its availability. Clearly, redundant nonrenewable resources as well as non-executable and inefficient modes may be deleted from the project data without affecting the optimal makespan.

As described by Sprecher et al. [28], there are interaction effects between the elimination of modes and nonrenewable resources. For example, removing a nonrenewable resource may cause inefficiency of a mode while deleting a mode may lead to redundancy of a nonrenewable resource. Consequently, the project data is adapted as follows: First, all non-executable modes are deleted. Second, all redundant nonrenewable resources and, subsequently, all inefficient modes are removed. The second step is repeated until no redundant nonrenewable resources are left. The result of this modification is a reduction of the number of feasible as well as infeasible solutions, that is, a smaller search space.

Consider the project instance given in Figure 1. If activity 5 was performed in mode 2, the whole project would require at least 17 units of the nonrenewable resource whereas only 15 units are available. Hence, mode 2 of activity 5 is non-executable with respect to to the nonrenewable resource and may therefore be deleted.

3.3 Definition of Individuals

As we are dealing with the multi-mode extension of the standard RCPSP, our genetic representation has to reflect both the scheduling problem (which assigns start times to activities) and the mode assignment problem (which assigns modes to activities). Our goal is to deal with both problems simultaneously because of the interactions between them (e.g., there is no “good” mode assignment

itself as long as we don't know how the resulting processing times and resource requirements influence possible start times and hence the makespan).

With this in mind, we represent an individual by a pair $I = (\lambda, \mu)$ of a precedence feasible activity list $\lambda = (j_1, \ldots, j_J)$ and a mode assignment μ. An activity list is precedence feasible if any predecessor of an activity appears before this activity in the list, that is, $\mathcal{P}_{j_i} \subseteq \{j_1, \ldots, j_{i-1}\}$ for $i = 1, \ldots, J$. A mode assignment μ is a mapping which assigns to each activity $j \in \{1, \ldots, J\}$ one of its modes $\mu(j) \in \mathcal{M}_j$. We will use the following alternative notation for the individuals:

$$I = \begin{pmatrix} j_1 & \cdots & j_J \\ \mu(j_1) & \cdots & \mu(j_J) \end{pmatrix}.$$

Each genotype $I = (\mu, \lambda)$ is related to a uniquely determined schedule (phenotype) which is obtained from fixing the modes according to the mode assignment μ and then applying the so-called serial schedule generation scheme to activity list λ for the resulting single-mode problem. That is, we first start the dummy source activity at time 0. Then we successively take the next unscheduled activity from the list λ and schedule it at the earliest precedence and resource feasible start time, using the mode specified by the mode assignment μ.

Clearly, the schedule related to an individual is feasible with respect to the precedence relations and the renewable resource constraints, but not necessarily with respect to the nonrenewable resource constraints. However, it is useful to include schedules that are infeasible with respect to the nonrenewable resources into the search space because, as proven by Kolisch and Drexl [17], already finding a feasible schedule is an NP-complete problem if at least two nonrenewable resources are given. In other words, we cannot find a procedure which constructs a nonrenewable resource feasible individual (e.g., for the initial population) in polynomial time. Therefore, the search for nonrenewable resource feasible mode assignments and the search for near-optimal schedules are done simultaneously by the GA.

3.4 Fitness Computation

The fitness of an individual $I = (\lambda, \mu)$ is computed as follows: Let $L_k^\nu(\mu)$ denote the leftover capacity of nonrenewable resource $k \in \mathcal{K}^\nu$ with respect to the mode assignment μ, that is,

$$L_k^\nu(\mu) = R_k^\nu - \sum_{j=1}^{J} r_{j\mu(j)k}^\nu.$$

A negative leftover capacity $L_k^\nu(\mu) < 0$ implies infeasibility of mode assignment μ with respect to nonrenewable resource k. Let the number of nonrenewable resource units that exceed the capacities be given by

$$L^\nu(\mu) = \sum_{\substack{k \in \mathcal{K}^\nu \\ L_k^\nu(\mu) < 0}} |L_k^\nu(\mu)|.$$

We denote the makespan of a schedule related to an individual I as $C_{\max}(I)$. Moreover, let T be the upper bound on the project's makespan that is given by the sum of the maximal durations of the activities. Now we are ready to define the fitness of an individual I as

$$f(I) = \begin{cases} C_{\max}(I) & \text{if } I \text{ is feasible with respect to the nonrewewable resources} \\ T + L^\nu(\mu) & \text{otherwise.} \end{cases}$$

That is, the fitness of a feasible individual is given by the makespan of the related schedule. Infeasibility with respect to the nonrenewable resource constraints is penalized in the fitness function by adding the number of requested nonrenewable resource units that exceed the capacity. From the definitions given above it is clear that a lower fitness of an individual implies a better quality

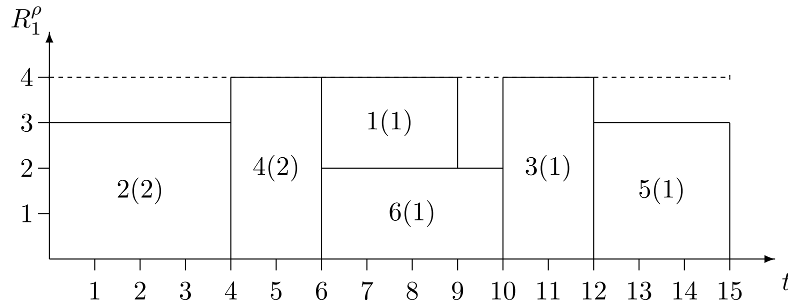

Figure 2: Schedule of example individual I^M

of the related schedule. In particular, a feasible individual always has a lower (i.e., better) fitness than an infeasible one.

For illustration, we consider the project instance given in Figure 1 and the two example individuals

$$I^M = \begin{pmatrix} 2 & 4 & 1 & 6 & 3 & 5 \\ 2 & 2 & 1 & 1 & 1 & 1 \end{pmatrix}, \quad I^F = \begin{pmatrix} 1 & 3 & 2 & 5 & 4 & 6 \\ 1 & 2 & 1 & 1 & 2 & 2 \end{pmatrix}. \tag{1}$$

Clearly, the mode assignment of individual I^M is feasible as 15 units of the single nonrenewable resource are requested, that is, the capacity is not exceeded. Now we derive a schedule from the genotype of I^M which can be found in Figure 2, where $j(m)$ stands for activity j being performed in mode m. The fitness of I^M is equal to the makespan of the schedule, that is, we have $f(I^M) = 15$. Individual I^F induces a nonrenewable resource requirement of 19 units which exceeds the availability by 4 units. Computing $T = 22$, we obtain a fitness value of $f(I^F) = 26$.

