Biased and unbiased random-key genetic algorithms: An experimental analysis ⋆

José F. Gonçalves[1], Mauricio G. C. Resende[2], and Rodrigo F. Toso[3]

[1] Universidade do Porto, 4200-464 Porto, Portugal,
jfgoncal@fep.up.pt
[2] AT&T Labs Research, Florham Park, NJ 07932, USA,
mgcr@research.att.com
[3] Rutgers University, Piscataway, NJ 08854, USA,
rtoso@cs.rutgers.edu

Abstract. We study the runtime performance of three types of random-key genetic algorithms: the unbiased algorithm of Bean (1994); the biased algorithm of Gonçalves and Resende (2011); and a greedy version of Bean's algorithm on 12 instances from four types of covering problems: general-cost set covering, Steiner triple covering, general-cost set k-covering, and unit-cost covering by pairs. The experiments show that, in 11 of the 12 instances, the greedy version of Bean's algorithm is faster than Bean's original method and that the biased variant is faster than both variants of Bean's algorithm.

Keywords: Genetic algorithm, biased random-key genetic algorithm, random keys, combinatorial optimization, heuristics, metaheuristics, experimental algorithms.

1 Introduction

Genetic algorithms with random keys, or *random-key genetic algorithms* (RKGA), were first introduced by Bean [2]. In a RKGA, solutions are encoded as vectors of randomly generated real numbers in the interval $[0, 1)$. Though Bean proposed they be used to solve combinatorial optimization problems where solutions can be represented as permutation vectors, e.g. sequencing and quadratic assignment, they are in fact applicable to a much wider range of problems [6]. A deterministic algorithm, called a *decoder*, takes as input a solution vector and associates with it a feasible solution of the combinatorial optimization problem for which an objective value or *fitness* can be computed. In a minimization (resp. maximization) problem, we say that solutions with smaller (resp. larger) objective function values are more fit than those with larger (resp. smaller) values.

A RKGA evolves a set, or *population*, of random-key vectors, or *individuals*, over a number of iterations, or *generations*. The initial population is made up of

⋆ This research was partially supported by funds granted by the ERDF through the Programme COMPETE and by the Portuguese Government through FCT – Foundation for Science and Technology, project PTDC/ EGE-GES/ 117692/ 2010.

p real n-vectors of random keys. Each component of an initial solution vector is generated independently of each other, at random, in the real interval $[0, 1)$. After the fitness of each individual is computed by the decoder in generation k, the population is partitioned into two groups of individuals: a small group of p_e *elite* individuals, i.e. those with the best fitness values, and the remaining set of $p - p_e$ *non-elite* individuals. To evolve the population, a new generation of individuals must be produced. All elite individuals of the population of generation k are copied without modification to the population of generation $k + 1$. Mutation, in genetic algorithms as well as in biology, is key for evolution of the population. RKGAs implement mutation by introducing *mutants* into the population. A mutant is simply a vector of random keys generated the same way as an element of the initial population. At each generation, a small number (p_m) of mutants are introduced into the population. With the p_e elite individuals and the p_m mutants accounted for in population $k + 1$, $p - p_e - p_m$ additional individuals need to be produced to complete the p individuals that make up the new population. This is done by producing $p - p_e - p_m$ offspring through the process of *mating* or *crossover*, by combining pairs of individuals of the current population.

Bean [2] selects two parents at random from the entire population to implement mating in a RKGA and allows a parent to be selected more than once in a given generation. One parent is referred to as *parent A* while the other is *parent B*. A *biased random-key genetic algorithm,* or BRKGA [6], differs from Bean's algorithm in the way parents are selected for mating. In a BRKGA, each element is generated combining one element selected at random from the elite partition of the current population (this is *parent A*) and one from the non-elite partition (*parent B*). We say the selection is *biased* since an elite parent has a higher probability of being selected for mating than a non-elite parent. Repetition in the selection of a mate is allowed and therefore an individual can produce more than one offspring in the same generation. *Parameterized uniform crossover* [13] is used to implement mating in both RKGAs and BRKGAs. Let $\rho_A > 0.5$ be the probability that an offspring inherits the vector component of parent A. As before, let n denote the number of components in the solution vector of an individual. For $i = 1, \ldots, n$, the i-th component $c(i)$ of the offspring vector c takes on the value of the i-th component $a(i)$ of parent A with probability ρ_A and the value of the i-th component $b(i)$ of parent B with probability $\rho_B = 1 - \rho_A$. In this paper, we also introduce a slight variation of Bean's algorithm, which we call RKGA*, where once two parents are selected for mating, the best fit of the two takes on the role of parent A while the other is parent B. Ties are broken by rank in the sorted population.

