Published in *Naval Research Logistics* 49:433{448 (2002).

A Self-Adapting Genetic Algorithm for Project Scheduling under Resource Constraints

Sonke Hartmann

Christian-Albrechts-Universitat zu

was pointed out that the choice of the problem representation is crucial for the success of a GA. According a more recent study (cf. Hartmann and Kolisch [13]), the activity list based GA ranges among best heuristics currently available for the RCPSP. In fact, for medium and large sized project instances, it is the most promising one. This paper is a follow-up study that proposes a new RCPSP heuristic which builds upon the activity list based GA of [11]. We now examine the impact of the so-called decoding procedure which transforms the problem representation into a solution (in our case, schedule). As we will see, there are two algorithms that can be employed as decoding procedures. The idea is to leave the choice between them to the GA. Generally speaking, we obtain a GA in which an algorithmic component (the decoding procedure) is selected by means of evolution. The result is an extended GA paradigm which allows the GA to adapt itself. Hence, we call it self-adapting GA.

The basic framework for the self-adapting GA was developed in Hartmann [12]. Another general approach called adaptive GA was introduced by Derigs et al. [9] (but not yet applied to the RCPSP). The latter considers di erent crossover, mutation, and selection operators. A mechanism based on a so-called scoreboard evaluates the success of the alternative operators dynamically. The main di erences between this adaptive GA and our self-adapting GA are as follows: First, our self-adapting GA applies the same genetic operators and survival-of-the- ttest strategy to both the problem solution and the information related to self-adaptation (i.e., we do not have to employ a separate mechanism like the scoreboard). Second, in our application to the RCPSP, we use self-adaptation for selecting the decoding procedure rather than for the genetic operators (although, as we will point out later on, our approach can deal with di erent genetic operators as well).

The remainder is organized as follows: We begin with a description of the RCPSP in Section 2. Subsequently, the self-adapting GA approach is presented in Section 3. Section 4 then summarizes the results of our computational investigation. We analyze the behavior of the self-adapting GA and compare it to the plain activity list based GA without self-adaptation. Moreover, we compare it to several project scheduling heuristics from the literature. The paper closes with a general discussion of the proposed approach as well as a few remarks on research perspectives in Section 5.

2 The Resource-Constrained Project Scheduling Problem

The classical resource-constrained project scheduling problem (RCPSP) can be summarized as follows. We consider a project which consists of J activities (jobs) labeled $j = 1;:::;J$. The set of activities is referred to as $J = f1;:::;Jg$. Due to technological requirements, there are precedence relations between some of the jobs. These precedence relations are given by sets of immediate predecessors P_j indicating that an activity j may not be started before all of its predecessors are completed. Analogously, S_j is the set of the immediate successors of activity j. The precedence relations can be represented by an activityg46 0 Trtiv-(e)-TJ/msi11 9.9626(duce11 9.91nlssors)]TJ/F14 9.(5.)]TJl4c0

3 Self-Adapting Genetic Algorithm

3.1 Basic Scheme

the earliest precedence and resource feasible start time. In fact, in Hartmann [11] it is shown that the activity list representation (together with the serial SGS as decoding procedure) leads to better results than other representations for 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 [18]). It works as follows: Having scheduled the dummy sink activity at time 0, the parallel 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.

To the best of our knowledge, all metaheuristics in the RCPSP literature that employ the activity list representation also make use of the serial SGS as decoding procedure (see Baar et al. [1], Boctor [4], Bouleimen and Lecocq [5], Hartmann [11], and Pinson et al. [29]). Usually, no reason for the selection of the serial SGS is given | the serial SGS appears to be the \natural choice."

We propose to use not only the serial but also the parallel SGS as decoding procedure for the activity list representation. In fact, the parallel SGS can easily be applied to activity lists: In each

technique presented by Reeves [31] 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 q

population. The next step now is to select the individuals that survive and make up the next generation.

We have tested several variants of the selection operator which follow a survival-of-the- ttest strategy. We considered the ranking method, the proportional selection as well as the tournament selection (cf., e.g., Michalewicz [25]). In preliminary computational studies, we observed that the ranking method gave better results than the other alternatives. This is in line with the ndings of our previous study (Hartmann [11]). Therefore, we xed the selection component to the ranking approach. The ranking method sorts the individuals with respect to their tness values and selects the *POP* best ones while the remaining ones are deleted from the population (ties are broken arbitrarily).