3.5 Initial Population

Each individual of the initial population is computed by the following three steps which are repeated until POP individuals are generated.

First, a mode assignment is generated by randomly selecting $\mu(j) \in \mathcal{M}_j$ for activities $j = 1, \ldots, J$.

Second, the mode assignment μ is checked for nonrenewable resource feasibility. If the capacity of a nonrenewable resource would be exceeded, i.e., if $L^\nu(\mu) > 0$, then the following simple local search procedure which follows a first fit strategy tries to improve the current mode assignment: We randomly select an activity $j \in J$ with $M_j > 1$ and a mode $m_j \in \mathcal{M}_j$ with $m_j \neq \mu(j)$, leading to a new mode assignment μ'. If this new mode assignment is at least as good as the previous one, that is, if $L^\nu(\mu') \leq L^\nu(\mu)$, then we accept this new mode by setting $\mu := \mu'$, that is, $\mu(j) := m_j$. This process is repeated until J consecutive unsuccessful trials to improve the mode assignment (by reducing $L^\nu(\mu)$) have been made or until the mode assignment is feasible (i.e., $L^\nu(\mu) = 0$). Recall that there is no algorithm which guarantees the construction of feasible mode assignment in polynomial time, which is due to the NP-completeness of this problem.

Third, based on mode assignment μ, the priority rule based sampling method of Kolisch [15] for the single-mode case is employed to construct a precedence feasible activity list λ. More precisely, we (temporarily) fix the modes of the activities with respect to μ. Then, starting with the empty activity list, we obtain a precedence feasible activity list by repeatedly selecting an activity from the set of the eligible activities. The probability with which an activity is selected is derived from the latest finish times which are derived by so-called backward recursion on the basis of the activity durations given by the mode assignment.

3.6 Crossover

For our problem specific representation, we cannot use any of the standard crossover and mutation operators defined in the literature. Therefore, we introduce the following crossover approach which considers the activity list concept, precedence feasibility, and the mode assignment.

We select two "parent" individuals for crossover, a mother $I^M = (\lambda^M, \mu^M)$ and a father $I^F = (\lambda^F, \mu^F)$ with

$$I^M = \begin{pmatrix} j_1^M & \cdots & j_J^M \\ \mu^M(j_1^M) & \cdots & \mu^M(j_J^M) \end{pmatrix}$$

and

$$I^F = \begin{pmatrix} j_1^F & \cdots & j_J^F \\ \mu^F(j_1^F) & \cdots & \mu^F(j_J^F) \end{pmatrix}.$$

Then we draw two random integers q_1 and q_2 with $1 \leq q_1, q_2 \leq J$. Now two new individuals, a daughter $I^D = (\lambda^D, \mu^D)$ and a son $I^S = (\lambda^S, \mu^S)$, are produced from the parents. We first consider I^D which is defined as follows: In the activity list λ^D of I^D, the positions $i = 1, \ldots, q_1$ are defined by the mother, that is, we set

$$j_i^D := j_i^M.$$

The partial activity list of positions $i = q_1 + 1, \ldots, J$ in λ^D is derived from the father, but the activities that have already been taken from the mother may not be considered again, that is,

$$j_i^D := j_k^F \text{ where } k \text{ is the lowest index such that } j_k^F \notin \{j_1^D, \ldots, j_{i-1}^D\}.$$

Note that this definition ensures that the relative positions in the parents' activity sequences are preserved. Observe that the resulting job sequence is precedence feasible.

The modes of the activities on the positions $i = 1, \ldots, q_2$ in daughter I^D are defined by the mother's mode assignment μ^M, that is, we have

$$\mu^D(j_i^D) := \mu^M(j_i^D).$$

The modes of the remaining jobs on the positions $i = q_2 + 1, \ldots, J$ in I^D are derived from the father's mode assignment μ^F:

$$\mu^D(j_i^D) := \mu^F(j_i^D).$$

The son I^S of the parent individuals I^M and I^F is computed accordingly. However, the positions $1, \ldots, q_1$ of the son's activity list λ^S are taken from the father and the remaining positions are determined by the mother. Analogously, the first part up to position q_2 of the mode assignment μ^S of I^S is taken from μ^F while the second part is derived from μ^M.

These definitions are illustrated by the following example. We use again the project instance of Figure 1 and the example parents of (1). Setting $q_1 = 3$ and $q_2 = 4$, we obtain

$$I^D = \begin{pmatrix} 2 & 4 & 1 & 3 & 5 & 6 \\ 2 & 2 & 1 & 1 & 1 & 2 \end{pmatrix}, \quad I^S = \begin{pmatrix} 1 & 3 & 2 & 4 & 6 & 5 \\ 1 & 2 & 1 & 2 & 1 & 1 \end{pmatrix}. \tag{2}$$

Consider the daughter I^D. The first three positions of the activity list are equal to those of the mother's activity list. The order of the remaining activities is taken from I^F. According to the value of q_2, the modes of the first four activities in the activity list of I^D are determined by the mother's mode assignment while the last two activities get their modes from the father's mode assignment. Observe that, as we have $q_1 < q_2$ in this example, the fourth activity of the daughter's activity list, activity 3, is determined by the father's activity list. The mode of activity 3, however, is taken from the mother.

3.7 Mutation

The representation-specific mutation operator included in our GA is applied to each newly generated child individual $I = (\lambda, \mu)$. It first modifies the related activity list λ: For all positions $i = 1, \ldots, J-1$, activities j_i and j_{i+1} are exchanged with a probability of p_{mutation}, if the result is an activity list which fulfills the precedence assumption. Note that this does not affect the mode assignment, that is, all activities keep their modes even if their positions within the activity list are changed. Next, the mutation operator modifies the mode assignment μ: Each position $i = 1, \ldots, J$ is modified with probability p_{mutation}. If some position i is to be modified, we reselect $\mu(j_i)$ by randomly drawing a mode out of $\mathcal{M}_{j_i}$. Preliminary computational tests have shown that $p_{\text{mutation}} = 0.05$ is a good choice for the mutation rate.

While the first step may create partial activity sequences (i.e., gene combinations) that could not have been procuced by the crossover operator, the second step may introduce a mode (i.e., gene) that did not occur in the current population.

3.8 Selection

We have considered several variants of the selection operator (see, e.g., Michalewicz [21]). All of them follow a survival-of-the-fittest strategy. The ranking method sorts the individuals with respect to their fitness values and selects the POP best ones while the remaining ones are deleted from the population (ties are broken arbitrarily). The proportional selection derives fitness based probabilities for the individuals in order to decide which individuals are selected for the next generation. Finally, in the tournament selection, a number of individuals (in our case two or three) compete for survival. These competitions, in which the least fit individual is removed from the population, are repeated until POP individuals are left. Based on the results of preliminary computational studies, we have chosen the ranking method as selection operator for our GA.

