Hartmann, Sönke

Self-adapting genetic algorithms with an application to project scheduling

Manuskripte aus den Instituten für Betriebswirtschaftslehre der Universität Kiel, No. 506

Provided in Cooperation with:
Christian-Albrechts-University of Kiel, Institute of Business Administration

Suggested Citation: Hartmann, Sönke (1999) : Self-adapting genetic algorithms with an application to project scheduling, Manuskripte aus den Instituten für Betriebswirtschaftslehre der Universität Kiel, No. 506, Universität Kiel, Institut für Betriebswirtschaftslehre, Kiel

This Version is available at:
https://hdl.handle.net/10419/147594

Manuskripte
aus den
Instituten für Betriebswirtschaftslehre
der Universität Kiel

No. 506

Self-Adapting Genetic Algorithms with an Application to Project Scheduling

Sönke Hartmann

Abstract

This paper introduces a new general framework for genetic algorithms to solve a broad range of optimization problems. When designing a genetic algorithm, there may be several alternatives for a component such as crossover, mutation or decoding procedure, and it may be difficult to determine the best alternative (e.g., the best crossover strategy) a priori. For such cases, we suggest to include alternative components into the genetic algorithm. Indicating the component to be actually used in the genotype, this allows the genetic algorithm to adapt itself. That is, the genetic algorithm learns which of the alternative components is the most successful by means of genetic optimization. In order to demonstrate the potential of the self-adapting genetic algorithm concept, we apply it to the classical resource-constrained project scheduling problem (RCPSP). Motivated by previous computational studies as well as theoretical insight, we employ two different decoding procedures and leave the decision which of them to select to the evolution. The approach is further enhanced by a problem-specific local search extension. An in-depth computational experimental analysis shows that the self-adapting genetic algorithm approach is currently the most promising heuristic for the RCPSP.

Keywords: Genetic Algorithm, Self-Adaptation, Local Search, Project Scheduling, Computational Results.

1 Introduction

Genetic algorithms (cf. Holland [11], Goldberg [8]) have become a popular metaheuristic strategy to tackle difficult optimization problems. They are known as robust and effective methods that allow to obtain near-optimal solutions in adequate computation times. When designing a genetic algorithm for a specific problem, one has to define the components of the algorithm such as problem representation, decoding procedure, and crossover operator. There is often a choice between several alternative components, e.g., alternative decoding procedures or alternative crossover operators. The usual approach then is to select one alternative for each component on the basis of computational experiments.

In this paper, we describe a new and more general genetic algorithm approach in which the components are not necessarily fixed. In some cases, it is difficult if not impossible to determine one alternative for a component as the best. This may occur if, e.g., the different alternatives perform differently on different sets of test instances which would make the genetic algorithm design dependent on the test set. We propose a genetic algorithm framework which allows to use several alternatives for each component. The best suited component is selected by genetic mechanisms while the genetic algorithm solves some particular problem instance. We call the resulting meta strategy *self-adapting genetic algorithm* because it automatically learns which of the component alternatives perform best.

After a description of the general self-adapting genetic algorithm framework, we demonstrate its power by applying it to the resource-constrained project scheduling problem (RCPSP). In the 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. The RCPSP is a classical and challenging optimization problem that has attracted many researchers; for overviews see Brucker et al. [4], Kolisch and Hartmann [16],

Kolisch and Padman [17], and Özdamar and Ulusoy [24]. Moreover, it is important for practitioners as well with various applications ranging from project management software to leitstand systems for production planning.

Applying the self-adapting genetic algorithm concept to the RCPSP, we show that it is useful to consider two different decoding procedures in the genetic algorithm because it cannot be decided in advance which of them will be the more promising one for an RCPSP instance to be scheduled. Therefore, we define a genetic algorithm that is capable of adapting itself by determining the decoding procedure that is found to be more successful when solving the project instance under consideration. Subsequently, we provide the results of our computational investigation in which we compared the new approach with several project scheduling heuristics from the literature.

2 Framework for Self-Adapting Genetic Algorithms

2.1 Genetic Algorithm Components

Introduced by Holland [11], genetic algorithms (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 only the best of the existing ones for recombination. For an introduction into GAs, we refer to Goldberg [8]. The classical genetic algorithm concept includes the following four basic components:

- The **fitness function** measures the quality of a solution, usually in terms of the underlying objective function, but it may also consider violations of constraints.
- The **crossover** operator defines how two (or, occasionnally, more than two) existing solutions are recombined to a new solution.
- The **mutation** operator randomly changes parts ("genes") of an existing solution.
- The **selection** operator is responsible for deciding which solutions are good enough to "survive" (and produce offspring in the next generation) and which are deleted from the population.

In many optimization problems, GAs operate on so-called representations of solutions rather than on solutions themselves. Such an approach is common if it is impossible to find appropriate genetic operators that modify existing solutions. For example, in many scheduling problems, it would be difficult to find a crossover operator that would recombine two existing solutions (i.e., schedules) to a new solution while observing all constraints of the problem at hand. Hence, in GAs, schedules are often represented by, e.g., scheduling orders or priority sequences from which the schedules have to be computed. In general, many GAs also include the following two components:

- The **representation** makes up the individuals' genotypes and encodes the solutions for the optimization problem.
- The **decoding procedure** is needed for computing the actual solution that is represented by an individual.

While these GA components are the most important ones, there are several more that can be incorporated for certain problem characteristics. As an example, consider so-called repair functions that are sometimes used to turn infeasible solutions (or their representations) into feasible ones. For a more detailed discussion of this and further GA components, we refer to Michalewicz [21]. In this paper, we focus on the six components listed above.

2.2 Designing a Genetic Algorithm

When designing a GA for a specific optimization problem, each of the general components has to be defined with respect to the requirements of the problem. Typically, there is more than one choice for a component. For example, the selection operator may be deterministic or probabilistic (cf. Michalewicz [21]), and the crossover operator may follow a one-point, two-point or uniform strategy (cf. Hartmann [9]). These choices give the designer of the GA a lot of possibilities to construct a very good heuristic. But this also means that many computational tests of alternative GA settings are necessary before really the best crossover operator or the best decoding procedure is determined, because the choices can be crucial for the success of the heuristic. Using genetic algorithm principles alone does not guarantee a good heuristic; the choice of the components seems to have a higher impact than the metaheuristic strategy (cf. Hartmann and Kolisch [10]). We will consider this issue in our computational experiments.

Consequently, the usual way to design a GA is to test as many alternative GA settings as possible on the basis of a set of test problems that is representative for the problem instances to be solved. Then the best GA variant is chosen, in other words, the combination of representation, crossover operator, decoding procedure and so on that performs best. This foregoing is appropriate for many situations. There are, however, some important cases in which this foregoing may not always lead to the best possible GA heuristic:

- The problems (and hence also the test problems) are heterogenous in the sense that for some problem characteristics one variant (e.g., one representation) performs best but for other problem characteristics another variant is better. This would make the choice of a GA variant as the overall best questionable.