When the next population is complete, i.e. when it has p individuals, fitness values are computed by the decoder for all of the newly created random-key vectors and the population is once again partitioned into elite and non-elite individuals to start a new generation.

As aforementioned, the role of mutants is to help the algorithm escape from local (non-global) optima. An escape occurs when a locally-optimal elite solution is combined with a mutant and the resulting offspring is better fit than both parents. Another way to avoid getting stuck in local optima is to embed the random-key genetic algorithm in a multi-start strategy. After $i_r > 0$ generations without improvement in the fitness of the best solution, the best overall solution is tentatively updated with the best fit solution in the population, the population is discarded and the algorithm is restarted.

The paper is organized as follows. In Section 2 we describe the four covering problems and in Section 3 propose random-key genetic algorithms for each problem. Experimental results comparing implementations of RKGA, RKGA*, and BRKGA for each of the four covering problems are presented in Section 4.

2 Four set covering problems

In this section, we define four set covering problems for which later, in Section 3, we propose random-key genetic algorithms. All problems addressed in this section are NP-hard [5].

Given n finite sets $P_1, P_2, \ldots, P_n$, let sets I and J be defined as $I = \cup_{j=1}^{n} P_j = \{1, 2, \ldots, m\}$ and $J = \{1, \ldots, n\}$. Associate a cost $c_j > 0$ with each element $j \in J$. A subset $J^* \subseteq J$ is called a *cover* if $\cup_{j \in J^*} P_j = I$. The cost of the cover is $\sum_{j \in J^*} c_j$. The *general-cost set covering problem* is to find a minimum cost cover.

A special case of set covering is where $c_j = 1$, for all $j \in J$. This problem is called the *unit-cost set covering problem* and its objective can be thought of as finding a cover of minimum cardinality.

The *set k-covering problem* is a generalization of the set covering problem, in which each object $i \in I$ must be covered by at least k elements of $\{P_1, \ldots, P_n\}$. Note that the general-cost set covering problem as well as the unit-cost set covering problem are special cases of set k-covering. In both, $k = 1$, and furthermore, in unit-cost set covering $c_j = 1$ for all $j \in J$.

Let $I = \{1, 2, \ldots, m\}$, $J = \{1, 2, \ldots, n\}$, and associate with each element of $j \in J$ a cost $c_j > 0$. For every pair $\{j, k\} \in J \times J$, with $j \neq k$, let $\pi(j, k)$ be the subset of elements in I covered by pair $\{j, k\}$. A subset $J^* \subseteq J$ is a cover by pairs if

$$\bigcup_{\{j,k\} \in J^* \times J^*} \pi(j, k) = I.$$

The cost of J^* is $\sum_{j \in J^*} c_j$. The *set covering by pairs problem* is to find a minimum cost cover by pairs.

3 Random-key genetic algorithms for covering

Biased random-key genetic algorithms for set covering have been proposed in [4, 8, 11]. These include a BRKGA for the Steiner triple covering problem, a unit-cost set covering problem [11], BRKGA heuristics for the set covering and set k-covering problems [8], and a BRKGA for set covering by pairs [4]. We review these heuristics in the remainder of this section.

The random-key genetic algorithms for the set covering problems of Section 2 that we describe in this section all encode solutions as a n-vector $\mathcal{X}$ of random keys, where $n = |J|$. The j-th key $\mathcal{X}_j$ corresponds to the j-th element of set J, for $j = 1, \dots, n$.

Decoding is similar for all four covering problems. The decoding scheme has three steps. In step 1, a tentative cover solution J^* is constructed by placing in J^* all elements $j \in J$ for which $\mathcal{X}_j > 1/2$. If J^* is a feasible cover, then step 2 is skipped. Otherwise, in step 2, a greedy algorithm is used to construct a valid cover starting from J^*. Later in this section, we describe these greedy algorithms. Finally, in step 3, a local improvement procedure is applied to the cover. Later in this section, we describe the different local improvement procedures.