3.9 Extensions of the General Scheme

We close this section with a few remarks on extensions of the general GA scheme, namely the island model and the repetition approach. The island model considers different islands on which the evolution develops almost (but not totally) independently (cf. Kohlmorgen et al. [14]). That is, the island model takes into account several separated populations instead of only one, and an individual can only mate with others of the same population. The islands are connected by migration, that is, an individual with a good fitness value can "swim to another island" and is then part of another population. This is done in order to spread promising genes into other populations.

In the repetition approach, the execution of the GA is replicated, that is, it is restarted several times on each problem instance. Then the overall best solution found is selected. Observe that this is similar to the island model without migration because each replication corresponds to the evolution on an independent island.

In our experiments, we observed that the island model and the repetition approach did not improve the results in our tests when rather small computation times were allowed. Testing the island model based GA and the standard GA both with the same time or schedule number limit implies that the population size on each separate island becomes much smaller than that used in the standard GA. A small population size leads to a small gene pool, which is a drawback in the optimization process. Clearly, the same explanation holds for the repetition approach. However, for substantially higher computation times, the two extensions slightly improved the results of the standard GA scheme. Nevertheless, we will restrict ourselves to the standard GA scheme in the remainder of this paper. The rather small standard project sets used in our tests made smaller time limits more appropriate. As the standard GA scheme yields better results for smaller time limits, it was considered to be sufficient. One should keep in mind, however, that the island model and the repetition approach can be promising if higher computation times are allowed.

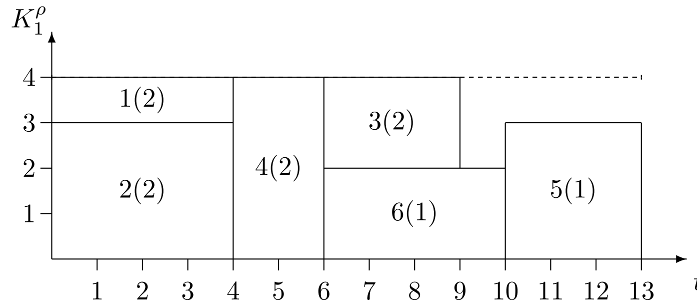

Figure 3: Improved schedule of example individual I^M

4 Improving Schedules by Local Search

We will now introduce a local search method for the MRCPSP to improve the schedule related to an individual. The approach is based on the definition of a multi-mode left shift which was originally defined for a bounding rule in an exact algorithm (cf. Sprecher et al. [28]). A multi-mode left shift of an activity j is an operation on a given schedule which reduces the finish time of activitity j without changing the modes or finish times of the other activities and without violating the constraints. Thereby, the mode of activity j may be changed. Two characteristics make the multi-mode left shift a promising way to improve feasible schedules within our GA: First, multi-mode left shifts consider start times and modes simultaneously. Second, they cannot deteriorate a current feasible schedule, that is, the schedule remains feasible and its makespan cannot be increased.

In what follows, we discuss different approaches to incorporate multi-mode left shift improvement into our GA. In each of these variants, the local search procedure is only applied to feasible schedules.

4.1 Single Pass Improvement

We employ the local search procedure after transforming an individual $I = (\lambda, \mu)$ into a schedule. If the schedule is feasible with respect to the nonrenewable resources, we try to improve it as follows: For each activity $j_1, \ldots, j_J$, we check whether a multi-mode left shift can be performed. Thereby, the modes of an activity j_i are tested with respect to non-decreasing duration, that is, the mode with the shortest duration is checked first. For each activity, the first feasible multi-mode left shift found (if any) is applied to the schedule. Then we skip to the next activity in the activity list. The result is a feasible schedule with a makespan equal to or lower than the makespan of the original schedule. Now the fitness $f(I)$ of individual I is set to be equal to the new makespan. We call it a single pass procedure because each activity is considered only once for a multi-mode left shift.

For illustration, we consider again the project instance given in Figure 1. As already mentioned, example individual I^M of (1) is related to the schedule shown in Figure 2. This schedule is improved by the procedure described above as follows: While activities 2 and 4 cannot be left shifted, we can perform a multi-mode left shift on activity 1. Leaving activity 6 unchanged, we can now apply a multi-mode left shift to activity 3. Finally, left shifting activity 5 yields a new makespan of 13 periods which is 2 periods shorter than the makespan of the original schedule. The resulting schedule is displayed in Figure 3.

Note that the basic GA without local search leads to the computation of one schedule for each individual, that is, we obtain $POP \cdot GEN$ schedules for a project instance. Single pass local search computes a second (and hopefully improved) schedule for an individual if the original schedule for that individual was feasible. Consequently, the GA with the single pass extension constructs at most $2 \cdot POP \cdot GEN$ schedules for an instance.

4.2 Multi Pass Improvement

As each activity is considered only once for a multi-mode left shift, a schedule derived by the above single pass local search algorithm is not necessarily tight (a tight schedule is a schedule to which no multi-mode left shift can be applied, cf. Sprecher et al. [28]). This is because a multi-mode left shift of some activity j_i might allow a multi-mode left shift of some activity j_k with $k < i$ that had not been possible before. In the improved schedule of Figure 3, for example, a multi-mode left shift of activity 2 is possible because the total consumption of the nonrenewable resource decreased due to the multi-mode left shift of activity 1.

Consequently, we have tested a second variant of our local search extension which will be called multi pass procedure: We repeatedly apply the above single pass improvement algorithm until we have obtained a tight schedule. In contrast to the single pass algorithm, the multi pass approach always leads to a local optimum as the resulting schedule cannot be further improved using multi-mode left shifts.

Note that neither the single pass nor the multi pass approach can improve schedules for the single-mode RCPSP found by the GA. This is because in the single-mode case the set of tight schedules coincides with the set of the active ones (cf. Sprecher et al. [28]), and each schedule computed by the serial schedule generation scheme is active.

4.3 Inheritance Beyond the Genetic Metaphor

So far, our local search improvement—either in the single pass or multi pass variant—can be viewed as a second step of the genotype evaluation as it computes schedule-related fitness values. But we can do more: Applying the local search approach to a schedule S related to an individual I, we obtain an improved schedule S'. Now we can transform the new schedule S' into an individual I', that is, we can find an individual I' (genotype) which corresponds to the improved schedule S' (phenotype). Subsequently, we can replace individual I with the improved individual I' in the current population. Clearly, the activity list of individual I' is given by the sequence of activities ordered with respect to non-decreasing start times within schedule S'; the mode assignment is straightforward.