- It is not a priori clear which set of test problems can be viewed as representative for the real-world application of the GA after its design is completed. Clearly, different test sets may favor different GA variants.

- A component might be better suited for longer computation times whereas an alternative one is superior for short-term optimization.

- Some component may perform well in the first few generations of the GA (which is usually characterized by a rough search not yet close to the optimum) while an alternative one is favorable for the later generations. In-depth computational studies allow to detect such an effect. In this case, considering both components in the GA is more promising that selecting only one.

2.3 Self-Adaptation

Instead of a GA in which each component is fixed by the designer, we propose a more general scheme which we call *self-adapting genetic algorithm*. The underlying idea is to include more than one alternative for a component into the GA and make the choice which alternative is actually used subject to genetic mechanisms. This means that not only the solutions of the optimization problem to be solved are subject to genetic optimization but also the GA components themselves. In other words: We extend the optimization problem from finding a good solution to finding a good GA that finds a good solution for the specific problem instance currently solved. The basic idea is to include the information which GA components are to be used into the individual's genotypes, extending the latter by one gene for each variable component.

Classical GAs operate on individuals $r_1, r_2, \ldots$ of (fixed) representation type R. Via (fixed) decoding procedure d, they lead to solutions $s_1, s_2, \ldots$ of the optimization problem to be solved, that is, $d(r_1) = s_1, d(r_2) = s_2, \ldots$. Let us for the following discussion assume that we want to use alternative decoding procedures d and d' in the GA. Now an individual's genotype consists of both the problem representation and the decoding procedure. That is, the genotype has the form $(r_1, d), (r_1, d'), (r_2, d), (r_2, d'), \ldots$. In the classical GA described above, r_1 would always be decoded by procedure d to $d(r_1)$. Now the individual additionally includes the information which decoding procedure is to be used, that is, genotype (r_1, d) leads to solution $d(r_1)$ while genotype (r_1, d') leads to solution $d'(r_1)$. Note that we may have $d(r_1) \neq d'(r_1)$ for some representation r_1 (otherwise both decoding procedures would be identical). We now need to extend the crossover operator already defined for representations of type R by stating from which of its parents a new child individual inherits the gene indicating the decoding procedure. Alternatively, we could allow only individuals with the same decoding procedure to mate (although this may be a bit too restrictive). As the GA lets only the best individuals survive, the percentage of the more promising decoding procedure (d or d') will increase in the population. This means that the better suited GA variant (with the better suited decoding procedure component) develops while it is solving the actual problem instance.

Clearly, other alternative components of the GA can also be included into an extended genotype. For example, an individual of (r, d, c, m) would contain the representation r of the solution itself, the decoding procedure d, the crossover operator c which applies when this individual mates, and the mutation operator used in its offspring. It is also possible to have more than one representation type. This, however, may lead to the particular difficulty to define a crossover operator to recombine genotypes containing different problem representations. This problem can always be considered by allowing only genotypes with representations of the same type to recombine. Generally speaking, a self-adapting GA based on an extended genotype is able to choose its components with respect to the experience gained over the previous generations spent on the actual problem instance.

It should be mentioned, though, that one should be careful when deciding which components should be chosen by the GA and which fixed in advance by the designer. Of course, it may deteriorate the GA not to consider alternative components, but using alternatives for another component may in fact also deteriorate the GA. The latter case occurs if one includes a component that is generally inferior to an alternative one because time is wasted for constructing solutions using inferior operators. Therefore, specific knowledge

of the problem to be solved as well as an in-depth computational analysis should always be the basis for designing a GA—especially a self-adapting GA.

3 The Resource-Constrained Project Scheduling Problem

Our goal is to demonstrate the applicability of the self-adapting GA using the classical resource-constrained project scheduling problem (RCPSP). This section briefly describes this problem.

We consider a project which consists of J activities (jobs) labeled $j = 1, \ldots, J$. The set of activities is referred to as $\mathcal{J} = \{1, \ldots, J\}$. Due to technological requirements, 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. Analogously, $\mathcal{S}_j$ is the set of the immediate successors of activity 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 single source and $j = J + 1$ representing the single sink activity of the network.

With the exception of the (dummy) source and (dummy) sink activity, each activity requires certain amounts of (renewable) resources to be performed. The set of resources is referred to as K. For each resource $k \in K$ the per-period-availability is constant and given by R_k. The processing time (duration) of an activity j is denoted as p_j, its request for resource k is given by r_{jk}. Once started, an activity may not be interrupted. W.l.o.g., we assume that the dummy source and the dummy sink activity have a duration of zero periods and no request for any resource.

The parameters are assumed to be nonnegative and integer valued. The objective is to determine a schedule with minimal makespan such that both the precedence and resource constraints are fulfilled. Mathematical programming formulations of the RCPSP have been given by, e.g., Demeulemeester and Herroelen [6, 7], Mingozzi et al. [22], and Sprecher [30].

4 Self-Adapting Genetic Algorithm for Project Scheduling

4.1 Basic Scheme

This section introduces a self-adapting GA for the RCPSP. It is based on a genotype which consists of a problem representation of so-called activity lists as well as information which of two available decoding procedures is to be used. That is, the self-adaptation mechanism selects the decoding procedure whereas the other components are fixed. A more detailed description of the genotype is given in Subsection 4.2.

The GA starts with the computation of an initial population, i.e., the first generation, which is described in Subsection 4.3. The number of individuals in the population is referred to as POP which is assumed to be an even integer. The GA then 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 (see Subsection 4.4) to produce two new (child) individuals. Subsequently, we apply the mutation operator (see Subsection 4.5) 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 (see Subsection 4.6) to reduce the population to its former size POP and obtain the next generation to which we again apply the crossover operator and so on. 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.

If the best solution found so far has not been improved over a given number of generations denoted as IMP and if the maximal number of generations GEN has net yet been reached, we stop the self-adapting GA prematurely and start a local search phase. The idea behind this is that the genetic search appears to be not successful in further improving the best solution found. In such a situation it seems more promising to continue with local search; exploring the neighborhood of a good solution already found may be a more efficient way to lead us very close to the optimum than the rather rough genetic search. We pick the best individual from the last population of the GA (ties are broken arbitrarily) and try to improve it by local search. If, during that local search execution, some stopping criterion is met, we skip to the next best individual of the last population for local search improvement and so on. If altogether $POP \cdot GEN$ schedules have been visited (i.e., by the GA itself and by local search) or if a global time limit has been reached, the heuristic stops. That is, denoting the last generation of the GA with G, at most $POP \cdot (GEN - G)$ schedules are generated by local search. Note that if the self-adapting GA continuously leads to improved solutions, the local search phase is not executed at all. A detailed description of the local search phase of our RCPSP heuristic in given in Subsection 4.7. The underlying local search neighborhood is outlined in Subsection 4.8.

