A Biased Random Key Genetic Algorithm with Rollout Evaluations for the Resource Constraint Job Scheduling Problem

Christian Blum[1], Dhananjay Thiruvady[3], Andreas T. Ernst[2](✉), Matthias Horn[4], and Günther R. Raidl[4]

[1] Artificial Intelligence Research Institute (IIIA-CSIC), Campus of the UAB, Bellaterra, Spain
christian.blum@iiia.csic.es

[2] School of Mathematical Sciences, Monash University, Melbourne, Australia
andreas.ernst@monash.edu

[3] School of Information Technology, Deakin University, Geelong, VIC, Australia
Dhananjay.Thiruvady@deakin.edu.au

[4] Institute of Logic and Computation, TU Wien, Vienna, Austria
{horn,raidl}@ac.tuwien.ac.at

Abstract. The resource constraint job scheduling problem considered in this work is a difficult optimization problem that was defined in the context of the transportation of minerals from mines to ports. The main characteristics are that all jobs share a common limiting resource and that the objective function concerns the minimization of the total weighted tardiness of all jobs. The algorithms proposed in the literature for this problem have a common disadvantage: they require a huge amount of computation time. Therefore, the main goal of this work is the development of an algorithm that can compete with the state of the art, while using much less computational resources. In fact, our experimental results show that the biased random key genetic algorithm that we propose significantly outperforms the state-of-the-art algorithm from the literature both in terms of solution quality and computation time.

Keywords: Job scheduling · Genetic algorithm · Rollout evaluation

1 Introduction

The resource constraint job scheduling (RCJS) problem is an *NP*-hard scheduling problem originally motivated by a mineral supply chain application. It involves simultaneously solving multiple single machine scheduling problems subject to a shared resource constraint. In mining supply chains this arises when multiple mines plan their production with a shared rail link that connects the mines to an export port.

Due to the complexity of the RCJS problem, several methods have been developed to solve it. Exact approaches such as integer linear programming [26] and constraint programming [15] have been attempted successfully [20,21]. However, these approaches are computationally expensive and can only solve rather small instances. Hence alternatives such as metaheuristics [4] have been explored. Overall, the most effective methods so far are hybrid approaches, e.g., combinations of ant colony optimisation (ACO) and integer programming [22], ACO and constraint programming [7,21], Lagrangian relaxation and particle swarm optimisation (PSO) [9] and column generation and differential evolution [18].

Project scheduling [2,6,8,17], a very well-known class of problems, is closely related to the RCJS problem. There are two main differences: (1) in RCJS jobs must execute on the machine to which they are allocated, and (2) there is only one common shared resource. In addition most variants of project scheduling focus on minimising the makespan rather than tardiness. Brucker et al. [6] categorise project scheduling problem variants. Demeulemeester and Herroelen [8] investigate different heuristic and meta-heuristic approaches for the problem. Neumann et al. [17] tackle project scheduling with time windows and show that genetic algorithms, simulated annealing and exact approaches can be effective. Ballestin and Trautmann [2] explore a problem very similar to the RCJS problem, in which the objective is to minimise the cumulative deviation from the desired completion times of all the tasks. The approach they use is a population-based iterated local search. The studies from [5,23,24] investigate resource constrained project scheduling with the objective of maximising the net present value. Thiruvady et al. [23] show that a Lagrangian relaxation and ACO hybrid finds good heuristic solutions and upper bounds. Brent et al. [5] improve the same hybrid with a parallelisation in a multi-core shared memory architecture. Thiruvady et al. [24] show that a matheuristic derived from construct, solve, merge and adapt and parallel ACO improves upon previous approaches.

Unfortunately, current approaches require a substantial amount of computational resources, both in terms of computation time and in terms of parallel computing facilities. With the aim of deriving a computationally less intensive method, we tackle the RCJS problem in this work by means of a biased random key genetic algorithm (Brkga). This type of genetic algorithm [16] was first introduced in [11]. Since then, Brkgas have been shown to obtain excellent results for a substantial range of combinatorial optimization problems, including the maximum quasi-clique problem [19] and the project scheduling problem with flexible resources [1], to name just a few of the more recent applications. Furthermore, parallel and distributed versions of Brkga have been investigated [10,12]. Júnior *et al.* [12] explore an irregular strip packing problem and the study by Alixandre and Dorn [10] shows good performance on the CEC 2013 benchmark datasets.