The decoder not only returns the cover J^* but also modifies the vector of random keys $\mathcal{X}$ such that it decodes directly into J^* with the application of only the first phase of the decoder. To do this we reset $\mathcal{X}$ as follows:

$$\mathcal{X}_j = \begin{cases} \mathcal{X}_j & \text{if } \mathcal{X}_j \geq 0.5 \text{ and } j \in J^* \\ 1 - \mathcal{X}_j & \text{if } \mathcal{X}_j < 0.5 \text{ and } j \in J^* \\ \mathcal{X}_j & \text{if } \mathcal{X}_j < 0.5 \text{ and } j \notin J^* \\ 1 - \mathcal{X}_j & \text{if } \mathcal{X}_j \geq 0.5 \text{ and } j \notin J^*. \end{cases}$$

We use two greedy algorithms. The first is for set k-covering and its special cases, set covering and unit-cost set covering. The second one is for covering by pairs.

A greedy algorithm for set covering [7] starts from the partial cover J^* defined by $\mathcal{X}$. This greedy algorithm proceeds as follows. While J^* is not a valid cover, add to J^* the smallest index $j \in J \setminus J^*$ for which the inclusion of j in J^* corresponds to the minimum ratio π_j of cost c_j to number of yet-uncovered elements of I that become covered with the inclusion of j in J^*. For the special case of unit-cost set covering, this reduces to adding the smallest index $j \in J \setminus J^*$ for which the inclusion of j in J^* maximizes the number π_j of yet-uncovered elements of I that become covered with the inclusion of j in J^*. In this iterative process, we use a binary heap to store the π_j values of unused columns, allowing us to retrieve a column j with largest π_j value in $O(\log m)$-time and update the π values of the remaining columns in $O(\log n)$-time after column j is added to the solution.

A greedy algorithm for unit-cost covering by pairs is proposed in Breslau et al. [4]. It starts with set J^* defined by $\mathcal{X}$. Then, as long as J^* is not a valid cover, find an element $j \in J \setminus J^*$ such that $J^* \cup \{j\}$ covers a maximum number of yet-uncovered elements of I. Ties are broken by the index of element j. If the number of yet-uncovered elements of I that become covered is at least one, add j to J^*. Otherwise, find a pair of $\{j_1, j_2\} \subseteq J \setminus J^*$ such that $J^* \cup \{j_1\} \cup \{j_2\}$ covers a maximum number of yet-uncovered elements in I. Ties are broken first by the index of j_1, then by the index of j_2. If such a pair does not exist, then the problem is infeasible. Otherwise, add j_1 and j_2 to J^*.

Given a cover J^*, the local improvement procedure attempts to make elementary modifications to the cover with the objective of reducing its cost. We use two types of local improvement procedures: *greedy uncover* and *1-opt*. In the decoder for set k-covering we apply greedy uncover, followed by 1-opt, followed by greedy uncover if necessary. In the special case of unit-cost set covering only greedy uncover is used. The decoder implemented for set covering by pairs does not make use of any local improvement procedure. Instead, greedy uncover is applied to each elite solution after the iterations of the genetic algorithm end. In the experiments described in Section 4, this local improvement is not activated.

Given a cover J^*, *greedy uncover* attempts to remove superfluous elements of J^*, i.e. elements $j \in J^*$ such that $J^* \setminus \{j\}$ is a cover. This is done by scanning the elements $j \in J^*$ in decreasing order of cost c_j, removing those elements that are superfluous.

Given a cover J^*, *1-opt* attempts to find an element $j \in J^*$ and another element $i \notin J^*$ such that $c_i < c_j$ and $J^* \setminus \{j\} \cup \{i\}$ is a cover. If such a pair is found, i replaces j in J^*. This is done by scanning the elements $j \in J^*$ in decreasing order of cost c_j, searching for an element $i \notin J^*$ that covers all elements of I left uncovered with the removal of each $j \in J^*$.

4 Computational experiments

We report in this section computational results with the three variants of random-key genetic algorithms introduced in Section 1. They are the biased variant – BRKGA, Bean's unbiased variant – RKGA, and – RKGA*, the variant of Bean's algorithm that assigns the best fit parent the role of parent A.

Our goal in these experiments is to compare the performance of these three types of heuristics on different problem instances and show that the BRKGA variant is the most effective of the three.