4.2 Problem Representation and Decoding Procedure

The problem representation for the RCPSP is based on the activity list concept which performed better than other representations the computational experiments of Hartmann [9]. Within an activity list $\lambda = (j_1, \ldots, j_J)$, each of the J non-dummy activities of the project appears exactly once. We consider only precedence feasible activity lists. An activity list is called precedence feasible if an activity's predecessors occur always before that activity in the list. Formally, that is $\mathcal{P}_{j_i} \subseteq \{0, j_1, \ldots, j_{i-1}\}$ for $i = 1, \ldots, J$.

In several metaheuristics from the literature, the so-called serial schedule generation scheme (serial SGS for short) is employed as decoding procedure for activity list representations (cf., e.g., Baar et al. [1], Bouleimen and Lecocq [3], and Hartmann [9]). The serial SGS constructs schedules from activity lists as follows: First, the dummy source activity is started at time 0. Then the activities are scheduled in the order that is prescribed by the list $(j_1, \ldots, j_J)$. Thereby, each activity is assigned the earliest precedence and resource feasible start time. Clearly, the serial SGS always constructs a feasible schedule for any instance of the RCPSP.

In the context of priority rule based heuristics, another scheduling algorithm for the RCPSP, the so-called parallel SGS, has been employed (cf. Kolisch [14]). Having schedulesd the dummy sink activity at time 0, the prarallel SGS computes a so-called decision point which is the time at which an activity to be scheduled is started. This decision point is determined by earliest finish time of the activities currently in process. For each decision point, the set of eligible activities is computed as the set of those activities that can be feasibly started at the decision point. The eligible activities are selected successively

and started until none are left. Then the next decision point and a related set of eligible activities are computed. This is repeated until all activities are feasibly scheduled. We can adapt the parallel SGS to make it suitable as decoding procedure for activity lists by choosing that activity from the eligible set that has the lowest index in the activity list.

It is important to note that these two decoding procedures, the serial and the parallel SGS, are inherently different (we will see this in the computational results of Section 5). The reason for this is that the serial SGS constructs so-called active schedules whereas the parallel one constructs so-called non-delay schedules (cf. Sprecher et al. [31]). The set of the non-delay schedules is a (in most cases proper) subset of the set of the active schedules. While the set of the active schedules always contains an optimal schedule, this does not hold for the set of the non-delay schedules. Consequently, the parallel SGS may miss an optimal schedule. On the other hand, it produces schedules of good average quality because it tends to utilize the resources as early as possible, leading to compact schedules. This leads to different behavior in computational experiments (cf., e.g., Kolisch [14] and Hartmann and Kolisch [10]):

- On instances with many activities and/or scarce resource capacities, the parallel SGS performs better than the serial one.
- If long computation times are allowed (i.e., if many schedules can be computed), the serial SGS leads to better results than the parallel one.

These observations can be explained as follows: Many activities and scarce resources imply larger search spaces, where the focus on compact, but often only suboptimal non-delay schedules is a promising heuristic strategy. Longer computation times may allow to find an optimal or a near optimal solution, which is typically only possible if the serial SGS is used. However, it is often impossible to predict which SGS would perform better for a specific instance.

The idea is now to allow both SGS to be employed as decoding procedures. This allows the GA to adapt itself to the specific instance (which is from a heterogenous set of possible project instances) and to the chosen computation time. We define the genotype as follows: An individual $I = (\lambda, SGS)$ consists of an activity list λ and an indicator

$$SGS = \begin{cases} 1, & \text{if activity list } \lambda \text{ is to be decoded by the serial SGS} \\ 0, & \text{if activity list } \lambda \text{ is to be decoded by the parallel SGS.} \end{cases}$$

Having computed a schedule for an individual by the SGS specified in the genotype, the fitness of the individual is defined as the makespan of the schedule. That is, a lower fitness implies a better individual (and thus a higher chance to survive).

4.3 Initial Population

Having set up a genotype consisting of an activity list and a decoding procedure, we have to make clear how an initial population containing POP individuals of this genotype should be determined. We will proceed in two steps. First, we will consider the construction of an activity list. Second, we will describe the selection of a decoding procedure, i.e., an SGS with which an already determined actity list is to be transformed into a schedule.

An activity list is constructed by a modified serial priority rule based sampling heuristic (cf. Kolisch [14]). The next activity for the list is successively chosen from those unscheduled activities the predecessors of which have already been selected for the list. This way, we obtain a precedence feasible activity list. The decision which activity is chosen next is made on the basis of one of two well-known priority rules. We first select the priority rule; either the LFT (latest finish time) or the LST (latest start time) rule is chosen with a probability of 0.5 each. From the resulting priority values of the activities, we then derive regret based biased selection probabilities that are used to select the next activity (for details on the priority rules and the regret based biased sampling approach, we refer to Kolisch [14]). Using two good priority rules and a randomized activity selection method leads to a diversified initial population of good activity lists.

Now we have to choose an SGS in order to make the current activity list a complete genotype. We proceed as follows: The SGS for the first activity list in the initial population is the parallel one. Then, with an increasing number of completed individuals, the probability of selecting the parallel SGS decreases and, thus, the probability of selecting the serial SGS increases. Starting with an initial probability to select the parallel SGS of $p_1^{\text{parallel}} = 1$, this probability is successively updated for the i-th selection ($i = 2, \ldots, POP$) by

$$p_i^{\text{parallel}} = p_{i-1}^{\text{parallel}} \cdot \left(1 - \frac{\eta}{J^{\vartheta}}\right)$$

where $\eta > 0$ and $\vartheta \geq 1$ are parameters that determine how fast the probability decreases. Clearly, the respective probability to select the serial SGS is $1 - p_i^{\text{parallel}}$. Computational tests revealed that the parameter choices $\eta = 100$ and $\vartheta = 2$ lead to good results (considering real-world instances as well as available test sets, we assume $J > 10$).

The rationale behind this approach is that for a larger number of schedules allowed to be computed and a smaller number of activities in the project, the serial SGS yields, on the average, better results than the parallel one. Note that this method results in a "fuzzy" decision on the SGS for some genotype, leaving further decisions on the SGS to the process of genetic optimization. The SGS distribution in the initial population is guided by the probability mechanism which considers the instance to be solved and the population size, providing a good starting point for the self-adapting GA.

4.4 Crossover

Let us assume that two individuals of the current population have been selected for crossover. We have a mother individual $M = (\lambda^M, SGS^M)$ and a father individual $F = (\lambda^F, SGS^F)$. Now two child individuals have to be constructed, a daughter $D = (\lambda^D, SGS^D)$ and a son $S = (\lambda^S, SGS^S)$.