2 Resource Constrained Job Scheduling

The RCJS problem consists of a number of nearly independent single machine weighted tardiness problems that are linked by a single shared resource constraint. The problem can technically be described as follows. Each job from a given set $J = \{1, \ldots, n\}$ must execute in a non-preemptive way on one specific machine from a set M of machines. Each job $j \in J$ has the following data associated with it: a release time r_j, a processing time p_j, a due time d_j, the amount g_j required from the shared resource during the jobs execution, a weight w_j, and the machine $m_j \in M$ to which it belongs. The maximum amount of shared resource available at any time is G. Precedence constraints C may apply to two jobs on the same machine: $i \rightarrow j \in C$ requires that job i completes before job j starts. The objective is to minimise the total weighted tardiness. Note that this problem is NP-Hard as the single machine weighted tardiness problem is already NP-hard [13].

This problem can be expressed in terms of a time-discretized integer linear program (ILP) as follows. Let $T = \{1, \ldots, t_{\max}\}$ be the set of considered discrete times (with $t_{\max}$ being sufficiently large), and let z_{jt} be a binary variable for all $j \in J$ and $t \in T$ that takes value one if the processing of job j completes at time t or earlier. By defining the weighted tardiness for a job j at time t as $w_{jt} := \max\{0, w_j\,(t - d_j)\}$, the resulting ILP can be stated as follows:

$$\min \quad \sum_{j\in J}\sum_{t\in T} w_{jt} \cdot (z_{jt} - z_{jt-1}) \tag{1}$$

$$\text{s.t.} \quad z_{jt_{\max}} = 1 \qquad \forall\, j \in J \tag{2}$$

$$z_{jt} - z_{jt-1} \geq 0 \qquad \forall\, j \in J,\ t \in \{2, \ldots, t_{\max}\} \tag{3}$$

$$z_{jt} = 0 \qquad \forall\, t \in T : t < r_j + p_j,\ j \in J \tag{4}$$

$$z_{bt} - z_{a,t-p_b} \leq 0 \qquad \forall\, (a, b) \in C,\ t \in T : t > r_b + p_b \tag{5}$$

$$\sum_{j\in J^i} z_{j,t+p_j} - z_{jt} \leq 1 \qquad \forall\, i \in M,\ t \in T \tag{6}$$

$$\sum_{j\in J} g_j \cdot (z_{j,t+p_j} - z_{jt}) \leq G \qquad \forall\, t \in T \tag{7}$$

$$z_{jt} \in \{0, 1\} \qquad \forall\, j \in J,\ t \in T \tag{8}$$

Equalities (2) ensure that all jobs complete by $t_{\max}$. Inequalities (3) guarantee that once a job completes it stays completed. Equalities (4) account for the release times of jobs. Inequalities (5) ensure that precedence constraints are satisfied and inequalities (6) make sure that at any time only one job is processed on a machine. Inequalities (5) require that the resource constraint on the common resource is satisfied at any time. There are many other ways to formulate this problem, but this is one of the most computationally efficient formulations [20].

3 A BRKGA for the RCJS Problem

A biased random key genetic algorithm (BRKGA) is a steady-state genetic algorithm. The main machinery of the algorithm is problem-independent. Individuals are always coded in terms of random keys, that is, vectors of floating point values in $[0, 1]$. Moreover, the population management and the crossover operator

Algorithm 1. Brkga for the RCJS Problem

```
1: input: a RCJS problem instance
2: input: parameter values for $p_{\text{size}}$, $p_e$, $p_m$ and $prob_{\text{elite}}$
3: $P := \mathsf{GenerateInitialPopulation}(p_{\text{size}})$
4: $\mathsf{Evaluate}(P)$
5: while computation time limit not reached do
6:     $P_e := \mathsf{EliteSolutions}(P, p_e)$
7:     $P_m := \mathsf{Mutants}(P, p_m)$
8:     $P_c := \mathsf{Crossover}(P, P_e, prob_{\text{elite}})$
9:     $\mathsf{Evaluate}(P_m \cup P_c)$                {NOTE: $P_e$ is already evaluated}
10:    $P := P_e \cup P_m \cup P_c$
11: end while
12: output: Best solution in $P$
```