3.7 Dealing with Clones

Crossover (together with mutation) may produce children that are copies of individuals that already exist in the population. These identical individuals are called clones. Often, clones are considered to be worth avoiding because computational e ort is wasted by computing solutions (in our case, schedules) that had been computed before. Moreover, clones reduce the genetic variety in the population. On the other hand, one might argue that the occurence of clones means that more copies of t information are available for reproduction.

In the self-adapting GA, we analyzed the number of clones occuring during the evolution. In case of small search spaces (i.e., test sets with small projects), we had at most 6% clones during the evolution. The average fraction of clones was less than 2%. For large search spaces, we hardly observed any clone. Hence, we decided not to include a speci c mechanism to avoid clones. It should be noted, however, that for very long computation times, which allow to explore a huge number of individuals, avoiding clones can by promising.

3.8 Acceleration

We have implemented two methods to speed up the self-adapting GA. They are both based on the relaxation of the resource constraints.

The rst approach has similarly been used by Kolisch [17]. It can be summarized as follows: We compute a lower bound on the makespan which is given by the earliest possible project end that would be obtained from relaxing the resource constraints. If we have found a schedule with a makespan equal to the lower bound, we have found an optimal solution and stop the GA.

The second method 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 $j \in J$. LS_j re ects 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 $s_j > LS_j$, 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 $Z' > Z$. That is, it would be removed from the population by means of the ranking selection anyway. By not completely computing a schedule for some individual, we save computation time.

4 Computational Results

4.1 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 di erent designs. First, we have taken three standard sets of RCPSP instances from the literature which were constructed by the project generator ProGen of Kolisch et al. [22]. These instance sets are available from the web-based project scheduling problem library PSPLIB (cf. Kolisch and Sprecher [21]). The rst two sets contain 480 instances with 30 and 60 activities per project, respectively. The third one consists of 600 instances with 120 activities. The self-adapting GA computed 1000 schedules for each project (with parameter settings $POP = 40$, $GEN = 25$) and, in an additional run, not more than 5000 schedules (with $POP = 90$, $GEN = 55$). 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 [13]. In that study, also 1000 and 5000 schedules were computed by each heuristic for each instance. This allows to evaluate the heuristics both in a short term and in a medium term optimization. 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 e ort 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 [28]. 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 di erent computers and with di erent computation times). As a basis for the comparison, we therefore selected a time limit of 5 seconds per instance for the self-adapting GA.

4.2 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 [5], the three GAs of Hartmann [11] based on di erent representations, and the problem space based GA of Leon and Ramamoorthy [24]. The tested priority rule based sampling methods include the adaptive procedure of Kolisch and Drexl [19], the latest nish time (LFT) rule based method of Kolisch [18], and the adaptive approach of Schirmer [33]. The LFT based 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 sets with 60 and 120 activities per project some of the optimal solutions are not known, we measured for these sets the average percentage deviation from a lower bound. As lower bound, we chose the critical path based lower bound (cf. Stinson et al. [35]). As this bound can be easily computed, this allows other researchers to compare their results with those reported here. The lower bound based results for the instances with 60 and 120 activities can be found in Tables 2 and 3, respectively. In each table, the heuristics are sorted according to descending performance with respect to 5000 iterations.

For the self-adapting GA, Table 4 additionally displays the average percentage deviation from the best lower bounds currently known. The results show that the solution gap is rather small. The underlying lower bounds have been computed by Brucker and Knust [7], Heilmann and Schwindt [14] as well as Klein and Scholl [16]. The bounds are frequently updated in the library PSPLIB of Kolisch and Sprecher [21] (the results of Table 4 are based on the bounds reported there in October 2000).

