Pesquisa Operacional (2014) 34(2): 143-164

Printed version ISSN 0101-7438 / Online version ISSN 1678-5142
www.scielo.br/pope
doi: 10.1590/0101-7438.2014.034.02.0143

AN EXPERIMENTAL COMPARISON OF BIASED AND UNBIASED RANDOM-KEY GENETIC ALGORITHMS*

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

Received June 3, 2014 / Accepted June 14, 2014

ABSTRACT. Random key genetic algorithms are heuristic methods for solving combinatorial optimization problems. They represent solutions as vectors of randomly generated real numbers, the so-called random keys. A deterministic algorithm, called a decoder, takes as input a vector of random keys and associates with it a feasible solution of the combinatorial optimization problem for which an objective value or fitness can be computed. We compare three types of random-key genetic algorithms: the unbiased algorithm of Bean (1994); the biased algorithm of Gonçalves and Resende (2010); 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. Experiments are run to construct runtime distributions for 36 heuristic/instance pairs. For all pairs of heuristics, we compute probabilities that one heuristic is faster than the other on all 12 instances. 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 (1994) for combinatorial optimization problems for which solutions can be represented as a permutation vector, e.g. sequencing and quadratic assignment. In a RKGA, chromosomes are represented as vectors of randomly generated real numbers in the interval [0, 1). 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

*Invited paper

**Corresponding author
[1]Universidade do Porto, Rua Dr. Roberto Frias, s/n, 4200-464 Porto, Portugal. E-mail: jfgoncal@fep.up.pt
[2]AT&T Labs Research, 200 South Laurel Avenue, Room A5-1F34, Middletown, NJ 07748 USA.
E-mail: mgcr@research.att.com
[3]Rutgers University, 110 Frelinghuysin Road, Piscataway, NJ 08854-8019 USA. E-mail: rtoso@cs.rutgers.edu

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 population of random-key vectors over a number of iterations, called *generations*. The initial population is made up of p real n-vectors of random keys. Each component of the 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 (see Fig. 1): 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 individual of the population of generation k are copied without modification to the population of generation $k + 1$ (see Fig. 2). 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 in the same way as an element of the initial population. At each generation, a small number (p_m) of mutants is introduced into the population (see Fig. 2). 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 (see Fig. 3).

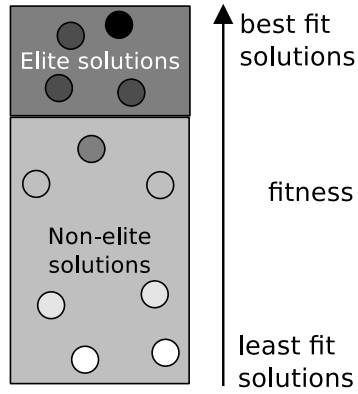

Figure 1 – Population of p solutions is partitioned into a smaller set of p_e elite (most fit) solutions and a larger set of $p - p_e$ non-elite (least fit) solutions.

Bean (1994) 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 (Gonçalves & Resende, 2011), differs from a RKGA 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 (this is *parent A*) in the current population and one from the non-elite

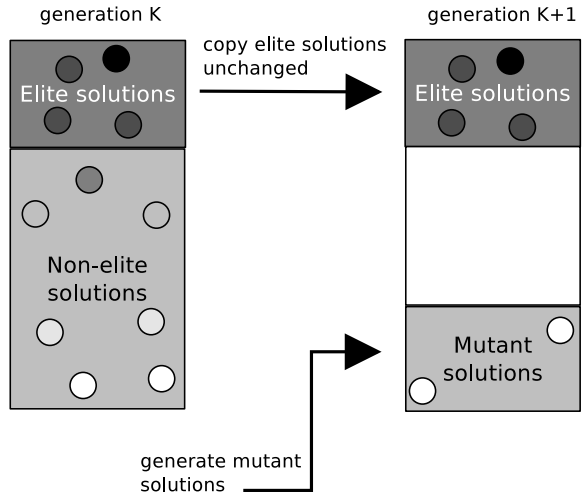

Figure 2 – All p_e elite solutions from population k are copied unchanged to population $k+1$ and p_m mutant solutions are generated in population $k+1$ as random-key vectors.