are problem-independent as well. The only problem-dependent part is the way in which individuals are translated into valid solutions for the specific problem. The problem-independent part of the algorithm is shown in Algorithm 1. It starts by a call to function $\mathsf{GenerateInitialPopulation}(p_{\text{size}})$ in order to generate a population P of p_{size} random individuals. Hereby, each individual $\pi \in P$ is a vector of length n (the number of jobs of the RCJS instance). The value of each position j of π (denoted by $\pi(j)$) is randomly chosen from $[0, 1]$. Note that $\pi(j)$ is associated with job j of the RCJS instance. The next step consists of the evaluation of the individuals from the initial population, that is, the translation of the individuals into valid schedules for the RCJS problem, which will be explained in Sect. 3.1. As a consequence, each individual obtains its objective function value denoted by $f(\pi)$. After that, the following actions are performed at each iteration of the algorithm's main loop. First, the best $\max\{\lfloor p_e \cdot p_{\text{size}} \rfloor, 1\}$ individuals are copied over from P to P_e (function $\mathsf{EliteSolutions}(P, p_e)$). Second, a set of $\max\{\lfloor p_m \cdot p_{\text{size}} \rfloor, 1\}$ so-called mutants—that is, randomly generated individuals—are produced and stored in P_m. Next, a set P_c of $p_{\text{size}} - |P_e| - |P_m|$ new individuals are generated by crossover (function $\mathsf{Crossover}(P, P_e, prob_{\text{elite}})$). The generation of an offspring individual π_{off} by crossover works as follows: (1) an elite parent π_1 is chosen uniformly at random from P_e, (2) a second parent π_2 is chosen uniformly at random from $P \setminus P_e$, and (3) π_{off} is generated on the basis of π_1 and π_2 and stored in P_c. Hereby, value $\pi_{\text{off}}(i)$ is set to $\pi_1(i)$ with probability $prob_{\text{elite}}$, and to $\pi_2(i)$ otherwise, for all $i = 1, \ldots, n$. After generating all new offspring in P_m and P_c, these new individuals are evaluated in function $\mathsf{Evaluate}(P_m \cup P_c)$. Remember that the individuals in P_e are already evaluated. Finally, the next generations' population is obtained by the union of P_e with P_m and P_c.

3.1 Evaluation of an Individual: The Decoder

The evaluation of an individual π (lines 4 and 9 of Algorithm 1) is the problem-dependent part of the Brkga. The function that evaluates individuals is called the *decoder*. In our BRKGA implementation for the RCJS problem, the decoder involves the application of a greedy construction heuristic that was introduced in [25]. This greedy heuristic works as follows. It chooses, at each construction step, exactly one of the so-far unscheduled jobs, and provides it with a feasible starting time and, therefore, also with a finishing time. Henceforth, let $J_{\text{done}} \subseteq J$ be the set of jobs that are already scheduled, and let s_j denote the starting time of $j \in J_{\text{done}}$. At the start of the solution construction process it holds that $J_{\text{done}} := \emptyset$. The process stops when $J_{\text{done}} = J$.

Let $\max_t := \max_{j=1}^{n} r_j + \sum_{j=1}^{n} p_j$ be a crude upper bound for the makespan of any feasible solution. Moreover, let C_j be the set of jobs that – -according to the precedence constraints in C—must be executed before j, and let $M_{m_h} \subseteq J$ be the subset of jobs that must be processed on machine $m_h \in M$. Furthermore, given a partial solution, let $g_t^{\text{sum}} \geq 0$ be the sum of the already consumed resource at time t.

Given J_{done}, the set of feasible jobs—that is, the set of jobs from which the next job to be scheduled can be chosen—is defined as follows: $\hat{J} := \{j \in J \backslash J_{\text{done}} \mid C_j \cap J_{\text{done}} = C_j\}$. In words, the set of feasible jobs consists of those jobs that (1) are not scheduled yet and (2) whose predecessors are already scheduled. A time step $t' \geq 0$ is a feasible starting time for a job $j \in \hat{J}$, if and only if

1. $t' \geq s_k + p_k$, for all $k \in J_{\text{done}} \cap C_j$;
2. $t' \geq s_k + p_k$, for all $k \in M_{m_j} \cap J_{\text{done}}$ (remember that m_j refers to the machine on which job j must be processed); and
3. $g_t^{\text{sum}} + g_j \leq G$, for all $t = t', \dots, t' + p_j$.

Here T' is the set of feasible starting times for a job $j \in \hat{J}$ and the earliest possible starting time $s_j^{\min}$ is defined as $s_j^{\min} := \min\{t' \mid t' \in T'\}$. Finally, for choosing a feasible job at each construction step, the jobs from $j \in \hat{J}$ must be ordered in some way. In many scheduling applications, ordering the jobs according to their earliest possible starting times (in an increasing way) is a powerful mechanism. Therefore, our decoder combines the earliest starting time information with the numerical values of π in the following way. It produces an ordered list L of all the jobs j in $\hat{J}$ sorted according to increasing values of $\pi(j) \cdot (s_j^{\min} + 1)$. Then, the first job of L—let us call this job j^*—is chosen and added to J_{done}, and its starting time s_{j^*} is fixed to $s_{j^*}^{\min}$.