We start with a definition of the daughter D. In a first step, we determine the daughter's activity list λ^D. Combining the parent's activity lists, we have to make sure that each activity appears exactly once in the daughter's activity list. Therefore, we adapt a general crossover technique presented by Reeves [27] for permutation based genotypes. Our approach also secures that precedence feasibility is maintained. We perform a two-point crossover for which we draw two random integers q_1 and q_2 with $1 \leq q_1 < q_2 \leq J$. Now the daughter's activity list λ^D is determined by taking the activity list of the positions

$i = 1, \ldots, q_1$ from the mother, that is,

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

The positions $i = q_1 + 1, \ldots, q_2$ are derived from the father. However, the activities already selected may not be considered again. We obtain:

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

The remaining positions $i = q_2 + 1, \ldots, J$ are again taken from the mother, that is,

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

The second step determines the daughter's decoding procedure. The daughter inherits the information which SGS should be used from the mother, that is, we set

$$SGS^D := SGS^M.$$

The son individual is computed analogously. For the son's activity list, the first and the third part are taken from the father and the second one is taken from the mother. He inherits the gene that determines his decoding procedure from the father.

4.5 Mutation

The following mutation operator is applied to each newly produced child individual. The mutation operator modifies the genes of the genotype with a probability of p_{mutation}. First, we show how the mutation operator modifies the individual's activity list. With a probability of p_{mutation}, we swap activities j_i and j_{i+1} in the activity list, where i runs from 1 to $J-1$. Such a swap is executed only if the resulting activity list is precedence feasible. That is, we do not swap activities j_i and j_{i+1} if j_i is a predecessor of j_{i+1}. Second, we consider the gene indicating the decoding procedure to be applied. Interpreting the gene as a boolean indicator for the SGS to be used, we set $SGS := \neg SGS$ with a probability of p_{mutation}. That is, by applying mutation, the serial SGS is replaced by the parallel one in the current individual and vice versa. On the basis of preliminary computational experiments, we selected a mutation probability of $p_{\text{mutation}} = 0.05$.

4.6 Selection

We have tested several variants of the selection operator (cf., 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. In preliminary computational studies, we observed that the ranking method consistently gave better results than the other alternatives, confirming the results obtained in Hartmann [9]. We therefore fixed the selection component to the ranking approach.

4.7 Local Search Phase

In what follows, we discuss the local search phase that starts when the self-adapting GA stops because IMP consecutive generations have not brought a improvement of the best solution found so far.

We first clarify how to perform local search on one individual. Our intention is to employ local search in order to get very close to an optimal solution. However, as outlined in Subsection 4.2, we may miss good or optimal solutions if we use the parallel SGS because of its reduced search space. Therefore, we want to perform local search using only the serial SGS. Thus, having selected an individual $I = (\lambda, SGS)$ based on the parallel SGS (i.e., $SGS = 0$) for local search, we transform activity list λ into activity list λ' such that applying the parallel SGS to λ leads to the same schedule as applying the serial SGS to λ'. Note that such a transformation is always possible because the search space of the parallel SGS is a subset of the search space of the serial SGS (cf. Subsection 4.2). Actually, λ' is obtained from sorting the activities with respect to non-decreasing start times. Now we start the local search phase from activity list λ'. Obviously, if the selected individual is already based on the serial SGS, we do not have to modify its activity list. During the local search phase, all activity lists are decoded by means of the serial SGS.

The local search approach follows a first fit strategy. That is, we generate a neighbor of the current activity list by performing a move as defined below. Then we compute the related schedule with the serial SGS. If the resulting makespan is worse than the previous one, we reject the neighbor and keep the original list, for which we test the next neighbor. Otherwise, we keep the neighbor activity list and test one of its neighbors. If a maximal number of consecutive rejected moves (i.e., a maximal number of consecutively tested worse neighbors) has been reached, the next individual from the last GA population is selected for local search improvement. Based on preliminary computational tests, we set the maximal number of consecutive rejected moves to the number of activities J in the current project instance. Of course, if $POP \cdot GEN$ schedules have been constructed altogether during the GA and the local search phases, the heuristic stops. Clearly, we have to count all computed schedules, i.e., the accepted as well as the rejected ones, in order to check whether we have already computed $POP \cdot GEN$ solutions altogether and need to stop.

The main difference between our greedy local search procedure and simulated annealing as well as tabu search is that it never accepts worse neighbors. If we cannot find a neighbor of at least equal quality for some time, we select the next best individual from the GA population. We do so because the individuals of the last population are already the product of an optimization process. This allows us to continue the search in a possibly different, but also promising region of the search space.

Finally, we describe an approach that avoids to get back to a solution that we just left and to test a worse neighbor again. This is achieved by a list of moves that are "forbidden." As long as we reject neighbor activity lists, we put the related moves into the list in order to avoid to test a (worse) neighbor twice. Once we have accepted a new activity list, the recorded moves are deleted (and hence no longer forbidden). Having accepted a neighbor of equal quality bears the possibility of cycling, that is, the next move may lead us back to the previous activity list. As we are dealing with right shifts (see Subsection 4.8), this may only occur if the last accepted move was a shift from some position i to $i+1$. In this case

we put the same move (from i to $i+1$) into the forbidden list because this would lead us back to the previous solution. This concept to avoid cycling can be viewed as some kind of simple "tabu list." Note that cycling can only occur when a neighbor of equal quality has been accepted because worse neighbors are never accepted. If all possible moves for the current individual are forbidden, the local search method skips to the next best individual of the last GA population.

4.8 Problem-Specific Local Search Neighborhood

The neighborhood of our local search component is defined by right shift moves which, given $i \in \{1,\ldots,J-1\}$ and $h \in \{i+1,\ldots,J\}$, transform a precedence feasible activity list

$$\lambda = (j_1,\ldots,j_i,\ldots,j_h,\ldots,j_J)$$

into a precedence feasible neighbor activity list

$$\lambda' = (j_1,\ldots,j_{i-1},j_{i+1},\ldots,j_h,j_i,j_{h+1},\ldots,j_J).$$

That is, some activity j_i is right shifted within the activity list and inserted immediately after some activity j_h without violating the precedence assumption. Recall, during the local search phase, the activity lists are always decoded by the serial SGS.