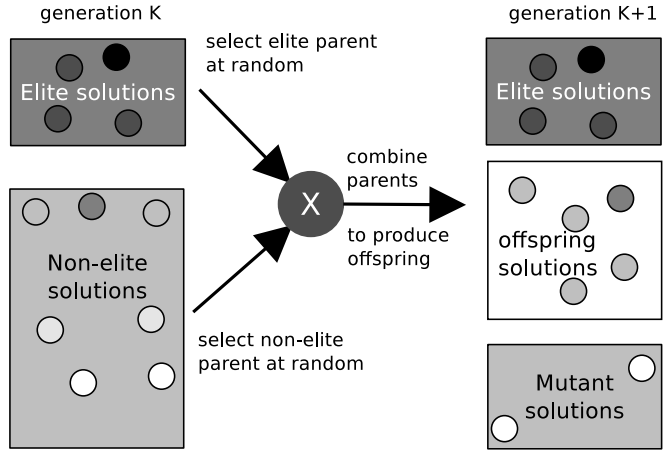

Figure 3 – To complete population $k+1$, $p - p_e - p_m$ offspring are created by combining a parent selected at random from the elite set of population k with a parent selected at random from the non-elite set of population k. Parents can be selected for mating more than once per generation.

partition (*parent B*). We say the selection is *biased* since one parent is always an elite individual. 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* (Spears & DeJong, 1991) 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. 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 $1 - \rho_A$. In this paper, we also consider 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 is called parent A while the other is parent B.

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 partitioned into elite and non-elite individuals to start a new generation. Figure 4 shows a flow diagram of the BRKGA framework with a clear separation between the problem dependent and problem independent components of the method.

Random-key genetic algorithms search the solution space of the combinatorial optimization problem indirectly by exploring the continuous n-dimensional hypercube, using the decoder to map solutions in the hypercube to solutions in the solution space of the combinatorial optimization problem where the fitness is evaluated.

As aforementioned, the role of mutants is to help the algorithm escape from local 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.

To describe a random-key genetic algorithm for a specific combinatorial optimization problem, one needs only to show how solutions are encoded as vectors of random keys and how these vectors are decoded to feasible solutions of the optimization problem. In the next section, we describe random-key genetic algorithms for the four set covering problems problems considered in this paper.

The paper is organized as follows. In Section 2 we describe the four covering problems and in Section 3 we 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. We make concluding remarks in Section 5.

2 FOUR SET COVERING PROBLEMS

In this section, we define four set covering problems and propose random-key genetic algorithms for each.

2.1 General-cost set covering

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 *set covering problem* is to find a

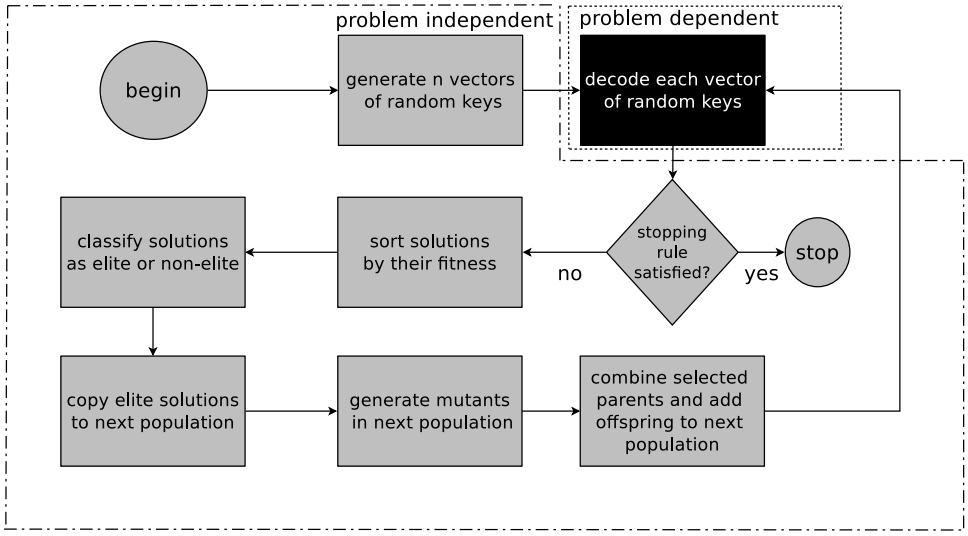

Figure 4 – Flowchart of random-key genetic algorithm with problem independent and problem dependent components.

minimum cost cover. Let A be the binary $m \times n$ matrix such that $A_{i,j} = 1$ if and only if $i \in P_j$. An integer programming formulation for set covering is

$$\min \{cx \ : \ Ax \geq e_m, \ x \in \{0,1\}^n\},$$

where e_m denotes a vector of m ones and x is a binary n-vector such that $x_j = 1$ if and only if $j \in J^*$. The set covering problem has many applications (Vemuganti, 1998) and is NP-hard (Garey & Johnson, 1979).

2.2 Unit-cost set covering

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.

2.3 General-cost set k-covering

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 $\{P_1, \ldots, P_n\}$. As before, let A be the binary $m \times n$ matrix such that $A_{i,j} = 1$ if and only if $i \in P_j$. An integer programming formulation for set k-covering is

$$\min \{cx \ : \ Ax \geq k_m, \ x \in \{0,1\}^n\},$$

where k_m denotes an m-vector of all ks. For example, $k_4 = (k, k, k, k)^\top$.

Note that the set covering problem (Subsection 2.1) as well as the unit-cost set covering problem (Subsection 2.2) are special cases of set k-covering. In both, $k = 1$ and, furthermore, in unit-cost set covering $c_j = 1$ for all $j \in J$.