3.2 Applying the Decoder in a Rollout Fashion

Any constructive heuristic can be applied in a so-called *rollout* fashion [3]. In the context of the decoder from the previous sub-section, this works as follows. Instead of ordering the jobs $j \in \hat{J}$ at each construction step according to their $\pi(j) \cdot (s_j^{\min} + 1)$ values, the decoder is completely applied to each *partial solution* $J_{\text{done}} \cup \{j\}$, for all $j \in \hat{J}$. Hereby, the starting time of j is set to $s_j^{\min}$ in each case. This provides us with $|\hat{J}|$ complete solutions whose objective function values—henceforth called the *rollout values*—are then used for producing the ordered list L of all jobs from $\hat{J}$ (in an increasing way). As in the standard decoder, the first job of L—let us call this job again j^*—is chosen and added to J_{done}, and its starting time s_{j^*} is set to $s_{j^*}^{\min}$.

Even though applying the decoder in a rollout fashion provides better evaluations of the individuals, the computational time needed for evaluating an individual increases substantially. Therefore, we make use of the following techniques for shortening the run time:

1. We use an explicit *rollout width* $\text{ro}_{\text{width}} > 0$. In those construction steps in which $\text{ro}_{\text{width}} < |\hat{J}|$, the rollout is only applied to the first ro_{width} jobs from list L (when ordered according to the $\pi(j) \cdot (\text{st}_j^{\min} + 1)$ values). The remaining jobs in L receive a rollout value of ∞. After that, the list L is reorderd according to the rollout values, the first job from L is selected and used to extend J_{done}, before we proceed to the next construction step.
2. The decoder is only applied in a rollout fashion (with a rollout width of ro_{width}) after a number of $n_{\text{noimpr}}^{\max} \geq 0$ consecutive Brkga iterations without an improvement of the best-so-far solution. After the execution of such a Brkga iteration in which the decoder is applied in a rollout fashion, the counter for consecutive non-improving Brkga iterations is re-initialized to zero, as at the start of the Brkga algorithm.

Clearly, ro_{width} and $n_{\text{noimpr}}^{\max}$ are two important algorithm parameters that control to what extent the decoder is applied in a rollout fashion.

4 Experimental Evaluation

All experiments concerning Brkga were performed on a cluster of machines with Intel® Xeon® CPU 5670 CPUs with 12 cores of 2.933 GHz and a minimum of 32 GB RAM. As mentioned before, the current state-of-the-art results for the RCJS problem were obtained by a recent hybrid algorithm labelled Cg-De-Ls that combines column generation with differential evolution and local search see [18]. Note that, while Brkga was run in a one-threaded mode with a limit of 3600 s of CPU time for each problem instance, Cg-De-Ls was implemented in a parallel framework and each run (limited by 3600 s of wall clock time) was given 16 cores on the Monash University's Campus Cluster. Each machine of the cluster provides 24 cores and 256 GB RAM. Each physical core consists of two hyper-threaded cores with Intel Xeon E5-2680 v3 2.5 GHz, 30M Cache, 9.60GT/s QPI, Turbo, HT, 12C/24T (120W). In summary, consider that a run of Cg-De-Ls consumes at least one order of magnitude more computation time than a run of Brkga.

Problem Instances. The comparison of Brkga with Cg-De-Ls was conducted on 36 instances from a dataset that was originally introduced in [20]. This dataset consists of problem instances with the number of machines ranging from three to twenty, and there are three instances per number of machines. Each machine has to process, on average, 10.5 jobs; that is, an instance with three machines has approximately 32 jobs. Further details concerning the problem instances and their job characteristics (processing times, release times, weights, etc.) can be obtained from the original study.

Tuning of Brkga. The proposed Brkga approach has six parameters which require suitable values. In this work we made use of the automatic configuration tool irace [14] for finding such parameter values. More specifically, we aimed at identifying one parameter setting that works well for all 36 test problem instances. For this purpose, we selected six problem instances (having between 3 and 12 machines) from the additional instances provided in [20] which have not been tested in [18]. In addition, we added instances 15–3 and 20–5 from the 36 instances that will be used for the final experimentation, because [20] does not contain any other instances of that size. In total, this makes a set of eight tuning instances. The following parameter value ranges were considered:

- $p_{\text{size}} \in \{10, 50, 100, 200, 500, 1000, 5000\}$.
- $p_e \in \{0.05, 0.1, 0.15, 0.2, 0.25\}$.
- $p_m \in \{0.1, 0.15, 0.2, 0.25, 0.3\}$.
- $prob_{\text{elite}} \in \{0.5, 0.6, 0.7, 0.8, 0.9\}$.
- $\text{ro}_{\text{width}} \in \{2, 3, 5, 10, 20\}$.
- $n^{\max}_{\text{noimpr}} \in \{10, 50, 100, 200, 500\}$.