It should be mentioned that the computational effort of transforming an improved schedule back into a genotype is neglectable when compared to that of computing the original or the improved schedule for an individual. That is, the GA with, e.g., single pass improvement in the two variants with and without inheritance will show the same computation times.

For illustration, we consider again the project instance of Figure 1 and example individual I^M which is, after application of the single-pass improvement, related to the schedule of Figure 3. This improved schedule leads to genotype

$$I^{M'} = \begin{pmatrix} 1 & 2 & 4 & 6 & 3 & 5 \\ 2 & 2 & 2 & 1 & 2 & 1 \end{pmatrix}.$$

Note that the genotype of $I^{M'}$ is not the only one that would lead to the schedule of Figure 3. Whereas each genotype is related to one unique schedule, a schedule may be related to more than one genotype. The reason for this is essentially that activities may be performed in parallel, whereas the genotypes prescribe a complete ordering of the activities due to the activity lists. For example, the schedule of Figure 3 also corresponds to individual

$$I^{M''} = \begin{pmatrix} 2 & 1 & 4 & 3 & 6 & 5 \\ 2 & 2 & 2 & 2 & 1 & 1 \end{pmatrix}.$$

In order to obtain a well-defined genotype for an improved schedule, we extend the transformation given above as follows: We sort the activities with respect to non-decreasing start times and, as a tie breaker for activities with the same start time, with respect to non-decreasing activity number.

Considering evolution in biology, the improvement procedure which only affects the phenotype (schedule) can be compared to individual or ontogenetic learning. The transformation of its results into a new genotype, i.e., into hereditary information, corresponds to the possibility to inherit the results of ontogenetic learning as proposed by Jean-Baptiste de Monet Chevalier de Lamarck (1744-1829). Lamarck claimed that physical changes of an individual occur if and because they are useful, and that these changes are passed on to its offspring. In nature, however, changes in the phenotype of an individual usually do not affect its genotype.

5 Computational Results

5.1 Experimental Design

In this section we present the results of the computational studies concerning the genetic algorithm introduced in the previous section. The experiments have been performed on a Pentium-based IBM-compatible personal computer with 133 MHz clock-pulse and 32 MB RAM. The GA has been coded in ANSI C, compiled with the GNU C compiler and tested under Linux.

We used a set of standard test problems systematically constructed by the project generator ProGen which has been developed by Kolisch et al. [20]. They are available in the project scheduling problem library PSPLIB from the University of Kiel. For detailed information the reader is referred to Kolisch and Sprecher [19]. Some of these instances have been used by Bouleimen and Lecocq [5], Kolisch and Drexl [17], and Özdamar [23] to evaluate their heuristics for the MRCPSP.

In our study, we have used the multi-mode problem sets containing instances with 10, 12, 14, 16, 18, 20, and 30 non-dummy activities. Each of the non-dummy activities may be performed in one out of three modes. The duration of a mode varies between 1 and 10 periods. We have two renewable and two nonrenewable resources. For each problem size, a set of instances was generated by systematically varying four parameters, that is, the resource factor and the resource strength of each resource category. The resource factor is a measure of the average portion of resources requested per job. The resource strength reflects the scarceness of the resources.

For each project size, 640 instances were generated. Those instances for which no feasible solution exists have not been considered. Hence, we have 536 instances with $J = 10$, 547 instances with $J = 12$, 551 instances with $J = 14$, 550 instances with $J = 16$, 552 instances with $J = 18$, and 554 instances with $J = 20$. The set with 20 non-dummy activities currently is the hardest standard set of multi-mode instances for which all optimal solutions are known, cf. Sprecher and Drexl [27]. For the set with 30 activities, not all optimal solutions are known until now, and for some instances it is currently not known if a feasible solution exists. Consequently, we use all 640 instances with 30 activities in our tests. Note that we can measure the quality of a heuristic in terms of deviation from the optimum if instance sets with up to 20 activities per project are considered. For the set with $J = 30$, we measure the deviation from a lower bound on the makespan. This lower bound is given by the makespan that results from starting each activity as early as possible in its shortest mode while relaxing the (renewable and nonrenewable) resource constraints.

5.2 Configuration of the Algorithm

In the numerical investigation reported in this subsection we determined the best configuration of our GA. We summarize the results only for the instance set with 20 non-dummy activities per project as the results for the smaller projects are similar.

We start with the examination of the preprocessing procedure of Subsection 3.2. It eliminated 4.4% of the modes. Moreover, it was able to show that for 29% of the projects both nonrenewable resources are redundant. Given the low computational effort for the preprocessing procedure, we include it into our GA.

Next, we test the performance of the methodology to generate the initial population. Considering the instances with at least one nonrenewable resource left after preprocessing, only 53%

Improvement	inheritance	$POP =$	30	60	90	120
—	—		3.00%	1.83%	1.51%	1.79%
single-pass	no		1.77%	1.21%	1.38%	1.66%
single-pass	yes		2.42%	1.81%	1.69%	1.85%
multi-pass	no		1.51%	1.32%	1.53%	1.71%
multi-pass	yes		2.13%	1.66%	1.71%	1.75%

Table 1: Impact of local search improvement — 1 second, $J = 20$

of the mode assignments in the initial population are feasible if generated randomly. Adding the local search procedure to modify the mode assignment, already 99 % of the mode assignments in the initial population are nonrenewable resource feasible. This shows that our simple local search procedure is well suited to contribute to a good initial population for the GA and should therefore be incorporated. So far, we have shown that the preprocessing procedure and the local search procedure for modifying the mode assignment are part of the best configuration of the GA.

Next, we determine the best GA variant concerning the local search schedule improvement. We have five possible variants: The plain GA without schedule improvement, the GA with single pass schedule improvement, the GA with single pass improvement including additional inheritance of the local search results, the GA with multi-pass local search, and the GA with multi-pass improvement including additional inheritance of the local search results.

As the local search types require a different computational effort for each individual, we chose a time limit instead of a limited number of individuals as basis for the tests. Given a time limit, only a restricted number of individuals can be computed. Therefore, the population size POP is another important parameter to be determined. A large population size will lead to an evolution over only a few generations and, as we will see, the relationship between population size and number of generations has an impact on the solution quality. As we will also observe, there are interdependencies between the choice of the GA variant and the population size. Consequently, the following experiment simultaneously takes into account both the GA variant and the population size. Given a time limit of one second, Table 1 summarizes the average percentage deviation from the optimal makespan for the three GA variants and four population size settings.

There are several observations to be made: First, the best configuration makes use of the single pass local search without additional inheritance and, with respect to the time limit of one second, a population size of $POP = 60$.