2.4 Covering by pairs

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.

For all $j \in J$, let x_j be a binary variable such that $x_j = 1$ if and only if $j \in J^*$. For every pair $\{j, k\} \in J \times J$ with $j < k$, let the continuous variable y_{jk} be such that $y_{jk} \leq x_j$ and $y_{jk} \leq x_k$. Therefore, if $y_{jk} > 0$ then $x_j = x_k = 1$. Let $\pi^{-1}(i)$ denote the set of pairs $\{j, k\} \in J \times J$ that cover $i \in I$. An integer programming formulation for the covering by pairs problem is

$$\begin{aligned}
\min \quad & \sum_{j \in J} c_j x_j \\
\text{subject to} \quad & \sum_{\{j,k\} \in \pi^{-1}(i)} y_{jk} \geq 1, && \forall i \in I, \\
& y_{jk} \leq x_j, && \forall \{j, k\} \in J \times J, \ (j < k), \\
& y_{jk} \leq x_k, && \forall \{j, k\} \in J \times J, \ (j < k), \\
& x_j = \{0, 1\}, && \forall j \in J, \\
& 0 \leq y_{jk} \leq 1, && \forall \{j, k\} \in J \times J, \ (j < k).
\end{aligned}$$

3 RANDOM-KEY GENETIC ALGORITHMS FOR COVERING

Biased random-key genetic algorithms for set covering have been proposed by Resende et al. (2012), Breslau et al. (2011), and Pessoa et al. (2011). These include a BRKGA for the Steiner triple covering problem, a unit-cost set covering problem (Resende et al., 2012), BRKGA heuristics for the set covering and set k-covering problems (Pessoa et al., 2011), and a BRKGA for set covering by pairs (Breslau et al., 2011). 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 $|J|$-vector $\mathcal{X}$ of random keys. The j-th key $\mathcal{X}_j$ corresponds to the j-th element of set J.

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:

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

3.1 Greedy algorithms

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.

3.1.1 Greedy algorithm for set k-covering

A greedy algorithm for set covering (Johnson, 1974) 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.

3.1.2 Greedy algorithm for unit-cost covering by pairs

A greedy algorithm for unit-cost covering by pairs is proposed in Breslau et al. (2011). 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 $J^* \cup \{f\}$ 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^*.

3.2 Local improvement procedures

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.

3.2.1 Greedy uncover

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.

3.2.2 1-Opt

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 two variants of random-key genetic algorithms (the biased variant – BRKGA – and Bean's unbiased variant – RKGA) of Section 1 as well as a variant of Bean's algorithm which we shall refer to as RKGA*. Like Bean's RKGA, RKGA* selects both parents at random from the entire population. This is unlike a BRKGA, where one parent is always selected from the elite partition of the population and the other from the non-elite partition. Unlike Bean's algorithm, a RKGA* assigns the best fit of both parents as *parent A* and the other as *parent B* for crossover. Ties are broken by rank in the sorted population. RKGA assigns the selected parents to the roles of *parent A* and *parent B* at random.

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

In the experiments, we consider four problem types: general-cost set covering, Steiner triple (unit-cost) covering, general-cost set k-covering, and unit-cost covering by pairs. For general-cost set covering we consider instances `scp41`, `scp51`, and `scpa1` of Beasley (1990). The Steiner triple covering instances used are `stn135`, `stn243`, and `stn405` (Resende et al., 2012). For general-cost set k-covering we consider instances `scp41-2`, `scp45-11`, and

scp48-7 of Pessoa et al. (2013). For unit-cost covering by pairs, we consider instances n558-i0-m558-b140, n220-i0-m220-b220, and n190-i9-m190-b95 of Breslau et al. (2011).

The experiment consists in running the three variants (BRKGA, RKGA, and RKGA*) 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 (Aiex et al., 2007) 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. (2012). This way, the three variants can be compared on each instance.

Table 1 lists the instances, their dimensions, the values of the target solutions used in the experiments, and the best solution known to date.

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

All algorithms were implemented in C++ using the BRKGA Application Programming Interface (API) of Toso & Resende (2014). 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. 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 due to coding.

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

The goal of the experiment is to derive runtime distributions for the three heuristics on a set of 12 instances from the four problem classes. Runtime distributions, or time-to-target plots (Aiex et al., 2007), 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 runtime distributions. To do this, we multiple 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 5 –16 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 $(i - 0.5)/100$, 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

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

Table 3 – Average CPU time per 100 generations for each problem and each algorithm. For each instance, the table list the 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

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 $(i - 0.5)/100$, for $i = 1, \ldots, 100$.