Having randomly selected some position $i \in \{1,\ldots,J-1\}$ of an activity to be right shifted, we need to determine between which positions activity j_i should be allowed to be shifted. Let us first consider the most right position $\psi(i)$ which is defined as the highest index that would yield a precedence feasible neighbor activity list. If activity j_i has no (non-dummy) successor in the list, it can be shifted to the end of the list. Otherwise, we must not shift it to a position higher than one of its successors in the list. This leads to

$$\psi(i) = \begin{cases} J, & \text{if } \mathcal{S}_{j_i} \cap \mathcal{J} = \emptyset \\ \min\{k \mid j_k \in \mathcal{S}_{j_i}\} - 1, & \text{otherwise.} \end{cases} \tag{1}$$

Next, we consider the most left position $\varphi(i)$. As we are dealing with right shifts, we could set $\varphi(i) = i+1$, implying that activity j_i would have to be right shifted by at least one position. Now we could randomly draw an index $h \in \{\varphi(i),\ldots,\psi(i)\}$ and shift j_i immediately after j_h. This would result in a precedence feasible neighbor activity list. The drawback of this straightforward approach, however, is that this leads us to a different activity list but eventually not to a different schedule.

The consequence is to adapt the definition of the most left position $\varphi(i)$ in order to exclude as many right shifts as possible that do not change the schedule. To do so, we incorporate schedule-dependent knowledge into the right shift move. The foundation for this is laid by the following theorem which examines right shifts by one position.

Theorem 4.1 *Consider an activity list $\lambda = (j_1,\ldots,j_J)$ which leads, by means of the serial SGS, to schedule $S(\lambda)$ associated with start times s_{j_i} and finish times f_{j_i}, $i = 1,\ldots,J$. Let further $s_{j_i}^{\text{prec}} = \max\{f_h \mid h \in \mathcal{P}_{j_i}\}$ denote the earliest precedence feasible start time of activity j_i in schedule $S(\lambda)$. Now consider a right shift of activity j_i by one position after activity j_{i+1} with $j_i \notin \mathcal{P}_{j_{i+1}}$ in λ, leading to neighbor activity list λ' with related schedule $S(\lambda')$. If at least one of the following four conditions holds, then we have $S(\lambda) = S(\lambda')$:*

$$s_{j_i} \geq s_{j_{i+1}} \tag{2}$$

$$f_{j_i} \leq s^{\text{prec}}_{j_{i+1}} \tag{3}$$

$$s_{j_{i+1}} = s^{\text{prec}}_{j_{i+1}} \tag{4}$$

$$f_{j_i} < s_{j_{i+1}} \text{ and } s_{j_{i+1}} - 1 - s^{\text{prec}}_{j_{i+1}} < p_{j_{i+1}} \tag{5}$$

Proof. Basically, we can obtain a different schedule from such an adjacent pairwise interchange only if activity j_{i+1} can be assigned an earlier start time in schedule $S(\lambda')$ than in $S(\lambda)$. Hence, the main idea is to show that activity j_{i+1} cannot be started earlier if it is scheduled before activity j_i is scheduled. With this in mind, we now examine the four conditions separately. The conditions are illustrated in Figure 1.

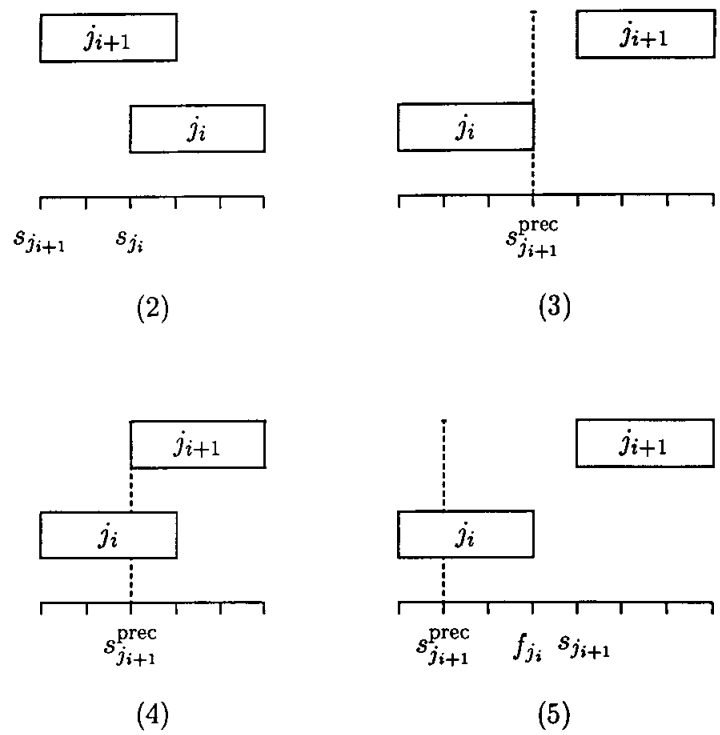

Figure 1: Illustration of conditions for equal neighbor schedules

Consider condition (2): Scheduling activity j_{i+1} before j_i cannot lead to an earlier start time of activity j_{i+1} because delaying activity j_i does not free any resources before the previous start time of activity j_{i+1}.

Condition (3) implies that activity j_{i+1} cannot be started before the previous finish time of activity j_i due to the precedence relations. Thus delaying activity j_i does not free resources that can be used by activity j_{i+1} to allow the latter to start earlier.

Condition (4) considers the case of activity j_{i+1} starting at its earliest precedence feasible start time. Clearly, it cannot be started earlier.

Finally, condition (5) works as follows: We assume $s^{\text{prec}}_{j_{i+1}} < s_{j_{i+1}}$ (otherwise, condition (4) would already be fulfilled). Activity j_{i+1} cannot be in process at $t = s_{j_{i+1}} - 1$ due to the resource constraints because delaying activity j_i would not free any resources at that time. Consequently, after the right shift, activity j_{i+1} must be finished at or before t

in order to start earlier. Therefore, activity j_{i+1} must be performed within $\{s^{\text{prec}}_{j_{i+1}}, \ldots, t\}$. This time span must be at least of the same length as its processing time; otherwise, activity j_{i+1} cannot start earlier. □

Theorem 4.1 states conditions under which a right shift by one position does not change the related schedule. Exploiting the fact that each right shift can be obtained from successively applying right shifts by one position, the following theorem extends this approach to arbitrary right shifts.

Theorem 4.2 *Consider activity list $\lambda = (j_1, \ldots, j_J)$ leading to schedule $S(\lambda)$ with start times s_{j_i}, finish times f_{j_i}, and precedence feasible start times $s^{\text{prec}}_{j_i}$ as in Theorem 4.1. Let j_i be an activity to be right shifted. Furthermore, let $l \in \{i+1, \ldots, \psi(i)\}$ denote the highest position for which at least one of the following conditions holds:*

$$s_{j_i} \geq s_{j_l} \tag{6}$$

$$f_{j_i} \leq s^{\text{prec}}_{j_l} \tag{7}$$

$$s_{j_l} = s^{\text{prec}}_{j_l} \tag{8}$$

$$f_{j_i} < s_{j_l} \text{ and } s_{j_l} - 1 - s^{\text{prec}}_{j_l} < p_{j_l} \tag{9}$$

Define $\varphi(i) := l + 1$. Then right shifting activity j_i behind any activity j_k with $k < \varphi(i)$ would lead to a schedule equal to $S(\lambda)$. Moreover, if $\varphi(i) > \psi(i)$, all right shifts of activity j_i lead to the same schedule.