Second, the GA with single pass improvement, but without inheritance is, independently from the population size, always better than the plain GA. This confirms that the GA benefits from the local search improvement. This approach, however, cannot be improved by using the multi pass procedure or by additionally allowing inheritance of the local search results. The disappointing results of the multi pass procedure are due to the fact that multiple application only leads to minor improvements compared to the single pass application while it increases the computation time needed to compute the schedule for one individual. Hence, within some time limit, less individuals can be computed, and the minor effect of obtaining tight schedules is over-consumed. On the other hand, the observation that the inheritance mechanism consistently worsens the results of the GA with local search improvement is rather a surprise as one may have assumed that inheriting improved genes should be advantageous. The computational effort of the inheritance mechanism cannot be the reason as it is neglectable when compared to that of the local search procedure itself. In fact, we need to examine the behavior of the GA variants more carefully to provide an explanation, as will be done in the next subsection.

Third, all variants perform best for a medium population size. If the population size is too large, only a few generations can be computed within the time limit, and the procedure cannot fully exploit the advantages of genetic optimization. Otherwise, if it is too small, the gene pool is too small. Observe also that the best population size for the plain GA is larger than that

Improvement	inheritance	0.20 sec	0.40 sec	0.80 sec	1.60 sec
single-pass	no	4.70%	2.69%	1.42%	0.98%
single-pass	yes	4.12%	2.45%	1.89%	1.64%

Table 2: Impact of local search improvement — intermediate results, $J = 20$

for the GA with single pass improvement but without inheritance. This is due to the additional computational effort needed for the local search improvement. That is, within the same time limit less individuals can be computed, leading to a smaller favorable population size (and also a smaller number of generations).

Finally, we remark that we have tested a huge number of GA variants that resulted from different operators (selection) and parameters (mutation probability, number of crossover points, population size, number of repetitions as well as number of islands, migration probability). For the sake of shortness, we cannot list all details of the results here. Hence, we restrict ourselves to a few comments on the mutation and selection operators. We have mentioned earlier that a mutation rate of $p_{\text{mutation}} = 0.05$ and the ranking strategy for selection are part of the best configuration. While the best variant with these settings leads to an average deviation of 1.21% from the optimal makespan (cf. Table 1), mutation rates of 0.01 and 0.10 result in devitions of 1.39% and 1.31%, respectively. Employing the proportional selection instead of the ranking approach leads to a deviation of 1.66%. It should be noted that all experiments conducted for this paper were repeated many times (and for several different time limits) in order to confirm the results.

5.3 Population Analysis

Up to this point, we know that it pays to incorporate our single pass local search component for schedule improvement into the GA. This subsection proceeds with an analysis of the generations produced by the GA in order to answer a question that has been arising in the previous subsection: Why is it disadvantageous to inherit the local search results? To obtain comparable results, we fix the population size to $POP = 60$ individuals within the GA variants with and without inheritance. Again, we employ only the set with 20 activities for each project.

Our first experiment takes a closer look at the intermediate results of the artificial evolution. Table 2 gives the average percentage deviations after 0.2, 0.4, 0.8, and 1.6 CPU seconds for both GA variants. Shortly after the beginning of the evolution, i.e., after only a few generations have been computed, the GA variant with inheritance leads to better intermediate results than its counterpart without. Then, however, the GA without inheritance becomes superior. Hence, the inheritance mechanism performs better for the smaller computation times, that is, when the number of generations is relatively small compared to the population size. This result indicates that the inheritance mechanism should be used together with a larger population size and thus, considering some fixed time limit, a smaller number of generations. This explains that, according to Table 1, the best POP value for the improvement with inheritance is larger than that for the variant without. However, Table 1 also shows that, given the best population size for both variants, the GA without inheritance is superior.

So far, we have seen that inheriting the local search results has some positive effect in the first few generations. Now we want to find out why there is a negative effect in the long term. We need some definitions for an analysis of the population.

First, we define a measure for the similarity of two individuals $I = (\lambda, \mu)$ and $I' = (\lambda', \mu')$. We start with a definition of the similarity of the related activity lists. Our goal is to check if two activities have the same relative positions in the activity lists λ and λ'. Since it is sufficient to consider only those activities that are not precedence related, we define

$$\mathcal{Q} = \{\{i,j\} \mid i,j = 1,\ldots,J;\ i \neq j;\ i \notin \overline{\mathcal{P}}_j;\ j \notin \overline{\mathcal{P}}_i\}.$$

Given two activities i and j that are not precedence related, i.e., $\{i,j\} \in \mathcal{Q}$, we reflect their relative positions within the activity lists of I and I' by

$$\alpha^{I,I'}_{\{i,j\}} = \begin{cases} 1, & \text{if } i \text{ is before } j \text{ in } \lambda \text{ and } \lambda', \text{ or if } j \text{ is before } i \text{ in } \lambda \text{ and } \lambda', \\ 0, & \text{otherwise.} \end{cases}$$

Now we are ready to define the following measure for the similarity of the activity lists of I and I': If there are activities that are not precedence related, i.e., $\mathcal{Q} \neq \emptyset$, we set

$$\alpha^{I,I'} = \frac{1}{|\mathcal{Q}|} \sum_{\{i,j\} \in \mathcal{Q}} \alpha^{I,I'}_{\{i,j\}}.$$

Otherwise, if $\mathcal{Q} = \emptyset$, we have a serial network structure which implies that there is only one precedence feasible activity sequence. In this case, we define $\alpha^{I,I'} = 1$, reflecting that the activity sequences of individuals I and I' are equal.

Note that $\alpha^{I,I'}$ always counts different activity lists as different, even if they lead to the same schedule (cf. again the discussion about the relationship of genotypes and schedules in Subsection 4.3). This is what we intend because we do not want to count the different schedules in the population but the differences in the gene pool. Consider two genotypes I_1 and I_2 that have different activity lists but lead to the same schedule. I_1 and I_2 could mate with some other individual I. Let us assume that the same crossover points q_1 and q_2 are used. The daughter D_1 of I_1 and I on the one hand and the daughter D_2 of I_2 and I on the other hand may correspond to different schedules although their parents correspond to the same schedules. Hence, the differences in the chromosomes are important when analyzing the evolution, and this is taken into account by the measure $\alpha^{I,I'}$.