In total, we allowed a maximum of 5000 experiments—with a computation time limit of 3600 s per run—for tuning. The results provided by irace were as follows: $p_{\text{size}} = 1000$, $p_e = 0.25$, $p_m = 0.15$, $prob_{\text{elite}} = 0.5$, $\text{ro}_{\text{width}} = 3$, and $n^{\max}_{\text{noimpr}} = 200$. These parameter value settings were used for the final experimentation. The parameter settings of Cg-De-Ls (for the same set of problem instances) are described in [18].

4.1 Numerical Results

Brkga was applied ten times to all 36 considered problem instances with a CPU time limit of 3600 s per run. The numerical results—in comparison to those of Cg-De-Ls taken from [20]—are presented in Table 1. The first column provides the instance names. The following three columns show the results of Cg-De-Ls in terms of the best solution found in 30 runs (column with heading **best**), the average of the values of the 30 solutions found in 30 runs (column with heading **avg**) and the corresponding standard deviation (column with heading **std**). The same three columns (based on tens runs per problem instance) are provided for Brkga. Two additional columns provide information about the average computation time at which the best solution of each run was found and the corresponding standard deviation. Finally, note that values in columns **avg** are marked in bold font when the corresponding result is better (with statistical significance according to Student's t-test with $\alpha = 0.05$) than the result of the competing algorithm.

The results in Table 1 allow for the following observations:

Table 1. A comparison of Brkga with CG-DE-LS [18]. Both algorithms were run 30 times on each problem instance and allowed 3600 s of run-time. Statistically significant results at $\alpha = 0.05$ are shown in bold.

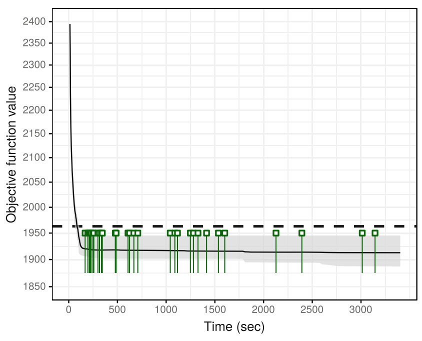
Instance	Cg-De-Ls			Brkga				
	best	avg	std	best	avg	*std*	time	std
3–5	505.00	505.00	0.0	505.00	505.00	0.0	2.5	0.5
3–23	149.07	149.29	0.7	149.07	**149.07**	0.0	13.2	30.3
3–53	69.36	69.44	0.2	69.36	**69.36**	0.0	1.0	0.2
4–28	23.81	23.91	0.1	23.81	23.93	0.10	221.5	298.4
4–42	66.73	66.92	0.3	67.64	67.64	0.0	4.4	1.1
4–61	45.96	**45.98**	0.1	45.96	46.47	0.3	696.5	680.2
5–7	252.90	253.79	1.9	253.38	**253.69**	0.4	1827.8	1260.4
5–21	168.63	168.63	0.0	168.63	168.63	0.0	8.9	2.2
5–62	249.68	256.61	2.3	249.50	255.66	2.7	979.8	1016.9
6–10	812.90	**822.45**	6.7	817.10	828.09	6.9	2209.2	1210.7
6–28	218.37	**219.02**	1.6	219.48	228.07	6.6	290.1	556.5
6–58	238.84	242.89	3.3	238.84	**241.33**	1.8	915.7	697.2
7–5	418.06	426.96	7.6	418.06	430.15	10.1	961.7	1027.6
7–23	540.60	**555.17**	5.4	553.40	557.54	4.3	826.3	843.6
7–47	404.09	420.63	7.7	412.41	**418.46**	3.9	1356.8	1116.7
8–3	619.58	634.00	9.2	618.50	**629.76**	8.8	1493.1	1116.6
8–53	449.40	459.16	6.7	442.18	**452.84**	7.4	1345.7	1192.6
8–77	1175.56	1214.36	20.4	1163.78	**1194.32**	20.5	1583.0	1044.6
9–20	871.72	887.18	6.4	877.30	**882.18**	4.5	1626.4	1209.0
9–47	1189.14	1219.74	17.6	1158.25	**1185.53**	17.4	1095.7	1148.8
9–62	1395.08	1449.99	17.5	1382.63	**1399.67**	12.4	1254.2	964.6
10–7	2401.99	2471.82	32.8	2384.04	**2400.26**	13.9	1601.8	1230.1
10–13	2100.96	2148.57	22.1	2082.71	**2106.96**	11.6	1816.7	944.3
10–31	577.54	595.37	8.9	572.03	**586.76**	11.2	2146.8	1193.3
11–21	968.12	1001.94	33.2	964.04	**973.49**	7.2	2037.4	1296.7
11–56	1748.48	1798.08	24.1	1674.49	**1694.78**	15.9	2147.0	1164.4
11–63	1963.26	1994.49	18.1	1887.17	**1912.81**	16.5	2004.2	1004.3
12–14	1670.97	1728.63	26.8	1636.39	**1658.02**	13.0	1693.3	1258.0
12–36	2799.20	2904.02	41.3	2764.17	**2796.94**	28.5	1644.7	1225.5
12–80	2319.92	2372.37	31.5	2226.67	**2258.13**	20.6	1673.1	1032.1
15–2	3797.59	3867.99	41.7	3596.50	**3627.43**	19.8	2191.4	1038.2
15–3	4174.87	4251.49	49.1	3948.22	**3994.25**	40.3	2060.4	1086.1
15–5	3378.38	3433.19	35.4	3234.74	**3275.01**	35.5	1805.3	921.2
20–2	8243.78	8339.35	58.3	7755.29	**7890.49**	66.8	2090.9	1043.8
20–5	13818.30	14120.69	163.3	12899.17	**13123.85**	138.0	2446.2	988.6
20–6	7246.64	7347.18	52.6	6907.80	**6998.20**	74.2	2243.1	742.4