Proof. Let l be as given in the theorem. We define λ_k to be the activity list obtained from right shifting activity j_i behind activity j_k in λ for $k = i+1, \ldots, l$. We have to show $S(\lambda_k) = S(\lambda)$ for all $k = i+1, \ldots, l$. This is done by induction: Assume that we have $S(\lambda_k) = S(\lambda)$ for some $k \in \{i+1, \ldots, l-1\}$. We want to show $S(\lambda_{k+1}) = S(\lambda)$.

Clearly, right shifting j_i after j_{k+1} can be done in two steps: First, j_i is right shifted after j_k in λ, leading to λ_k with $S(\lambda_k) = S(\lambda)$ according to the assumption of the induction. Second, in λ_k, j_k (the former activity j_i of λ) is right shifted by one position after j_{k+1}, leading to λ_{k+1}. As we have $k+1 \leq l$, at least one of the conditions (6)–(9) holds for the right shift of j_i after j_{k+1} in λ. Then at least one of the conditions (2)–(5) of Theorem 4.1 holds for the right shift of j_k by one position after j_{k+1} in λ_k, because j_i in λ is the same activity as j_k in λ_k and $S(\lambda_k) = S(\lambda)$. That is, Theorem 4.1 leads to $S(\lambda_{k+1}) = S(\lambda_k)$. With $S(\lambda_k) = S(\lambda)$ due to the assumption of the induction, we obtain $S(\lambda_{k+1}) = S(\lambda)$.

The second part of the proof is straightforward. □

With the definition of $\varphi(i)$ in Theorem 4.2, we have completed the definition of the neighborhood moves used in the local search procedure: After randomly selecting a position $i \in \{1, \ldots, J-1\}$ of an activity to be right shifted, we determine $\psi(i)$ as defined in (1) and $\varphi(i)$ according to Theorem 4.2. If we have $\varphi(i) > \psi(i)$, we select another position $i \in \{1, \ldots, J-1\}$ of an activity to be shifted. Otherwise, we randomly chose a position

$$h \in \{\varphi(i), \ldots, \psi(i)\}$$

in order to insert activity j_i immediately after j_h.

It should be emphasized, however, that the definition of $\varphi(i)$ does not ensure that the resulting right shift leads to a different schedule. In other words, the conditions of Theorem 4.2 (and hence also those of Theorem 4.1) are sufficient but not necessary. It does, however, exclude a large number of right shifts that would lead to the same schedule at a very low computational effort. Also excluding the remaining of such right shifts would increase the computation time needed to determine the related $\varphi(i)$ substantially. In the worst case, further excluding positions by computing an improved most left position $\varphi'(i) > \varphi(i)$ would lead to the same effort as not excluding that position and testing the respective neighbor.

5 Computational Results

5.1 Test Design

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

We have performed experiments with two different designs. First, we have taken two standard sets of RCPSP instances from the literature which were constructed by the project generator ProGen of Kolisch et al. [18]. The first set contains 480 instances with 30 activities per project while the second one consists of 600 instances with 120 activities. The self-adapting GA computed 1000 schedules for each project (with parameter settings $POP = 40$, $GEN = 25$, $IMP = 12$) and, in an additional run, 5000 schedules (with $POP = 100$, $GEN = 50$, $IMP = 25$). This test design allowed us to compare our results with those obtained for several RCPSP heuristics from the literature which were tested for the evaluation study of Hartmann and Kolisch [10]. In their study, also 1000 and 5000 schedules were computed by each heuristic for each instance. The authors of the heuristics tested their approaches themselves such that they were able to adjust the parameters in order to obtain the best possible results. As the computational effort for constructing one schedule can be assumed to be similar in all of the tested heuristics, this test design should allow for a fair comparison.

The second experimental design uses the well-known instance set assembled by Patterson [26]. It contains 110 RCPSP instances with up to 51 activities. This instance set enabled us to compare the self-adapting GA with some heuristics for which no results for the ProGen set were available. We report the results of the respective heuristics given in the literature by the authors of the approaches. Here, however, the number of schedules computed for each instance was not equal (and the results were obtained on different computers and with different computation times). As a basis for the comparison, we therefore selected a time limit of 5 seconds per instance for the self-adapting GA.

5.2 Behavior of the Self-Adapting Genetic Algorithm

This subsection analyzes the new GA approach for the RCPSP, with a focus on the mechanism of self-adaptation. We summarize the main observations of our computational experiments.

Let us first have a look at the distribution of the two alternative decoding procedures in the initial population. For the projects with 120 activities, the parallel SGS occurs more often in the initial population than for the projects with 30 activities. Computing 5000 schedules for an instance instead of only 1000 leads to a higher percentage of the serial SGS in the initial population. This is what we intended; see again the discussion of the differences between both decoding procedures in Subsection 4.2.

The percentage of the decoding procedures in the population changes over the generations. In most cases, one of the SGS dominates in the last generation, that is, it has led to better results during the evolution. There is a tendency that the parallel SGS is dominant at the end of shorter evolutions (in our case after the computation of 1000 schedules) and if larger projects are considered (in our case with 120 activities). However, the length of the evolution and the project size alone do not allow to predict which SGS will be dominant. We observed that scarce resource capacities (as measured by the so-called resource strength parameter of ProGen, cf. Kolisch et al. [18]) tend to make the parallel SGS favorable, but still we never know exactly which SGS survives. Generally, we can state that the instance sets are heterogenous with respect to the quality of the performance of the two decoding procedures. As the genetic metaphor assumes the more successful decoding procedure to survive, the difficulty to predict which SGS dominates at the end of the evolution makes self-adaptation of the GA promising.

There are some cases in which the percentage of one of the decoding procedures increases over the first generations and then decreases. This can especially be observed for the parallel SGS which, as explained in Subsection 4.2, is well suited for quickly finding schedules of good average quality but not so much for getting very close to the optimum. In other words, the advantage of the parallel SGS is exploited in the first generations while using the serial SGS pays in a later phase of the evolution. Note that the process of first increasing and later decreasing of some component's occurrance in the population is supported by the mutation operator which occasionnally reintroduces the component that has almost diminished in the current population.