The next definition reflects whether an activity j is assigned the same mode by the mode assignments μ and μ' of two individuals I and I':

$$\beta^{I,I'}_j = \begin{cases} 1, & \text{if } \mu^I(j) = \mu^{I'}(j), \\ 0, & \text{otherwise.} \end{cases}$$

This enables us to define the following measure for the similarity of the mode assignments of I and I':

$$\beta^{I,I'} = \frac{1}{J} \sum_{j=1}^{J} \beta^{I,I'}_j.$$

Combining the above definitions, we obtain a measure $\sigma^{I,I'}$ for the similarity of individuals I and I' in which both the activity lists and the mode assignments are considered:

$$\sigma^{I,I'} = \frac{\alpha^{I,I'} + \beta^{I,I'}}{2}.$$

Clearly, the higher $\sigma^{I,I'}$, the more identical information is contained in the genotypes of individuals I and I'. Observe that we have $\sigma^{I,I'} \in [0,1]$. Especially, we have $\sigma^{I,I'} = 0$ if I and I' do not have any genetic information in common, and $\sigma^{I,I'} = 1$ if they are identical, that is, if we have $I = I'$.

Now we use this similarity measure for analyzing the different generations produced by our GA. More precisely, we want to partition each generation into sets of similar individuals. In order to obtain these partitions, we have implemented a cluster analysis algorithm.[1] Given a generation with POP individuals, we first compute the similarity value for each pair of individuals. Then the

[1]For a general introduction into cluster analysis the reader is referred to, e.g., Backhaus et al. [1].

Improvement	inheritance	1	5	10	15	20	25	30	35	40	45	50
single-pass	no	60	38	32	25	17	14	10	9	8	8	7
single-pass	yes	60	31	20	12	9	7	5	4	3	2	2

Table 3: Average number of clusters with respect to generation number

cluster analysis algorithm starts with the trivial partition in which each individual forms a cluster, that is, we have POP clusters. After that we unite clusters as follows: Consider two clusters C and C'. C and C' are united if we have $\sigma^{I,I'} \geq 0.8$ for all individuals $I \in C$ and $I' \in C'$. This is repeated until there are no clusters left that can be united with respect to this similarity criterion. Each resulting cluster contains highly similar individuals.

We have analyzed the populations generated by the GAs with and without inheritance of the single pass local search results. Table 3 lists the average number of clusters that have been obtained by both variants for every fifth generation. As the first generation is randomly determined, there are no similarities of more than 0.8, inducing 60 clusters with one individual each. With an increasing number of generations, the number of clusters in a generation decreases, that is, more similar individuals occur in the population. Clearly, this is due to the crossover and selection operators which tend to copy "fit" and remove "unfit" information. Table 3, however, shows that the inheritance mechanism accelerates the reduction of clusters: While the GA without inheritance leads to 25 different clusters after 15 generations, the GA with inheritance results in only 12 clusters at the same time.

These results explain why including the inheritance mechanism into the GA deteriorates the quality of the solutions in a long term evolution: The basic strategy of any GA is to gather information about promising regions of the search space. Each of our clusters can be viewed as such a promising region. At the same time, however, information that is not considered as promising is removed. Clearly, this leads to a loss of genetic variety. If the number of clusters decreases too fast—in other words, if too much information is lost too fast—the GA gets stuck in some promising regions of the search space. As each cluster contains similar information, the GA is likely to fail to construct individuals from previously unsearched regions of the (usually huge) search space if only few clusters are left. Consequently, many regions remain unexplored, and better solutions may be left undetected. On the other hand, the focus on a few promising clusters is advantageous if the evolution is restricted to a small number of generations, see again Table 2. However, a short term evolution does not fully exploit the power of genetic optimization.

Basically, one encounters the following difficulty when designing a GA: An evolution proceeding too fast leads to a loss of genetic variety and is thus disadvantageous. On the other hand, an evolution proceeding too slowly may be unable to identify promising regions of the search space. Therefore, the variants and parameters have to be chosen carefully.

The results obtained here are in line with the findings of Whitley et al. [30]. They state that a Larmarckian evolution (like that induced by our inheritance approach) leads to a faster search with a tendency to converge to a local optimum whereas a genetic algorithm enhanced by individual learning, but without inheritance of the results may by more likely to converge to a global optimum.

5.4 Comparison with other Metaheuristics

In this subsection we summarize the results obtained from a comparison of our GA with other metaheuristics for solving the MRCPSP that have recently been proposed in the literature. We use the best variant of our GA, that is, with single-pass improvement, but without inheritance of the local search results.

Kolisch and Drexl [17] suggested a local search procedure which successively constructs a neighbor mode assigment based on slack calculations of the current schedule and, after fixing the modes accordingly, computes a schedule for the resulting single-mode RCPSP using a single pass priority

Heuristic	average dev.	feasible	optimal
new GA	0.10%	100.0%	98.1%
Kolisch, Drexl [17]	0.50%	100.0%	91.8%
Özdamar [23]	0.86%	100.0%	88.1%

Table 4: New GA vs. two other heuristics — 6000 schedules, $J = 10$

rule method. After a given number of moves, the procedure tries to improve the schedule for the best current mode assignment by the adaptive sampling approach of Kolisch and Drexl [16]. The experimental investigation of Kolisch and Drexl [17] shows that their approach outperforms the algorithms of Boctor [2] and of Drexl and Grünewald [6].

Özdamar [23] developed a GA based on an encoding which is made up by a sequence of priority rules and a mode assignment. As decoding procedure, the so-called parallel schedule generation scheme is used (cf. also Kolisch and Hartmann [18]). For each individual, two schedules are computed by forward-backward scheduling.

Bouleimen and Lecocq [5] proposed a simulated annealing procedure which makes use of a precedence feasible activity list and a mode assignment. It contains two stages which are repeated alternately. In the first stage, a move alters the mode assignment. In the second stage, the resulting mode assigment is (temporarily) fixed, and the activity list is modified by moves that shift activities within the list.

Let us begin with the heuristics of Kolisch and Drexl [17] and Özdamar [23]. As a basis for the comparison of our GA with these two heuristics, we use the result reported by Özdamar [23] obtained from computing 3000 individuals (which corresponds to 6000 schedules due to a forward and a backward scheduling pass for each individual) for each instance of the ProGen multi-mode set with 10 non-dummy activities in a project. We recompiled the original PASCAL code of Kolisch and Drexl [17] and limited the number of schedules to 6000 for each ProGen instance with $J = 10$. Finally, we tested our GA with 3000 individuals (which corresponds to 6000 schedules due to the single pass improvement for each individual) for each project instance of the same set without imposing a time limit. We may assume that the effort to compute one schedule is similar in the three procedures. Hence, this should yield a fair comparison (in fact, as our GA does not require to compute eligible activities or priority values, it is probably the fastest). We obtained the results that are displayed in Table 4. They show that our GA clearly outperforms the other two heuristics. Note especially that the two genetic algorithms show a different behavior. This indicates that the problem representation and also other components such as the local search extensions have a much higher influence on the performance than the metaheuristic strategy alone.