– For the small-medium problem instances (see the first 14 rows of Table 1) there is no a clear pattern, with Brkga outperforming Cg-De-Ls in some cases, and vice versa in others.
– Starting from instance 7–47 (i.e., and all larger instances with 8 machines or more) Brkga clearly outperforms Cg-De-Ls. Hereby, the advantage of Brkga over Cg-De-Ls seems to grow with increasing problem instance size. In the case of the largest 11 instances, for example, the average performance of Brkga is better than the best solution values found by Cg-De-Ls.

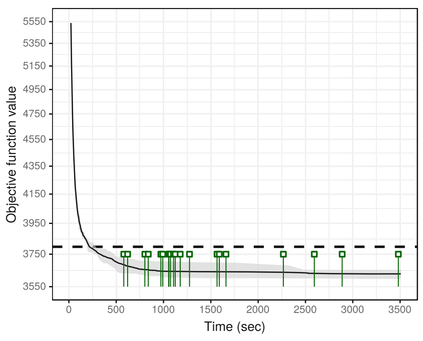
In order to better understand the behaviour of Brkga, we provide graphics about the evolution of the best-so-far solution over time for four rather large problem instances in Fig. 1. More specifically, the graphics show the mean performance of Brkga over 10 runs, while the grey-shaded area around the curves show, for each time step on the y-axis, the performance of the worst run and of the best run among the 10 runs. Furthermore, the dashed horizontal lines indicate the value of the best solutions found by Cg-De-Ls within 30 runs, where each run made use of 16 threads in parallel. Finally, the vertical bars indicate the initiation of iterations with rollout evaluations (in any of the ten runs). In those cases in which such a vertical bar has a white square head, the rollout iteration was successful in the sense that the best-so-far solution was improved. Otherwise—that is, in those cases in which such a bar has a black diamond head—the rollout iteration was not successful. Note that in the context of instances 11–63 and 15–2 (Fig. 1a and b) only the successful rollout iterations are indicated, because showing all rollout iterations would have made these graphics unreadable.

The graphics in Fig. 1 allow us to make the following conclusions:

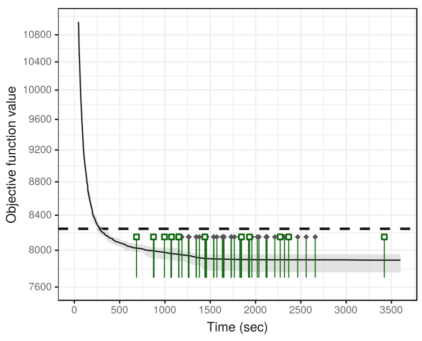
– First, in all four cases all ten runs of Brkga improve over the best solution found by Cg-De-Ls after a few hundred seconds. This is despite the fact that Cg-De-Ls makes use of 16 threads in parallel, while Brkga is run in one-threaded mode.
– Second, the best moment to make use of rollout iterations seems to be when the algorithm is stuck for quite a while in a local minimum. Remember that the parameter setting was determined by our tuning procedure with irace, as described in the third paragraph of Sect. 4. The chosen settings are $\mathrm{ro}_{\mathrm{width}} = 3$ and $n^{\max}_{\mathrm{noimpr}} = 200$, that is, a very narrow rollout-width and a rather high number of consecutive non-improving iterations before a rollout iteration is initiated. The effect of this can be nicely seen in the four graphics. In fact, the first rollout iterations are—in all four cases—initiated after the algorithm has already outperformed Cg-De-Ls. The reason for making use of rollout iteration in this way is the significant difference in computation time requirements: a standard iteration requires 0.157 s for instance 11–63, 0.34 s for instance 15–2, 0.52 s for instance 20–2, and 0.64 s for instance 20–5. In contrast, a rollout iteration requires 12.7 s for instance 11–63, 41.1 s instance for 15–2, 89.3 s for 20–2, and 115.0 s for 20–5. That is, a rollout iteration consumes about two orders of magnitude more time than a standard algorithm iteration.



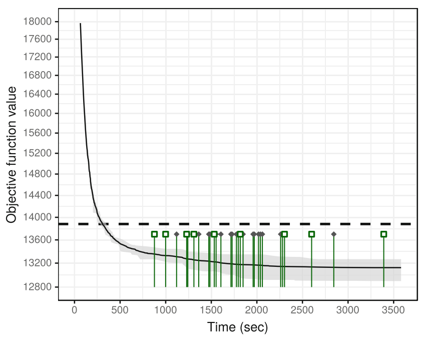
(a) Instance 11–63.



(b) Instance 15–2.



(c) Instance 20–2.



(d) Instance 20–5.

Fig. 1. Evolution of the best-so-far solution of BRKGA for four large problem instances. The curves show the mean performance over 10 runs, while the gray-shaded area behind the curves shows the spread of the 10 runs. The dashed horizontal bars indicate the best result of CG-DE-LS after 30 runs. The vertical bars indicate the initiation of rollout iterations.

Summarizing, we can say that our BRKGA algorithm significantly outperforms the current state-of-the-art algorithm CG-DE-LS, especially with growing problem instance size. Moreover, the algorithm requires much less computational resources than its competitor from the literature.

5 Conclusions and Future Work

We considered the resource constraint job scheduling problem where multiple single machine scheduling problems are linked by one limited shared resource. The objective is to minimize the total weighted tardiness of all jobs. We tackled this problem by means of a biased random key genetic algorithm, which is a quite generic framework. For the problem dependent part of the algorithm—the decoder—we apply a greedy construction heuristic which processes the jobs in an order determined by the jobs' random keys in combination with the earliest starting times. The basic greedy heuristic is further substantially enhanced by applying rollouts in a carefully controlled way in order to obtain a more promising ranking of the jobs. As rollouts are time-expensive, they are only used when the optimization gets stuck with the standard greedy criterion for a certain number of iterations.

Our experimental results show that in particular with growing problem instance size our approach significantly outperforms the leading column generation/differential evolution hybrid from the literature, both in terms of solution quality and computation time.

Acknowledgements. This work was partially funded by the Doctoral Program "Vienna Graduate School on Computational Optimization", Austrian Science Foundation (FWF) Project No. W1260-N35. Moreover, this work was partially supported by the EU H2020 Research and Innovation Program under the LOGISTAR project (Grant Agreement No. 769142).