Iteration count distributions are a useful graphical tool to compare algorithms on a given instance. Consider, for example, the plots in Figure 5 which shows iteration count distributions for BRKGA, RKGA*, and RKGA for general-cost set covering instance scp41 using the optimal solution of 429 as the target solution. The plots in this figure clearly show an ordering of the heuristics with respect to the number of iterations needed to find an optimal solution. For example the probabilities that BRKGA, RKGA*, and RKGA will find n optimal solution in at most 2000 iterations are, respectively, 83.5%, 58.5%, and 49.5%. Similarly, with a probability of 60.5% the heuristics BRKGA, RKGA*, and RKGA will find an optimal solution in at most 1076, 2139, and 2715 iterations.

However, we are often more interested in the distribution of the random variable *time-to-target solution* than in iterations to target solution, so if the heuristic that took the largest number of iterations to reach the optimal had the fastest running time per iteration, it could be that its runtime distribution would be better than the distribution of the heuristic that took the fewest iterations. For problem instance scp41, this is not a problem since the average running times per 100 iterations for BRKGA, RKGA*, and RKGA are, respectively, 22.02s, 22.67s, and 24.71s. As can be seen in Table 3, this relative ordering is maintained for all instances with the exception of stn135 where RKGA is slightly faster per iteration than RKGA*.

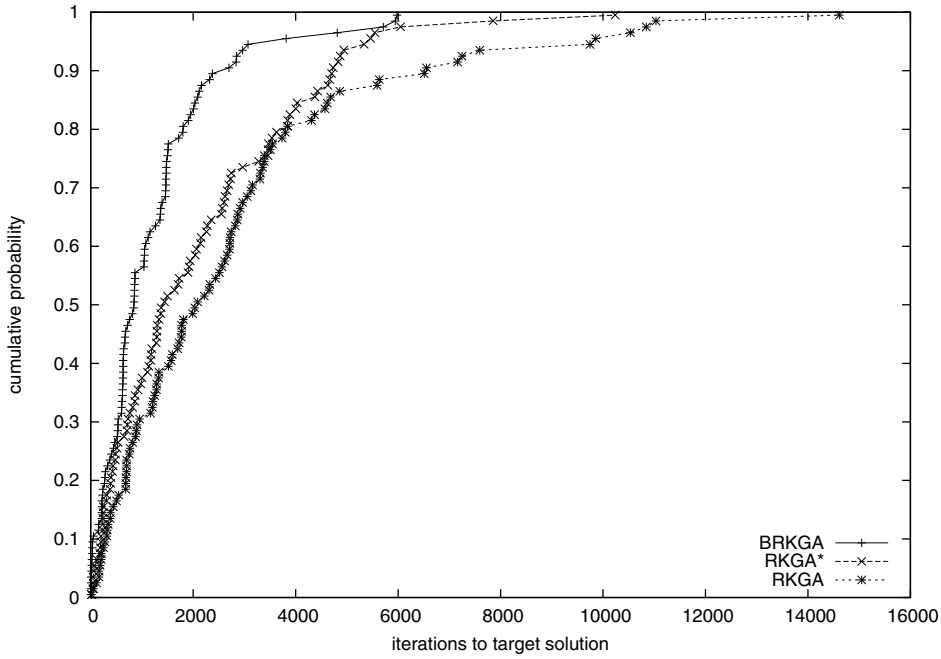

Figure 5 – Iteration count distributions for BRKGA, RKGA*, and RKGA on instance `scp41` with target solution 429.

To produce runtime distributions we would ideally run the experiments on the same machine. Since for these experiments this was not practical, we estimate runtime distributions on a machine by computing the average time per iteration of the heuristic on the instance on the machine and then multiply the entries of the iteration count distributions by the corresponding average time per iteration.

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. (2012). 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 algorithm iterates until the error is less than 0.001. A perl language script of their method is described in Ribeiro & Rosseti (2013). Applying their program 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. These values are shown in Table 4.

We make the following remarks about the experiment.

- The experiment consisted in generating runtime distributions for three types of heuristics: biased (Gonçalves & Resende, 2011) random-key genetic algorithm – BRKGA, random-

Table 4 – Probability that time-to-target solution of BRKGA (t_{BRKGA}) will be less than that of RKGA (t_{RKGA}) and RKGA* (t_{RKGA^*}) and probability that 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 three heuristics using the iterative method of Ribeiro et al. (2012).

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

key genetic algorithm (Bean, 1994) – RKGA, and a slightly modified variant of RKGA – RKGA*, on four classes of combinatorial optimization problems: general-cost set covering, Steiner triple covering, general-cost set k-covering, and unit-cost covering by pairs. For each of the four problem classes, runtime distributions were generated for each of the three heuristics on three problem instances.

- Since the number of runs required to carried out the experiment was large (3600) and many runs were lengthy, we distributed them on four multi-core linux computers and generated iteration count distributions for all 36 heuristic/instance pairs. Each iteration count distribution was produced running the particular heuristic on the instance 100 times, each using a different initial seed as input for the random number generator.

- 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 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.