In the experiments, we consider four problem types: general-cost set covering (instances `scp41`, `scp51`, and `scpa1` of Beasley [3]), Steiner triple (unit-cost) covering (instances `stn135`, `stn243`, and `stn405` of Resende et al. [11]), general-cost set k-covering (instances `scp41-2`, `scp45-11`, and `scp48-7` of Pessoa et al. [9]), and unit-cost covering by pairs (instances `n558-i0-m558-b140`, `n220-i0-m220-b220`, and `n190-i9-m190-b95` of Breslau et al. [4]).

The experiment consists in running the three variants 100 times on each of the 12 instances. Therefore, the genetic algorithms are run a total of 3600 times. Each run is independent of the other and stops when a solution with cost at least as good as a given target solution value is found. The objective of these runs is to derive empirical runtime distributions, or time-to-target (TTT) plots [1], for each of the 36 instance/variant pairs and then estimate the probabilities that a variant is faster than each of the other two, using the iterative method proposed in Ribeiro et al. [12].

Table 1 lists the instances, their dimensions, the values of the target solutions used in the experiments, and the best solution known to date. Of the 12 instances, a target solution value equal to the best known solution was used on 10 instances while on two (`stn405` and `n558-i0-m558-b140`) larger values were used. The best known solution to this date for `stn405` is 435 and we used a target solution

Table 1. Test instances used in the computational experiments. For each instance, the table lists its class, name, dimensions, value of the target solution used in the experiments, and the value of the best known solution to date.

Problem class	Instance name	m	n	k	Triples	Target	BKS
General set covering	`scp41`	200	1000	1	–	429	429
	`scp51`	200	2000	1	–	253	253
	`scpa1`	300	3000	1	–	253	253
Steiner triple covering	`stn135`	3015	135	1	–	103	103
	`stn243`	9801	243	1	–	198	198
	`stn405`	27270	405	1	–	339	335
Set k-covering	`scp41-2`	200	1000	2	–	1148	1148
	`scp45-11`	200	1000	11	–	188856	188856
	`scp48-7`	200	1000	7	–	8421	8421
Covering by pairs	`n558-i0-m558-b140`	558	140	1	1,301,314	55	50
	`n220-i0-m220-b220`	220	220	1	289,657	62	62
	`n190-i9-m190-b95`	190	95	1	173,030	37	37

value of 439 while for `n558-i0-m558-b140` the best known solution is 50 and we used 55. This was done since Bean's algorithm did not find the best known solutions for these instances after repeated attempts.

All algorithms were implemented in C++ using the BRKGA Application Programming Interface (API) of Toso and Resende [14]. The parameter settings for each problem class are shared by all three variants (BRKGA, RKGA, and RKGA*). These parameters are listed in Table 2. For each problem class, the table lists its name, population size, the sizes of the elite and mutant sets, the probability that the offspring will inherit the key of parent A, and the number of iterations without improvement of the incumbent solution that triggers a restart of the multi-start algorithm in which the genetic algorithms are embedded. For each problem class, variants BRKGA, RKGA, and RKGA* share the same C++ code, differing only in how parents are selected and which parent is assigned the role of parent A. This eliminates any differences in performance that could be attributed to coding.

Recall that our goal is to derive runtime distributions for the three heuristics on the set of 12 instances. Runtime distributions, or time-to-target plots, are useful tools for comparing running times of stochastic search algorithms. Since the experiments involve running the algorithms 3600 times, with some very long runs, we distributed the experiment over several heterogeneous computers. Since CPU speeds vary among the computers used for the experiment, instead of producing runtime distributions directly, we first derive computer-independent *iteration count distributions* and use them to subsequently derive

Table 2. Parameter settings used in the computational experiments. For each problem class, the table lists its name and the following parameters: size of population (p), size of elite partition (p_e), size of mutant set (p_m), inheritance probability (ρ_A), and number of iterations without improvement of incumbent that triggers a restart (i_r).

Problem class	p	p_e	p_m	ρ_A	i_r
General set covering	$10 \times m$	$\lceil 0.2 \times p \rceil$	$\lceil 0.15 \times p \rceil$	0.70	200
Steiner triple covering	$10 \times n$	$\lceil 0.15 \times p \rceil$	$\lceil 0.55 \times p \rceil$	0.65	200
Set k-covering	$10 \times m$	$\lceil 0.2 \times p \rceil$	$\lceil 0.15 \times p \rceil$	0.70	200
Covering by pairs	m	$\lceil 0.2 \times p \rceil$	$\lceil 0.15 \times p \rceil$	0.70	200

runtime distributions. To do this, we multiply iteration counts for each heuristic/instance pair by their corresponding mean running time per iteration. Mean running times per iteration of each heuristic/instance pair are estimated on an 8-thread computer with an Intel Core i7-2760QM CPU running at 2.40GHz. On the 12 instances, we ran each heuristic independently 10 times for 100 generations and recorded the average running (user) time. User time is the sum of all running times on all threads and corresponds to the running time on a single processor. These times are listed in Table 3.