Finally, the results for the classical Patterson instances are provided in Table 5. In addition to the self-adapting GA, it includes the two-phase heuristic of Bell and Han [2], the extended random

		Iterations	
Algorithm	reference	1000	5000
self-adapting GA	(new)	0.38	0.22
simulated annealing	Bouleimen, Lecocq [5]	0.38	0.23
activity list GA	Hartmann [11]	0.54	0.25
adaptive sampling	Schirmer [33]	0.65	0.44
tabu search	Baar et al. [1]	0.86	0.44
adaptive sampling	Kolisch, Drexl [19]	0.74	0.52
serial sampling (LFT)	Kolisch [18]	0.83	0.53
random key GA	Hartmann [11]	1.03	0.56
priority rule GA	Hartmann [11]	1.38	1.12
parallel sampling (LFT)	Kolisch [18]	1.40	1.29
problem space GA	Leon, Ramamoorthy [24]	2.08	1.59

Table 1: Average deviations (%) from optimal makespan | ProGen set J = 30

		Iterations	
Algorithm	reference	1000	5000
self-adapting GA	(new)	12.21	11.70
activity list GA	Hartmann [11]	12.68	11.89
simulated annealing	Bouleimen, Lecocq [5]	12.75	11.90
adaptive sampling	Schirmer [33]	12.94	12.59
priority rule GA	Hartmann [11]	13.30	12.74
adaptive sampling	Kolisch, Drexl [19]	13.51	13.06
parallel sampling (LFT)	Kolisch [18]	13.59	13.23
random key GA	Hartmann [11]	14.68	13.32
tabu search	Baar et al. [1]	13.80	13.48
problem space GA	Leon, Ramamoorthy [24]	14.33	13.49
serial sampling (LFT)	Kolisch [18]	13.96	13.53

Table 2: Average deviations (%) from critical path lower bound | ProGen set J = 60

		Iterations	
Algorithm	reference	1000	5000
self-adapting GA	(new)	37.19	35.39
activity list GA	Hartmann [11]	39.37	36.74
simulated annealing	Bouleimen, Lecocq [5]	42.81	37.68
priority rule GA	Hartmann [11]	39.93	38.49
adaptive sampling	Schirmer [33]	39.85	38.70
parallel sampling (LFT)	Kolisch [18]	39.60	38.75
adaptive sampling	Kolisch, Drexl [19]	41.37	40.45
problem space GA	Leon, Ramamoorthy [24]	42.91	40.69
serial sampling (LFT)	Kolisch [18]	42.84	41.84
random key GA	Hartmann [11]	45.82	42.25

Table 3: Average deviations (%) from critical path lower bound | ProGen set J = 120

		Iterations	
Algorithm	ProGen set	1000	5000
self-adapting GA	J = 60	3.26	2.88
	J = 120	9.69	8.33

Table 4: Average deviations (%) from best lower bound currently known

Algorithm	reference	average dev.	optimal	CPU-sec
self-adapting GA	(new)	0.00%	100.0%	5[a]
GA	Hartmann [11]	0.00%	100.0%	5[a]
simulated annealing	Cho, Kim [8]	0.14%	93.6%	14[b]
simulated annealing	Lee, Kim [23]	0.57%	82.7%	10[b]
problem space GA	Leon, Ramamoorthy [24]	0.74%	75.5%	5[c]
LCBA	Özdamar, Ulusoy [27]	1.14%	63.6%	0{25[d]
local search	Sampson, Weiss [32]	1.98%	55.5%	20[b]
tabu search	Thomas, Salhi [36]	2.30%	46.4%	178[e]
two-phase method	Bell, Han [2]	2.60%	44.5%	48[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 5: Comparison of heuristics | Patterson instance set

key based simulating annealing method of Cho and Kim [8], the activity list based GA of Hartmann [11], the random key based simulating annealing method of Lee and Kim [23], the problem space based GA of Leon and Ramamoorthy [24], the local constraint based analysis (LCBA) approach of Özdamar and Ulusoy [27], the local search procedure of Sampson and Weiss [32], and the tabu search method of Thomas and Salhi [36]. 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 activity list based GA of Hartmann [11]).

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.