- To compute average running time per iteration, we ran all 36 heuristic/ instance pairs 10 times for 100 iterations each on a Intel Core i7-2760QM CPU at 2.40GHz using eight threads. Average 100-iteration user times (sum of times over all threads) were measured for each pair and these values were each divided by 100 to compute average time per itera-

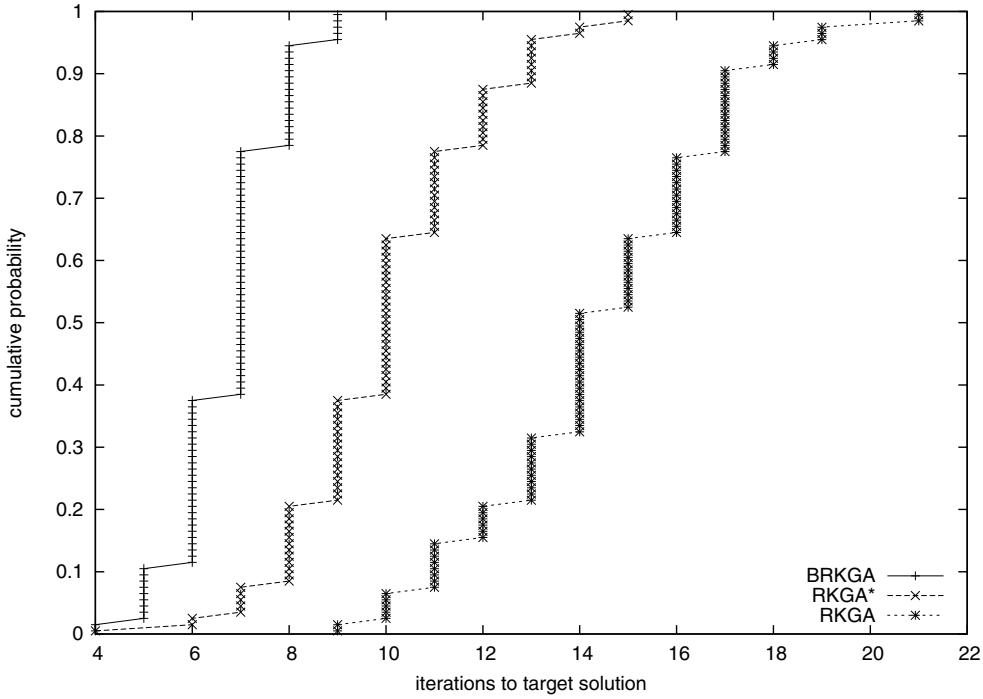

Figure 6 – Iteration count distributions for BRKGA, RKGA*, and RKGA on instance `scp51` with target solution 253.

tion. 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 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 (Resende & Ribeiro, 2010). 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. (2012) were such that the resulting heuristic was more random than BRKGAs used to solve

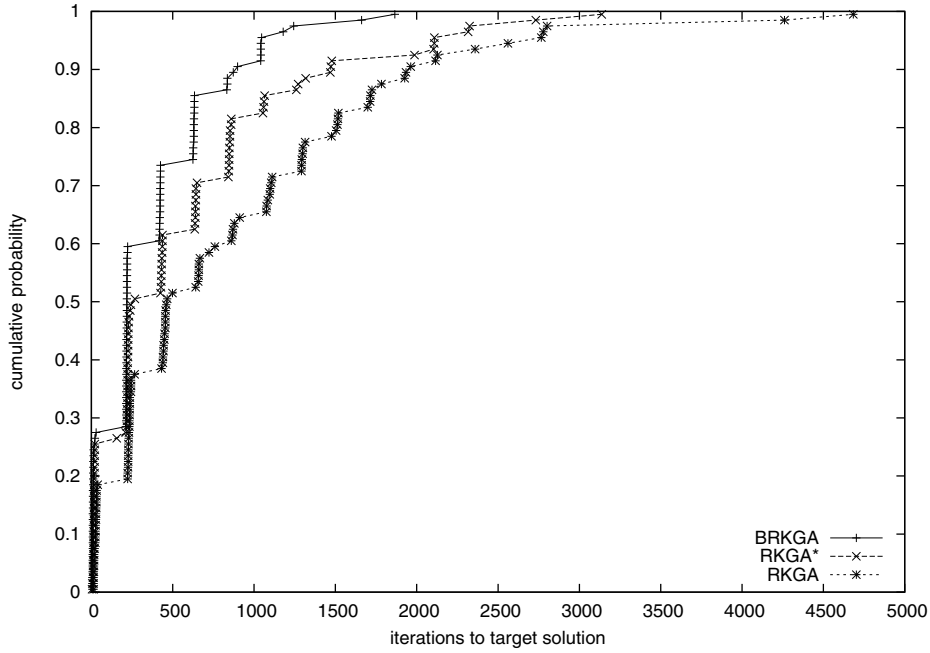

Figure 7 – Iteration count distributions for BRKGA, RKGA*, and RKGA on instance `scpa1` with target solution 253.