Now we turn to the simulated annealing procedure of Bouleimen and Lecocq [5]. Table 5 summarizes the results of their heuristic as reported by the authors and compares them to our GA. It can be seen that the new GA outperforms the simulated annealing heuristic. Also note that the computation time of our GA is lower (see the computers and time limits given in Table 5).

5.5 Comparison with Truncated Branch-and-Bound

In what follows, we compare the new GA with a truncated version of the branch-and-bound (B&B) procedure of Sprecher and Drexl [27]. We do so for two reasons: First, the authors have suggested to employ their exact algorithm as a heuristic by imposing a time limit. Second, as shown by Hartmann and Drexl [11], their approach is the currently most efficient exact method for solving the MRCPSP.

For the tests, we have used the reimplementation of the branch-and-bound approach that was done for the study of Hartmann and Drexl [11]. It should be noted that this is an improved version of the original method of Sprecher and Drexl [27] because it includes two new bounding rules. For

Heuristic	J	average dev.	max. dev.	optimal
new GA[a]	10	0.06%	6.3%	98.7%
	12	0.14%	9.1%	97.3%
	14	0.44%	10.3%	89.8%
	16	0.59%	10.5%	87.8%
	18	0.99%	13.3%	78.3%
	20	1.21%	14.2%	73.3%
Bouleimen, Lecocq [5][b]	10	0.21%	7.8%	96.3%
	12	0.19%	6.3%	91.2%
	14	0.92%	10.6%	82.6%
	16	1.43%	12.9%	72.8%
	18	1.85%	11.7%	69.4%
	20	2.10%	13.2%	66.9%

[a] Pentium 133 MHz, time limit: 1 sec
[b] Pentium 100 MHz, time limit: 5 times the instance size (in seconds)

Table 5: New GA vs. simulated annealing

activity selection, we employed the rule recommended by Sprecher [26], that is, the next eligible activity to be selected for branching is the one with the lowest number. The modes are selected with respect to non-decreasing processing time. A time limit is added as basis for the comparison with the GA. We have used the ProGen instance sets with up to 20 activities in a project.

Table 6 summarizes the results obtained from both algorithms for a time limit of one second. For each project size, it lists the average and the maximal deviation from the optimum, the percentage of instances for which an feasible solution was found, and the percentage of instances for which an optimal solution was found. While the truncated exact procedure solves all instances with 10 activities to optimality within one second, its average deviation for the instances with 20 activities is more than nine times higher than that obtained by the GA. In contrast to the GA which results in moderate maximal deviations of at most 15%, the maximal deviation of the truncated branch-and-bound algorithm is almost 80% for $J = 20$. While our GA finds a feasible solution for every instance with $J \leq 20$, the truncated exact procedure fails to do so for instances with more than 12 activities. It is important to note that the results for the set with $J = 30$ reflect deviations from a lower bound, and that for many instances of that set a feasible solution does not exist.

Table 7 gives the results for four different time limits between 1 and 125 seconds and the projects with 20 activities. We observe that both approaches benefit from increasing the computation time. The GA is clearly superior for all time limits and reaches near-optimal solutions within 125 seconds. The truncated branch-and-bound approach still fails to find a feasible schedule for some projects within this time.

Finally, we examine the impact of the renewable resource strength RS^{ρ} on the solution quality of the GA and the truncated branch-and-bound procedure. The resource strength is a parameter of the instance generator ProGen mentioned above. It ranges between 0 and 1. Project instances with a low resource strength are characterized by scarce capacities of the renewable resources. A resource strength of 1 implies that the renewable resource capacities are high enough not to impose restrictions on the schedules. Table 8 shows that a low renewable resource strength makes a project instance harder to solve for both heuristics. The average percentage deviations from the optimal makespan increase with decreasing resource strength for both procedures tested here, given the instance set with $J = 20$ and a time limit of one second. For all RS^{ρ} values, the deviation of the GA is approximately nine times lower than that of the truncated branch-and-bound algorithm. Information on the impact of other project parameters on the behavior of heuristics can be found in Hartmann and Kolisch [12].

Heuristic	J	average dev.[a]	max. dev.[a]	feasible	optimal[b]
new GA	10	0.06%	6.3%	100.0%	98.7%
	12	0.14%	9.1%	100.0%	97.3%
	14	0.44%	10.3%	100.0%	89.8%
	16	0.59%	10.5%	100.0%	87.8%
	18	0.99%	13.3%	100.0%	78.3%
	20	1.21%	14.2%	100.0%	73.3%
	30	16.93%	151.9%	86.3%	n/a
truncated B&B	10	0.00%	0.0%	100.0%	100.0%
	12	0.12%	17.9%	100.0%	98.2%
	14	1.46%	33.3%	99.6%	85.7%
	16	3.81%	52.4%	99.5%	69.5%
	18	7.48%	77.4%	98.0%	57.4%
	20	11.51%	78.6%	96.4%	47.3%
	30	57.22%	244.0%	55.8%	n/a

[a] deviation from optimum for $J \leq 20$ and from lower bound for $J = 30$
[b] not all optimal solutions for $J = 30$ are currently known

Table 6: New GA vs. truncated B&B with respect to project size — 1 sec

Heuristic	CPU-sec	average dev.	max. dev.	feasible	optimal
new GA	1	1.21%	14.2%	100.0%	73.3%
	5	0.49%	10.7%	100.0%	87.9%
	25	0.22%	10.5%	100.0%	94.4%
	125	0.11%	7.1%	100.0%	96.8%
truncated B&B	1	11.51%	78.6%	96.4%	47.3%
	5	6.24%	89.4%	98.6%	63.7%
	25	2.88%	53.6%	99.5%	76.4%
	125	1.04%	34.2%	99.6%	87.9%

Table 7: New GA vs. truncated B&B with respect to time limit — $J = 20$

It is interesting to note that the truncated branch-and-bound procedure dominates on the sets of small instances, that is, those instances that are very easy with respect to the project size (see again Table 6), but not on those that are easy with respect to the resource strength. This indicates that if the number of activities increases, the GA becomes superior, independently from the level of the resource strength. This observation already holds for the rather small instances in the test set, where the GA performs better for projects with more than 12 activities.