We close this subsection with a brief look at the computation times of the self-adapting GA needed for computing 1000 schedules. The average computation time on the ProGen instance set with 30 activities was 0.29 seconds per instance. On the ProGen set with 120 activities, the average computation time increased to 3.07 seconds. Computing 5000 schedules takes approximately five times as long. We remark here that we have implemented two methods to speed up the self-adapting GA that have similarly been used by Kolisch [12]. They are based on the relaxation of the resource constraints. The first approach can be summarized as follows: If we have found a schedule with a makespan equal to the earliest possible project end that would be obtained from relaxing the resource constraints, we have found an optimal solution and stop the GA. The second approach makes use of the worst upper bound Z on the makespan that occurs in the current population. Proceeding from this upper bound Z, we determine the so-called latest start time LS_j for each activity $j \in \mathcal{J}$. LS_j reflects the latest time at which activity j must start to allow the project to be completed in period Z when the resource constraints are relaxed. If, while computing the schedule for a new child individual, an activity j is assigned a start time $s_j \geq LS_j$, we can stop the scheduling process for this individual and remove it from the population immediately. Clearly, the latter situation would lead to a schedule with a makespan $Z' \geq Z$, that is, it would be removed from the population by means of the

ranking selection anyway.

5.3 Comparison with other Heuristics for the RCPSP

The results of our experimental study on the ProGen instance sets are summarized in Tables 1–3. They compare the self-adapting GA with several RCPSP heuristics from the literature. The metaheuristics considered here are the schedule scheme based tabu search method of Baar et al. [1], the activity list based simulated annealing approach of Bouleimen and Lecocq [3], the activity list based GA of Hartmann [9], and the problem space based GA of Leon and Ramamoorthy [20]. The tested priority rule based sampling methods include the adaptive procedure of Kolisch and Drexl [15], the latest finish time (LFT) rule and worst case slack (WCS) rule based methods of Kolisch [14], the random sampling heuristic of Kolisch [12], and the adaptive approach of Schirmer [29]. The LFT based as well as the random sampling method was tested separately with the serial and the parallel SGS.

Table 1 gives the average percentage deviations from the optimal makespan for the ProGen instance set with 30 activities in a project obtained from the evaluation of 1000 and 5000 schedules, respectively. As for the ProGen instance set with 120 activities per project some of the optimal solutions are not known, we measured for these sets the average percentage deviation from an upper and a lower bound, respectively. The upper bound is set to the best makespan that was found by any heuristic in our study for each project; the respective results are provided in Table 2. As lower bound, we chose the critical path based lower bound (cf. Stinson et al. [32]). As this lower bound can be easily computed, this allows researchers to compare their future results with those reported here. The lower bound based results for the instances with 120 activities can be found in Table 3. In each table, the heuristics are sorted according to descending performance with respect to 5000 iterations. In order to detect significant differences in the heuristic performance for 5000 iterations, we compared the results of each heuristic with those of the next best one by means of the Wilcoxon signed-rank test using SPSS (cf. Norusis [23]). A star (*) in the last column of Tables 1 and 2 indicates that the respective heuristic performs significantly better than the next best one at the 5% level of confidence.

Finally, the results for the classical Patterson instances are provided in Table 4. In addition to the self-adapting GA, it includes the two-phase heuristic of Bell and Han [2], the extended random key based simulating annealing method of Cho and Kim [5], the activity list based GA of Hartmann [9], the random key based simulating annealing method of Lee and Kim [19], the problem space based GA of Leon and Ramamoorthy [20], the local constraint based analysis (LCBA) approach of Özdamar and Ulusoy [25], the local search procedure of Sampson and Weiss [28], and the tabu search method of Thomas and Salhi [33]. We give the average percentage deviation from the optimal makespan, the percentage of instances for which an optimal schedule was found, and information about the computation time and the computer that was used for testing. The procedures are sorted according to increasing deviation from the optimum.

The results show that the new self-adapting algorithm leads to the best results on all instance sets, outperforming several heuristics from the literature. This makes it the most promising heuristic to solve the RCPSP. For all instances of the Patterson instance set, an optimal solution is found within at most 5 seconds of CPU time (this also holds for the

Algorithm	*reference*	*Iterations* 1000	5000
self-adapting GA	(new)	0.36	0.17
simulated annealing	Bouleimen, Lecocq [3]	0.38	0.23
GA	Hartmann [9]	0.54	0.25*
adaptive sampling	Schirmer [29]	0.65	0.44
tabu search	Baar et al. [1]	0.86	0.44*
adaptive sampling	Kolisch, Drexl [15]	0.74	0.52
serial sampling (LFT)	Kolisch [14]	0.83	0.53*
serial random sampling	Kolisch [12]	1.44	1.00
parallel sampling (WCS)	Kolisch [13, 14]	1.40	1.28
parallel sampling (LFT)	Kolisch [14]	1.40	1.29*
parallel random sampling	Kolisch [12]	1.77	1.48*
problem space GA	Leon, Ramamoorthy [20]	2.08	1.59

Table 1: Average deviations from optimal solution — $J = 30$

Algorithm	*reference*	*Iterations* 1000	5000
self-adapting GA	(new)	1.66	0.49*
GA	Hartmann [9]	2.83	1.13*
simulated annealing	Bouleimen, Lecocq [3]	5.98	2.10*
adaptive sampling	Schirmer [29]	3.34	2.50
parallel sampling (LFT)	Kolisch [14]	3.15	2.56
parallel sampling (WCS)	Kolisch [13, 14]	3.18	2.58*
adaptive sampling	Kolisch, Drexl [15]	4.19	3.57*
problem space GA	Leon, Ramamoorthy [20]	5.57	4.00*
serial sampling (LFT)	Kolisch [14]	5.03	4.35*
parallel random sampling	Kolisch [12]	6.70	5.70*
serial random sampling	Kolisch [12]	9.89	8.69

Table 2: Average deviations from best solution — $J = 120$

Algorithm	*reference*	*Iterations* 1000	5000
self-adapting GA	(new)	37.33	35.60
GA	Hartmann [9]	39.37	36.74
simulated annealing	Bouleimen, Lecocq [3]	42.81	37.68
adaptive sampling	Schirmer [29]	39.85	38.70
parallel sampling (LFT)	Kolisch [14]	39.60	38.75
parallel sampling (WCS)	Kolisch [13, 14]	39.65	38.77
adaptive sampling	Kolisch, Drexl [15]	41.37	40.45
problem space GA	Leon, Ramamoorthy [20]	42.91	40.69
serial sampling (LFT)	Kolisch [14]	42.84	41.84
parallel random sampling	Kolisch [12]	44.46	43.05
serial random sampling	Kolisch [12]	49.25	47.61

Table 3: Average deviations from critical path lower bound — $J = 120$

Algorithm	*reference*	*av. dev.*	*optimal*	*CPU-sec*
self-adapting GA	(new)	0.00 %	100.0 %	5.0[a]
GA	Hartmann [9]	0.00 %	100.0 %	5.0[a]
simulated annealing	Cho, Kim [5]	0.14 %	93.6 %	18.4[b]
simulated annealing	Lee, Kim [19]	0.57 %	82.7 %	17.0[b]
problem space GA	Leon, Ramamoorthy [20]	0.74 %	75.5 %	7.5[c]
LCBA	Özdamar, Ulusoy [25]	1.14 %	63.6 %	0–25[d]
local search	Sampson, Weiss [28]	1.98 %	55.5 %	10.2[b]
tabu search	Thomas, Salhi [33]	2.30 %	46.4 %	218.7[e]
two-phase method	Bell, Han [2]	2.60 %	44.5 %	28.4[f]

[a] maximal CPU-time on a Pentium 133 MHz
[b] average CPU-time on a Pentium 60 MHz
[c] average CPU-time on an IBM RS 6000
[d] CPU-time range on an IBM PC 486
[e] average CPU-time on a Sun Sparc Station 10
[f] average CPU-time on a Macintosh plus

Table 4: Comparison of heuristics — Patterson instance set

activity list based GA of Hartmann [9]).