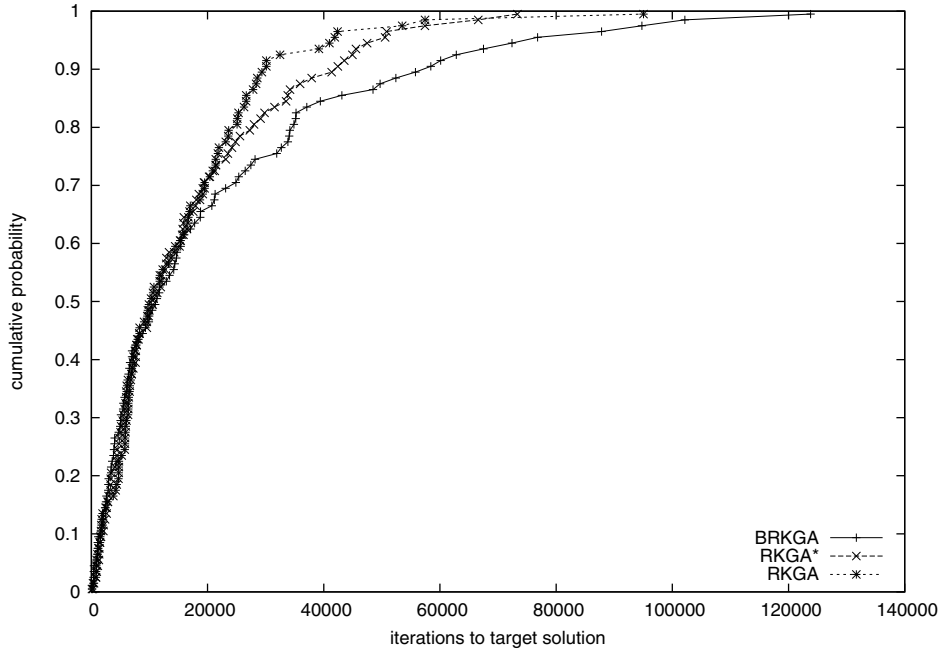

Figure 8 – Iteration count distributions for BRKGA, RKGA*, and RKGA on instance `stn135` with target solution 103.

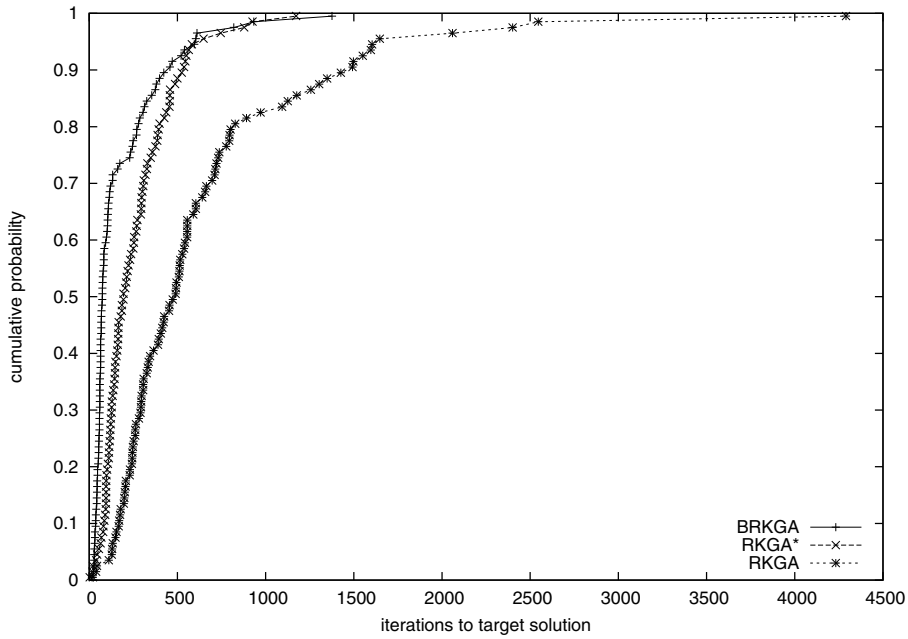

Figure 9 – Iteration count distributions for BRKGA, RKGA*, and RKGA on instance `stn243` with target solution 198.