Figures 1 and 2 show iteration count distributions for the three heuristics on each of the 12 problem instances that make up the experiment. Suppose that for a given variant, all 100 runs find a solution at least as good as the target and let $t_1, t_2, \ldots, t_{100}$ be the corresponding iteration counts sorted from smallest to largest. Each iteration count distribution plot shows the pairs of points

$$\{t_1, .5/100\}, \{t_2, 1.5/100\}, \ldots, \{t_{100}, 99.5/100\},$$

connected sequentially by lines. For each heuristic/instance pair and target solution value, the point $\{t_i, (i-0.5)/100\}$ on the plot indicates that the probability that the heuristic will find a solution for the instance with cost at least as good as the target solution value in at most t_i iterations is estimated to be $(i-0.5)/100$, for $i = 1, \ldots, 100$.

For each heuristic/instance pair, let τ denote the average CPU time for one iteration of the heuristic on the instance. Then a runtime distribution plot can be derived from an iteration count distribution plot with the pairs of points

$$\{\tau \times t_1, .5/100\}, \{\tau \times t_2, 1.5/100\}, \ldots, \{\tau \times t_{100}, 99.5/100\}.$$

For each heuristic/instance pair and target solution value, the point $\{\tau \times t_i, (i-0.5)/100\}$ on the plot indicates that the probability that the heuristic will find a solution for the instance with cost at least as good as the target solution value in at most time $\tau \times t_i$ is estimated to be $(i-0.5)/100$, for $i = 1, \ldots, 100$.

Table 3. Average CPU time per 100 generations for each problem and each algorithm. For each instance, the table lists the average CPU times (in seconds on an Intel Core i7-2760QM CPU at 2.40GHz) for 100 generations of heuristics BRKGA, RKGA, and RKGA*. Averages were computed over 10 independent runs of each heuristic.

Instance name	BRKGA	RKGA	RKGA*
`scp41`	21.01	24.71	22.67
`scp51`	29.64	34.20	31.57
`scpa1`	68.30	82.82	77.73
`stn135`	17.61	18.05	18.43
`stn243`	134.67	137.99	137.20
`stn405`	769.87	777.37	773.80
`scp41-2`	43.28	50.76	47.08
`scp45-11`	398.94	412.35	404.60
`scp48-7`	211.79	231.89	218.87
`n558-i0-m558-b140`	318.36	426.89	386.53
`n220-i0-m220-b220`	34.62	43.14	39.81
`n190-i9-m190-b95`	12.55	15.95	14.26

Visual examination of time-to-target plots usually allow easy ranking of heuristics. However, we may wish to quantify these observations. To do this we rely on the iterative method described in Ribeiro et al. [12]. Their iterative method takes as input two sets of k time-to-target values $\{t_a^1, t_a^2, \ldots, t_a^k\}$ and $\{t_b^1, t_b^2, \ldots, t_b^k\}$, drawn from unknown probability distributions, corresponding to, respectively, heuristics a and b, and estimates $\Pr(t_a \leq t_b)$, the probability that $t_a \leq t_b$, where t_a and t_b are the random variables time-to-target solution of heuristics a and b, respectively. Their method iterates until the error is less than 0.001. Applying their procedure to the sets of time-to-target values collected in the experiment, we compute $\Pr(t_{BRKGA} \leq t_{RKGA})$, $\Pr(t_{BRKGA} \leq t_{RKGA^*})$, and $\Pr(t_{RKGA^*} \leq t_{RKGA})$ for all 12 instances.[4] These values are shown in Table 4.

We conclude this section with some remarks about the experiment.

On all instances BRKGA was the fastest per iteration of the three heuristics while on all but one instance (`stn135`), RKGA* was faster per iteration than RKGA. BRKGA was as high as 34% faster than RKGA (on `n558-i0-m558-b140`) to as low as 1% faster (on `stn405`) while it was as high as 21% faster than RKGA* (on `n558-i0-m558-b140`) to as low as 0.5% faster (on `stn405`). These differences in running times per iteration can be explained by the fact that the incomplete greedy algorithms as well as the local searches implemented in the decoders take

[4] The authors of [12] kindly shared with us a perl script implementation of their iterative method to compute the probabilities.