References

1. Almeida, B.F., Correia, I., Saldanha-da Gama, F.: A biased random-key genetic algorithm for the project scheduling problem with flexible resources. Top **26**(2), 283–308 (2018)
2. Ballestin, F., Trautmann, N.: An iterated-local-search heuristic for the resource-constrained weighted earliness-tardiness project scheduling problem. Int. J. Prod. Res. **46**, 6231–6249 (2008)
3. Bertsekas, D.P., Tsitsiklis, J.N., Wu, C.: Rollout algorithms for combinatorial optimization. Journal of Heuristics **3**(3), 245–262 (1997)
4. Blum, C., Roli, A.: Metaheuristics in combinatorial optimization: overview and conceptual comparison. ACM Comput. Surv. **35**, 268–308 (2003)
5. Brent, O., Thiruvady, D., Gómez-Iglesias, A., Garcia-Flores, R.: A parallel lagrangian-ACO heuristic for project scheduling. In: IEEE Congress on Evolutionary Computation (CEC 2014), pp. 2985–2991. IEEE (2014)
6. Brucker, P., Drexl, A., Möhring, R., Neumann, K., Pesch, E.: Resource-constrained project scheduling: notation, classification, models, and methods. Eur. J. Oper. Res. **112**, 3–41 (1999)
7. Cohen, D., Gómez-Iglesias, A., Thiruvady, D., Ernst, A.T.: Resource constrained job scheduling with parallel constraint-based ACO. In: Wagner, M., Li, X., Hendtlass, T. (eds.) ACALCI 2017. LNCS (LNAI), vol. 10142, pp. 266–278. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-51691-2_23
8. Demeulemeester, E., Herroelen, W.: Project Scheduling: A Research Handbook. Kluwer, Boston (2002)
9. Ernst, A.T., Singh, G.: Lagrangian particle swarm optimization for a resource constrained machine scheduling problem. In: Li, X. (ed.) 2012 IEEE Congress on Evolutionary Computation (CEC), pp. 1–8. IEEE, Institute of Electrical and Electronics Engineers, United States (2012). https://doi.org/10.1109/CEC.2012.6256177
10. de Faria Alixandre, B.F., Dorn, M.: D-BRKGA: a distributed biased random-key genetic algorithm. In: 2017 IEEE Congress on Evolutionary Computation (CEC), pp. 1398–1405 (2017)

11. Gonçalves, J.F., Resende, M.G.C.: Biased random-key genetic algorithms for combinatorial optimization. J. Heuristics **17**(5), 487–525 (2011)
12. Júnior, B., Pinheiro, P., Coelho, P.: A parallel biased random-key genetic algorithm with multiple populations applied to irregular strip packing problems. Math. Probl. Eng. **2017**, 1–11 (2017). https://doi.org/10.1155/2017/1670709
13. Lawler, E.L.: A “pseudopolynomial” algorithm for sequencing jobs to minimize total tardiness. Ann. Discrete Math. **1**, 331–342 (1977)
14. López-Ibáñez, M., Dubois-Lacoste, J., Pérez Cáceres, L., Birattari, M., Stützle, T.: The irace package: iterated racing for automatic algorithm configuration. Oper. Res. Perspect. **3**, 43–58 (2016)
15. Marriott, K., Stuckey, P.: Programming with Constraints. MIT Press, Cambridge (1998)
16. Mitchell, M.: An Introduction to Genetic Algorithms. MIT Press, Cambridge (1998)
17. Neumann, K., Schwindt, C., Zimmermann, J.: Project Scheduling with Time Windows and Scarce Resources. Springer, Berlin (2003)
18. Nguyen, S., Thiruvady, D., Ernst, A.T., Alahakoon, D.: A hybrid differential evolution algorithm with column generation for resource constrained job scheduling. Comput. Oper. Res. **109**, 273–287 (2019)
19. Pinto, B.Q., Ribeiro, C.C., Rosseti, I., Plastino, A.: A biased random-key genetic algorithm for the maximum quasi-clique problem. Eur. J. Oper. Res. **271**(3), 849–865 (2018)
20. Singh, G., Ernst, A.T.: Resource constraint scheduling with a fractional shared resource. Oper. Res. Lett. **39**(5), 363–368 (2011)
21. Thiruvady, D., Singh, G., Ernst, A.T., Meyer, B.: Constraint-based ACO for a shared resource constrained scheduling problem. Int. J. Prod. Econ. **141**(1), 230–242 (2012)
22. Thiruvady, D., Singh, G., Ernst, A.T.: Hybrids of integer programming and ACO for resource constrained job scheduling. In: Blesa, M.J., Blum, C., Voß, S. (eds.) HM 2014. LNCS, vol. 8457, pp. 130–144. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07644-7_10
23. Thiruvady, D., Wallace, M., Gu, H., Schutt, A.: A lagrangian relaxation and ACO hybrid for resource constrained project scheduling with discounted cash flows. J. Heuristics **20**(6), 643–676 (2014)
24. Thiruvady, D., Blum, C., Ernst, A.T.: Maximising the net present value of project schedules using CMSA and parallel ACO. In: Blesa Aguilera, M.J., Blum, C., Gambini Santos, H., Pinacho-Davidson, P., Godoy del Campo, J. (eds.) HM 2019. LNCS, vol. 11299, pp. 16–30. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-05983-5_2
25. Thiruvady, D., Blum, C., Ernst, A.T.: Solution merging in metaheuristics for resource constrained job scheduling (2019, working paper)
26. Wolsey, L.A.: Integer Programming. Wiley-Interscience, New York (1998)