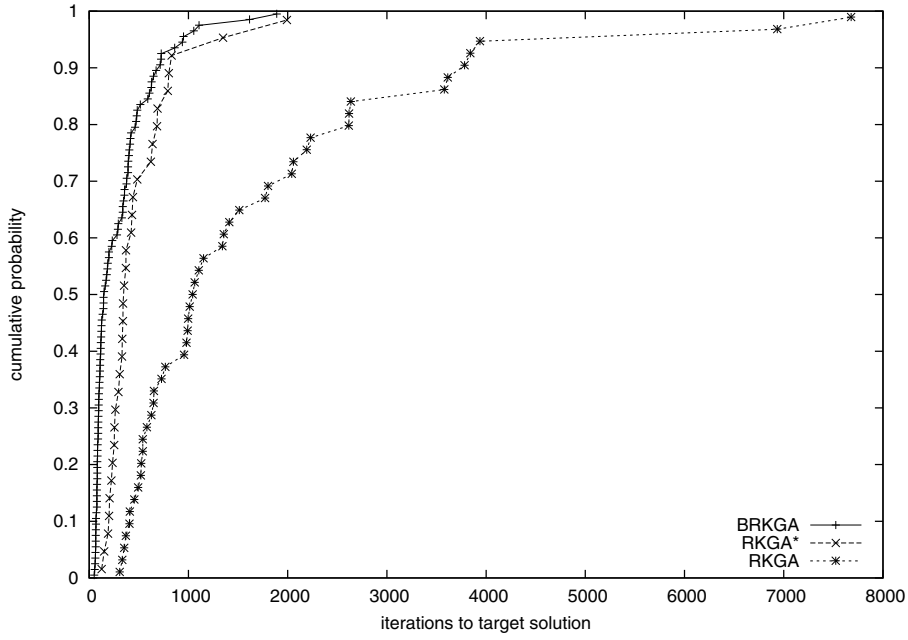

Figure 10 – Iteration count distributions for BRKGA, RKGA*, and RKGA on instance `stn405` with target solution 339.

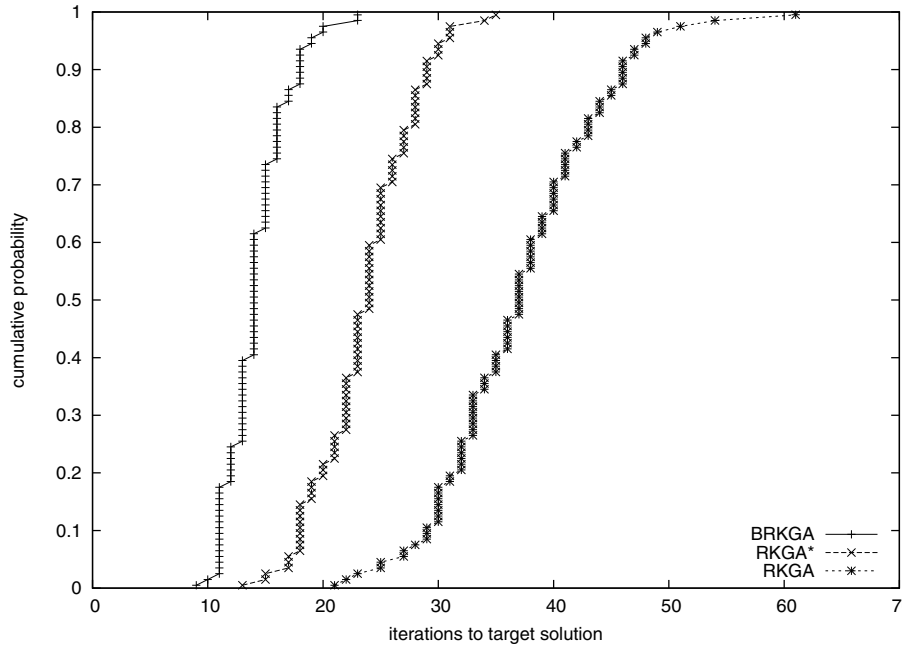

Figure 11 – Iteration count distributions for BRKGA, RKGA*, and RKGA on instance `scp41-2` with target solution 1148.

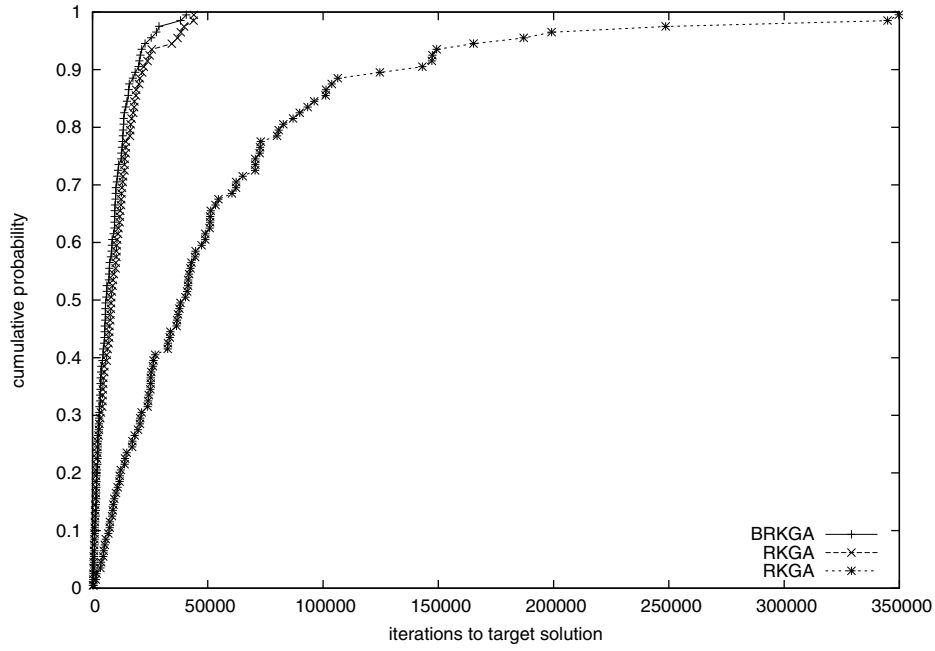

Figure 12 – Iteration count distributions for BRKGA, RKGA*, and RKGA on instance `scp45-11` with target solution 18856.