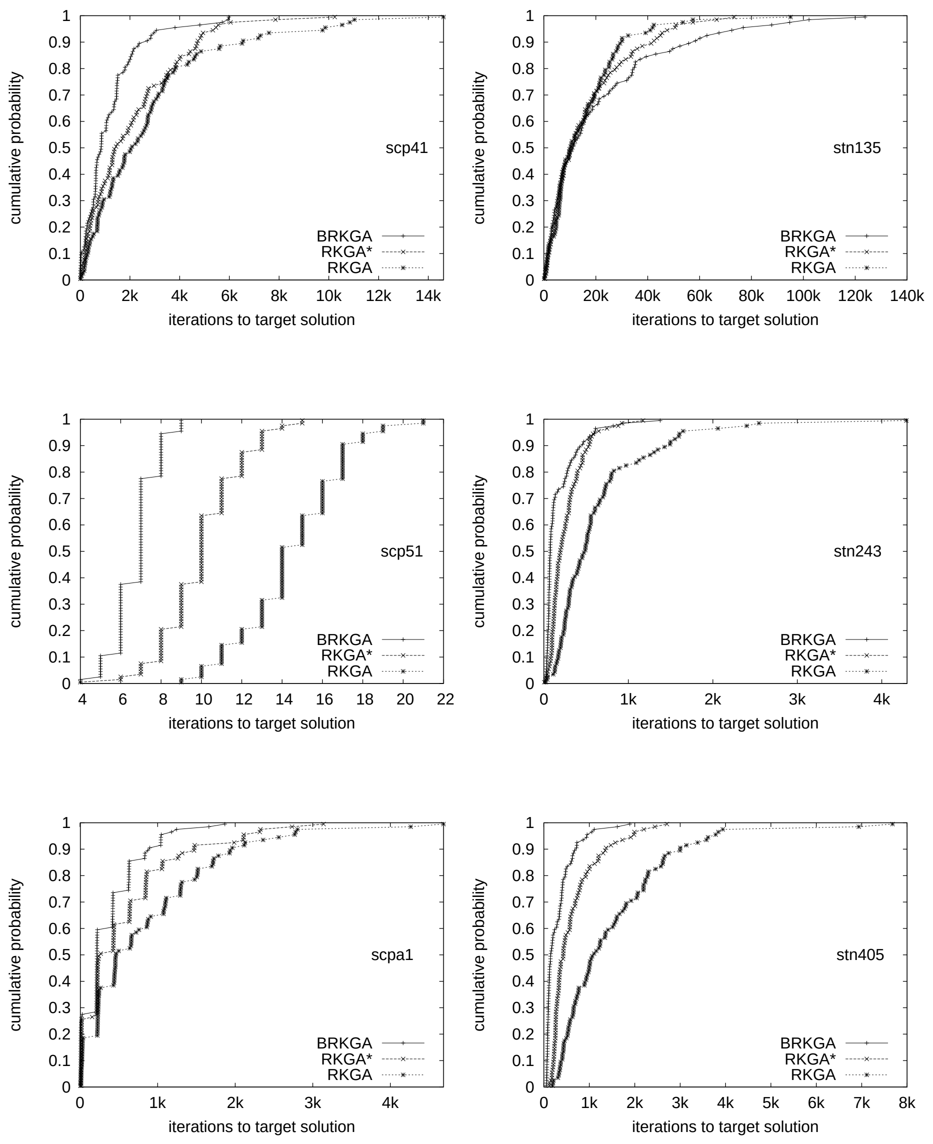

Fig. 1. Iteration count distributions for general-cost set covering instances `scp41`, `scp51`, and `scpa1` with target values 429, 253, and 253, respectively (on the left from top to bottom) and for Steiner triple covering instances `stn135`, `stn243`, and `405` with target values 103, 198, and 439, respectively (on the right from top to bottom).