6 Conclusions

We have presented a genetic algorithm (GA) for solving project scheduling problems with multiple modes. Our computational experiments show that this procedure outperformed several other heuristics that have been proposed for the MRCPSP in the literature. Several components of the GA contributed to the good results. First and probably most important, the problem specific representation is crucial for the success of the GA. The comparison with another GA approach from the literature suggests that using different representations within the same metaheuristic strategy

Heuristic	$RS^{\rho} = 0.25$	$RS^{\rho} = 0.50$	$RS^{\rho} = 0.75$	$RS^{\rho} = 1.00$
new GA	2.02%	1.26%	1.06%	0.69%
truncated B&B	18.27%	10.61%	9.93%	6.55%

Table 8: New GA vs. truncated B&B with respect to resource strength — average deviation from optimal makespan, 1 sec, $J = 20$

may lead to different results. Second, extending the genetic algorithm framework by local search concepts has been beneficial. We used two local search methods. One was designed to deal with the feasibility problem of the MRCPSP, while the other was used to improve the schedules found by the GA. It is important to note that these components which have proved to be successful are all based on problem-specific knowledge. On the other hand, using further general GA concepts such as the island model did not improve the results.

The results of this study are in line with the findings of Hartmann and Kolisch [12] who provided an experimental evaluation of several state-of-the-art heuristics for the single mode RCPSP. They found out that the best algorithms were metaheuristics (namely the simulated annealing approach of Bouleimen and Lecocq [5] and the GA of Hartmann [10]). Moreover, their results suggest that using a metaheuristic strategy alone does not lead to good results. Our observations for the multi-mode case lead to the same conclusions: Metaheuristic strategies are promising approaches to project scheduling problems, and as much problem-specific knowledge as possible should be incorporated into the heuristic. Future research could include the development of further metaheuristic algorithms for the MRCPSP and their comparison with the GA approach presented here.

Acknowledgements. I am indebted to Rainer Kolisch and Andreas Drexl, Christian-Albrechts-Universität zu Kiel, for making the source code of their heuristic available. I would also like to thank Joanna Józefowska, Poznań University of Technology, for helpful comments on an earlier version of this paper. Finally, I am grateful to the anonymous referees whose remarks helped to improve the paper.

References

[1] K. Backhaus, B. Erichson, W. Plinke, and R. Weiber. *Multivariate Analysemethoden: Eine anwendungsorientierte Einführung.* Springer, Berlin, Germany, 1996.

[2] F. F. Boctor. Heuristics for scheduling projects with resource restrictions and several resource-duration modes. *International Journal of Production Research*, 31:2547–2558, 1993.

[3] F. F. Boctor. An adaptation of the simulated annealing algorithm for solving resource-constrained project scheduling problems. *International Journal of Production Research*, 34:2335–2351, 1996.

[4] F. F. Boctor. A new and efficient heuristic for scheduling projects with resource restrictions and multiple execution modes. *European Journal of Operational Research*, 90:349–361, 1996.

[5] K. Bouleimen and H. Lecocq. A new efficient simulated annealing algorithm for the resource-constrained project scheduling problem. In G. Barbarosoglu, S. Karabati, L. Özdamar, and G. Ulusoy, editors, *Proceedings of the sixth international workshop on project management and scheduling*, pages 19–22. Bogazici University Printing Office, Turkey, 1998.

[6] A. Drexl and J. Grünewald. Nonpreemptive multi-mode resource-constrained project scheduling. *IIE Transactions*, 25:74–81, 1993.

[7] A. E. Eiben, E. H. L. Aarts, and K. M. van Hee. Global convergence of genetic algorithms: A markov chain analysis. *Lecture Notes in Computer Science*, 496:4–12, 1990.

[8] S. E. Elmaghraby. *Activity networks: Project planning and control by network models.* Wiley, New York, 1977.

[9] D. E. Goldberg. *Genetic algorithms in search, optimization, and machine learning.* Addison-Wesley, Reading, Massachusetts, 1989.

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

[11] S. Hartmann and A. Drexl. Project scheduling with multiple modes: A comparison of exact algorithms. *Networks*, 32:283–297, 1998.

[12] S. Hartmann and R. Kolisch. Experimental evaluation of state-of-the-art heuristics for the resource-constrained project scheduling problem. *European Journal of Operational Research*, 127:394–407, 2000.

[13] H. J. Holland. *Adaptation in natural and artificial systems.* University of Michigan Press, Ann Arbor, 1975.

[14] U. Kohlmorgen, H. Schmeck, and K. Haase. Experiences with fine-grained parallel genetic algorithms. *Annals of Operations Research*, 90:203–319, 1999.

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

[16] R. Kolisch and A. Drexl. Adaptive search for solving hard project scheduling problems. *Naval Research Logistics*, 43:23–40, 1996.

[17] R. Kolisch and A. Drexl. Local search for nonpreemptive multi-mode resource-constrained project scheduling. *IIE Transactions*, 29:987–999, 1997.

[18] R. Kolisch and S. Hartmann. Heuristic algorithms for solving the resource-constrained project scheduling problem: Classification and computational analysis. In J. Weglarz, editor, *Project scheduling: Recent models, algorithms and applications*, pages 147–178. Kluwer Academic Publishers, 1999.

[19] R. Kolisch and A. Sprecher. PSPLIB – a project scheduling problem library. *European Journal of Operational Research*, 96:205–216, 1996.

[20] R. Kolisch, A. Sprecher, and A. Drexl. Characterization and generation of a general class of resource-constrained project scheduling problems. *Management Science*, 41:1693–1703, 1995.

[21] Z. Michalewicz. Heuristic methods for evolutionary computation techniques. *Journal of Heuristics*, 1:177–206, 1995.

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

[23] L. Özdamar. A genetic algorithm approach to a general category project scheduling problem. *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, 29:44–59, 1999.

[24] R. Slowinski. Two approaches to problems of resource allocation among project activities: A comparative study. *Journal of the Operational Research Society*, 31:711–723, 1980.

[25] R. Slowinski, B. Soniewicki, and J. Weglarz. DSS for multiobjective project scheduling subject to multiple-category resource constraints. *European Journal of Operational Research*, 79:220–229, 1994.

[26] A. Sprecher. *Resource-constrained project scheduling: Exact methods for the multi-mode case.* Number 409 in Lecture Notes in Economics and Mathematical Systems. Springer, Berlin, Germany, 1994.

[27] A. Sprecher and A. Drexl. Multi-mode resource-constrained project scheduling by a simple, general and powerful sequencing algorithm. *European Journal of Operational Research*, 107:431–450, 1998.

[28] A. Sprecher, S. Hartmann, and A. Drexl. An exact algorithm for project scheduling with multiple modes. *OR Spektrum*, 19:195–203, 1997.

[29] F. B. Talbot. Resource-constrained project scheduling with time-resource tradeoffs: The nonpreemptive case. *Management Science*, 28:1197–1210, 1982.

[30] D. Whitley, V. S. Gordon, and K. Mathias. Lamarckian evolution, the Baldwin effect and function optimization. In *Proceedings of the parallel problem solving from nature*, pages 6–15. Springer, Berlin, Germany, 1994.