Observe that metaheuristics typically give better results than priority rule based methods. This is due to the fact that metaheuristics usually exploit knowledge from one or more previously examined solutions whereas priority rule based procedures generate each solution independently. It should be emphasized, however, that using a metaheuristic strategy alone does not guarantee a good performance; this can be seen from the different results of the tested GA approaches (cf. also the discussion of Subsection 2.2).

Let us now return to the difference in the behavior of the two SGS that motivated the definition of the genotype of the self-adapting GA (cf. Subsection 4.2). Consider the random sampling method and the sampling method based on the LFT priority rule. Both were tested in two variants, that is, separately with the serial and the parallel SGS. As the only difference lies in the SGS, the computational results show the impact of the choice of the SGS. On the average, the serial SGS performs better on the ProGen set with 30 activities while the parallel SGS becomes superior on the set with 120 activities, demonstrating that instance characteristics influence the performance. These results indicate that it is a promising approach to include both SGS into a GA and let the genetic operators select the more successful one—as done in our self-adapting GA.

6 Conclusions

We introduced an extension of the classical genetic algorithm paradigm called self-adapting genetic algorithm that can be applied to all kinds of optimization problems. It allows to identify several promising alternatives for each genetic algorithm component such as crossover operator, representation, and decoding procedure. The inherent survival-of-the-fittest strategy is extended to select the best components while the problem itself is solved, leading to a genetic algorithm that adapts itself by means of genetic optimization. It exploits what it learns from solving the problem by modyfying itself. That is, the evolution leads to a good solution for the problem and to a good algorithm to solve the problem at the same time. The main advantage of this approach is that the best algorithmic variant is determined automatically for the problem instance actually solved.

The general framework was applied to the well-known resource-constrained project scheduling problem. The genetic algorithm includes two alternative decoding procedures which are evaluated and selected by the self-adaptation mechanism. The approach is further extended by a local search procedure which makes use of a new problem-specific neighborhood that helps to avoid neighborhood moves that would lead back to the current schedule. A computational analysis indicated that our self-adapting genetic algorithm outperforms several state-of-the-art heuristics from the project scheduling literature, making it the best heuristic currently available for resource-constrained project scheduling. It should be emphasized that problem-specific knowledge as well as preliminary experiments led to the design of the heuristic—our computational tests also show that using a metaheuristic strategy alone does not necessarily lead to good results.

We believe that the generality of the concept, its easy applicability, and the good computational results make our self-adapting genetic algorithm framework a promising approach for many other difficult optimization problems in the future.

References

[1] T. Baar, P. Brucker, and S. Knust. Tabu-search algorithms and lower bounds for the resource-constrained project scheduling problem. In S. Voss, S. Martello, I. Osman, and C. Roucairol, editors, *Meta-heuristics: Advances and trends in local search paradigms for optimization*, pages 1–8. Kluwer, Boston, Massachusetts, 1998.

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

[3] K. Bouleimen and H. Lecocq. A new efficient simulated annealing algorithm for the resource-constrained project scheduling problem. Technical report, Université de Liège, Belgium, 1998.

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

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

[6] E. L. Demeulemeester and W. S. Herroelen. A branch-and-bound procedure for the multiple resource-constrained project scheduling problem. *Management Science*, 38:1803–1818, 1992.

[7] E. L. Demeulemeester and W. S. Herroelen. New benchmark results for the resource-constrained project scheduling problem. *Management Science*, 43:1485–1492, 1997.

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

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

[10] S. Hartmann and R. Kolisch. Experimental investigation of state-of-the-art heuristics for the resource-constrained project scheduling problem. *European Journal of Operational Research.* Forthcoming.

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

[12] R. Kolisch. *Project scheduling under resource constraints – Efficient heuristics for several problem classes.* Physica, Heidelberg, Germany, 1995.

[13] R. Kolisch. Efficient priority rules for the resource-constrained project scheduling problem. *Journal of Operations Management*, 14:179–192, 1996.

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

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

[16] 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, Amsterdam, the Netherlands, 1999.

[17] R. Kolisch and R. Padman. An integrated survey of project scheduling. Manuskripte aus den Instituten für Betriebswirtschaftslehre 463, Universität Kiel, Germany, 1997.

[18] 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.

[19] J.-K. Lee and Y.-D. Kim. Search heuristics for resource-constrained project scheduling. *Journal of the Operational Research Society*, 47:678–689, 1996.

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

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

[22] A. Mingozzi, V. Maniezzo, S. Ricciardelli, and L. Bianco. An exact algorithm for the resource-constrained project scheduling problem based on a new mathematical formulation. *Management Science*, 44:714–729, 1998.

[23] M. J. Norusis. *The SPSS guide to data analysis.* SPSS Inc., Chicago, Illinois, 1990.

[24] L. Özdamar and G. Ulusoy. A survey on the resource-constrained project scheduling problem. *IIE Transactions*, 27:574–586, 1995.

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

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

[27] C. R. Reeves. Genetic algorithms and combinatorial optimization. In V. J. Rayward-Smith, editor, *Applications of modern heuristic methods*, pages 111–125. Alfred Waller Ltd., Henley-on-Thames, 1995.

[28] S. E. Sampson and E. N. Weiss. Local search techniques for the generalized resource-constrained project scheduling problem. *Naval Research Logistics*, 40:665–675, 1993.

[29] A. Schirmer. Case-based reasoning and improved adaptive search for project scheduling. Manuskripte aus den Instituten für Betriebswirtschaftslehre 472, Universität Kiel, Germany, 1998.

[30] A. Sprecher. Solving the RCPSP efficiently at modest memory requirements. Manuskripte aus den Instituten für Betriebswirtschaftslehre 425, Universität Kiel, Germany, 1996.

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

[32] J. P. Stinson, E. W. Davis, and B. M. Khumawala. Multiple resource-constrained scheduling using branch and bound. *AIIE Transactions*, 10:252–259, 1978.

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