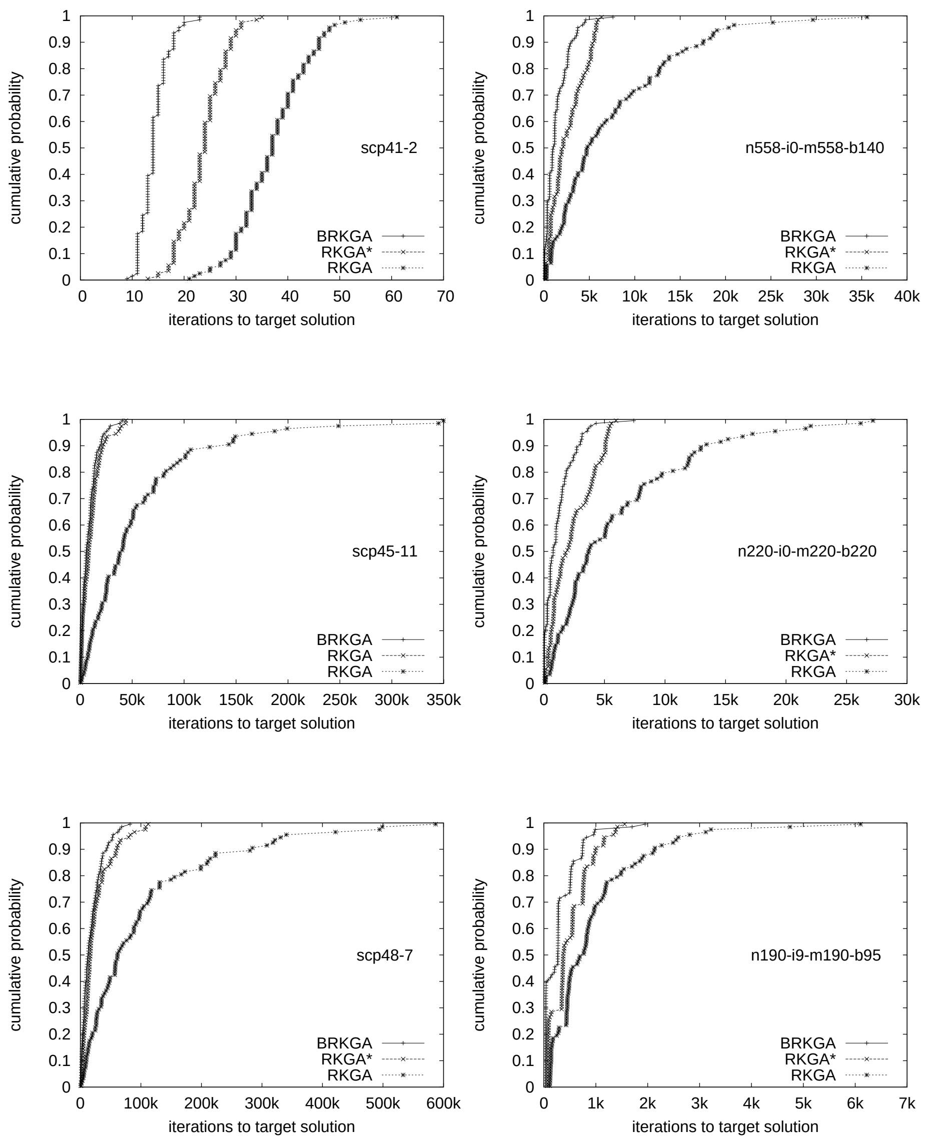

Fig. 2. Iteration count distributions for general-cost set k-covering instances `scp41-2`, `scp45-11`, and `scp48-7` with target values 1148, 18856, and 8421, respectively (on the left from top to bottom) and for unit-cost covering by pairs instances `n558-i0-m558-b558`, `n220-i0-m220-b220`, and `n190-i9-m190-b95` with target values 55, 62, and 37, respectively (on the right from top to bottom).

Table 4. Probability that random variable "time-to-target solution" of BRKGA (t_{BRKGA}) will be less than that of RKGA (t_{RKGA}) and RKGA* (t_{RKGA^*}) and that random variable "time-to-target solution" of RKGA* will be less than that of RKGA on an Intel Core i7-2760QM CPU at 2.40GHz. Computed for empirical runtime distributions of the heuristics using the method of Ribeiro et al. [12] with error 0.001.