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. Consider the sampling method based on the LFT priority rule. It was 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

	Iterations	
test set	1000	5000
$J = 30$	0.30	1.40
$J = 60$	0.57	2.52
$J = 120$	3.13	14.05

Table 6: Average computation times of the self-adapting GA (sec)

Generation	1	2	5	10	15	20	25	30	35	40	45	50
$RS = 0.2$	50	30	18	18	19	20	20	20	21	21	21	21
$RS = 0.5$	50	39	35	36	36	34	34	35	36	37	38	39
$RS = 0.7$	50	44	56	64	67	67	67	67	67	67	67	67

Table 7: Average fraction (%) of the serial SGS over the generations ($J = 60$)

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. This demonstrates that instance characteristics in uence 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.

4.3 Computation Times

Let us now take a brief look at the computation times. Table 6 lists the average computation times for the three ProGen sets of test instances, both for constructing 1000 and 5000 schedules. Recall that the times were obtained on a Pentium-based computer with 133 MHz. (Note that the design of the set with $J = 120$ is di erent from the set with J

SGS	$RS = 0.2$	$RS = 0.5$	$RS = 0.7$	overall
serial	24%	58%	67%	49%
parallel	76%	42%	33%	51%

Table 8: SGS in the best solution found w.r.t. resource strength ($J = 60$)

serial SGS decreases over the first generations and then increases again. Clearly, the GA always selects the more successful SGS. This shows that the parallel SGS is useful at an early stage of the search because it is well suited for quickly finding schedules of good average quality. In contrast, the serial SGS is the better choice for getting closer to the optimum. Thus, the advantage of the parallel SGS is exploited in the beginning of the evolution while using the serial SGS pays in a later phase.

So far, we have examined how often the two SGS types occur in the population. Next, we study how often the SGS types are contained in the best solution found for each project instance. Considering again the instance set with $J = 60$, Table 8 shows the distribution of the SGS among the best solutions found. On the average, approximately half of the best solutions contain the serial SGS while the other half contain the parallel SGS. Again, however, the resource strength RS of a project instance has an impact on the SGS selection. Table 8 shows that in case of scarce resources (i.e., a low resource strength), the parallel SGS leads to more best solutions than the serial one. In case of more resource capacities, the best solutions contain the serial SGS more often than the parallel one.

Summing up, the mechanism of self-adaptation is capable of exploiting the benefits of both SGS during the genetic search. In particular, it is able to adapt the selection of the decoding procedure to the resource scarceness of a project.

5 Conclusions

In this paper, we have presented a new genetic algorithm based heuristic for the classical resource-constrained project scheduling problem. The computational experiments on a large set of standard test instances have shown that the proposed heuristic leads to better results than several heuristics from the literature.

But we have not only obtained a promising heuristic for the RCPSP. The proposed self-adapting GA can be viewed as a powerful and general framework to tackle difficult optimization problems. When designing a classical GA for some optimization problem, one would do a lot of experiments using test instances in order to find the best configuration of the GA. However, this might lead to

Some 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 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.

These disadvantages associated with the design process of a classical GA are avoided in our self-adapting GA. We suggest to let the GA decide which component or parameter setting is promising. In this paper, we have demonstrated the benefit of self-adaptation for the choice among alternative decoding procedures. But it should be emphasized that self-adaptation can also deal with, e.g., alternative crossover strategies. One simply has to integrate all promising components that can be used within the GA. The evolution can make use of additional genes in order to decide which of them are the best. 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. In other words, the best algorithmic variant is determined dynamically for the problem instance actually solved. Future research in this direction appears to be promising, and it seems to be worthwhile to develop self-adapting GAs for other

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

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

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

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

[22] 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.

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

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

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

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

[27] 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.

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

[29] E. Pinson, C. Prins, and F. Rullier. Using tabu search for solving the resource-constrained project scheduling problem. In Proceedings of the fourth international workshop on project management and scheduling, pages 102{106. Leuven, Belgium, 1994.

[30] A. A. B. Pritsker, L. J. Watters, and P. M. Wolfe. Multiproject scheduling with limited resources: A zero-one programming approach. Management Science, 16:93{107, 1969.

[31] 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.

[32] 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.

[33] A. Schirmer. Case-based reasoning and improved adaptive search for project scheduling. Naval Research Logistics , 47:201{222, 2000.

[34] A. Sprecher, R. Kolisch, and A. Drexl. Semi-active, active and non-delay schedules for the resource-constrained project scheduling problem.