Instance name	$\Pr(t_{BRKGA} \leq t_{RKGA})$	$\Pr(t_{BRKGA} \leq t_{RKGA^*})$	$\Pr(t_{RKGA^*} \leq t_{RKGA})$
`scp41`	0.740	0.652	0.588
`scp51`	0.999	0.960	0.943
`scpa1`	0.733	0.643	0.642
`stn135`	0.485	0.496	0.489
`stn243`	0.864	0.730	0.768
`stn405`	0.917	0.721	0.859
`scp41-2`	0.999	0.975	0.975
`scp45-11`	0.881	0.547	0.854
`scp48-7`	0.847	0.591	0.797
`n558-i0-m558-b140`	0.892	0.743	0.754
`n220-i0-m220-b220`	0.883	0.735	0.734
`n190-i9-m190-b95`	0.841	0.728	0.701

longer to converge when starting from random vectors, i.e. vectors containing keys mostly from mutants (recent descendents of mutants). This behavior has been also observed in GRASP heuristics where variants with restricted candidate list (RCL) parameters leading to more random constructions usually take longer than those with more greedy constructions [10]. Of the three variants, RKGA is the most random, while BRKGA is the least. The exception is in the Steiner triple covering class where the algorithms have parameter settings that make them near equally random. In that class, we observe the most similar running times per iteration.

With respect to iteration count distribution, we observe visually that with the exception of instance `stn135`, BRKGA dominates both RKGA and RKGA* and RKGA* dominates RKGA. The parameter settings of the BRKGA for Steiner triple covering in Resende et al. [11] were such that the resulting heuristic was more random than BRKGAs used to solve other problems. The less random parameter settings simply did not result in BRKGAs that were as effective as the more random variant. With respect to `stn135`, it appears that even more randomness results in improved performance. Of the three heuristics, the most random is RKGA while the least random is BRKGA.

Putting together the measured average times per iteration and iteration count distributions allows us to compute the probabilities listed in Table 4. They show, with the exception of instance `stn135`, a clear domination of BRKGA over RKGA and RKGA*, and of RKGA* over RKGA.

Though we have confined this study only to set covering problems, we have observed that the described relative performances of the three variants occurs on most, if not all, other problems we have tackled with biased random-key genetic algorithms [6].

References

1. Aiex, R., Resende, M., Ribeiro, C.: TTTPLOTS: A perl program to create time-to-target plots. Optimization Letters **1** (2007) 355–366
2. Bean, J.C.: Genetic algorithms and random keys for sequencing and optimization. ORSA J. on Computing **6** (1994) 154–160
3. Beasley, J.: OR-Library: Distributing test problems by electronic mail. Journal of the Operational Research Society **41** (1990) 1069–1072
4. Breslau, L., Diakonikolas, I., Duffield, N., Gu, Y., Hajiaghayi, M., Johnson, D., Resende, M., Sen, S.: Disjoint-path facility location: Theory and practice. In: ALENEX 2011: Workshop on algorithm engineering and experiments. (January 2011)
5. Garey, M., Johnson, D.: Computers and intractability. A guide to the theory of NP-completeness. W.H. Freeman and Company, San Francisco, Calif (1979)
6. Gonçalves, J., Resende, M.: Biased random-key genetic algorithms for combinatorial optimization. J. of Heuristics **17** (2011) 487–525
7. Johnson, D.: Approximation algorithms for combinatorial problems. Journal of Computer and System Sciences **9** (1974) 256–278
8. Pessoa, L., , Resende, M., Toso, R.: Biased random-key genetic algorithms for set k-covering. Technical report, AT&T Labs Research, Florham Park, New Jersey (2011)
9. Pessoa, L., Resende, M., Ribeiro, C.: A hybrid Lagrangean heuristic with GRASP and path relinking for set k-covering. Computers and Operations Research (2012) Published online 3 January.
10. Resende, M., Ribeiro, C.: Greedy randomized adaptive search procedures: Advances and applications. In Gendreau, Potvin, J.Y., eds.: Handbook of Metaheuristics. 2nd edn. Springer (2010) 281–317
11. Resende, M., Toso, R., Gonçalves, J., Silva, R.: A biased random-key genetic algorithm for the Steiner triple covering problem. Optimization Letters **6** (2012) 605–619
12. Ribeiro, C., Rosseti, I., Vallejos, R.: Exploiting run time distributions to compare sequential and parallel stochastic local search algorithms. J. of Global Optimization **54** (2012) 405–429
13. Spears, W.M., DeJong, K.A.: On the virtues of parameterized uniform crossover. In: Proceedings of the Fourth International Conference on Genetic Algorithms. (1991) 230–236
14. Toso, R., Resende, M.: A C++ application programming interface for biased random-key genetic algorithms. Technical report, AT&T Labs Research, Florham Park, New Jersey (2012)