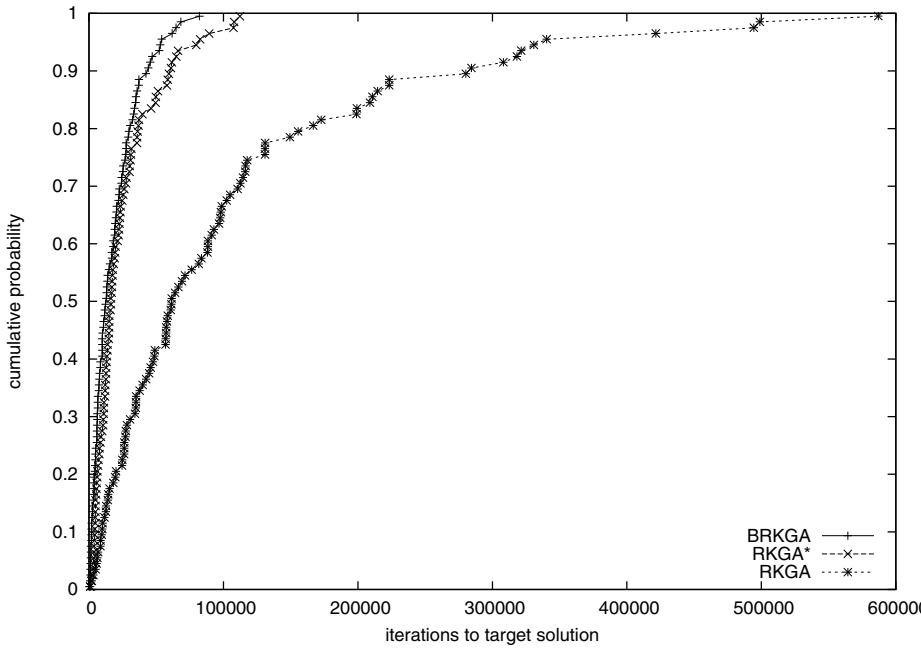

Figure 13 – Iteration count distributions for BRKGA, RKGA*, and RKGA on instance `scp48-7` with target solution 8421.

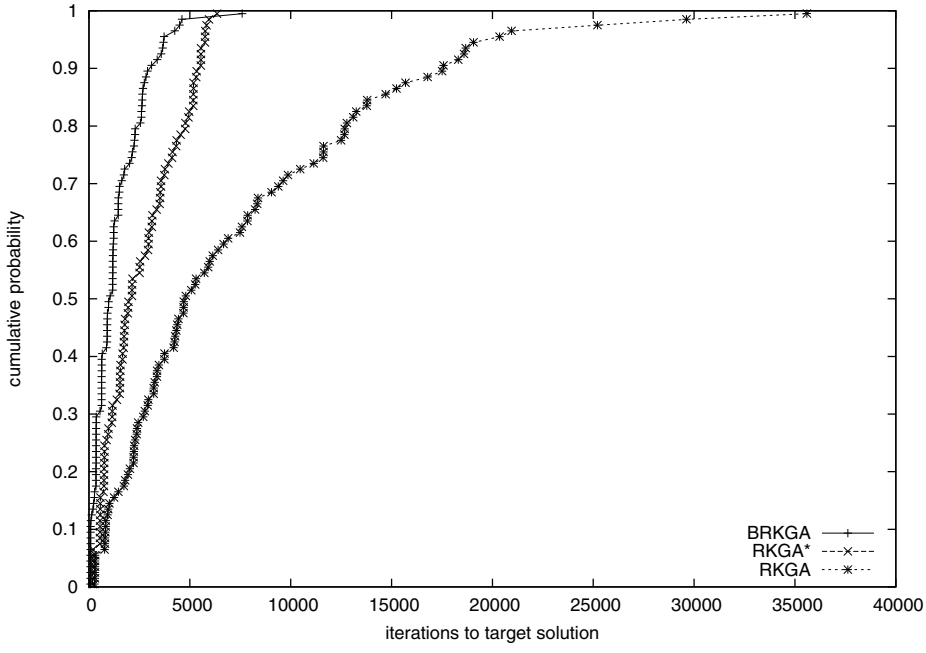

Figure 14 – Iteration count distributions for BRKGA, RKGA*, and RKGA on instance `n558-i0-m558-b140` with target solution 55.

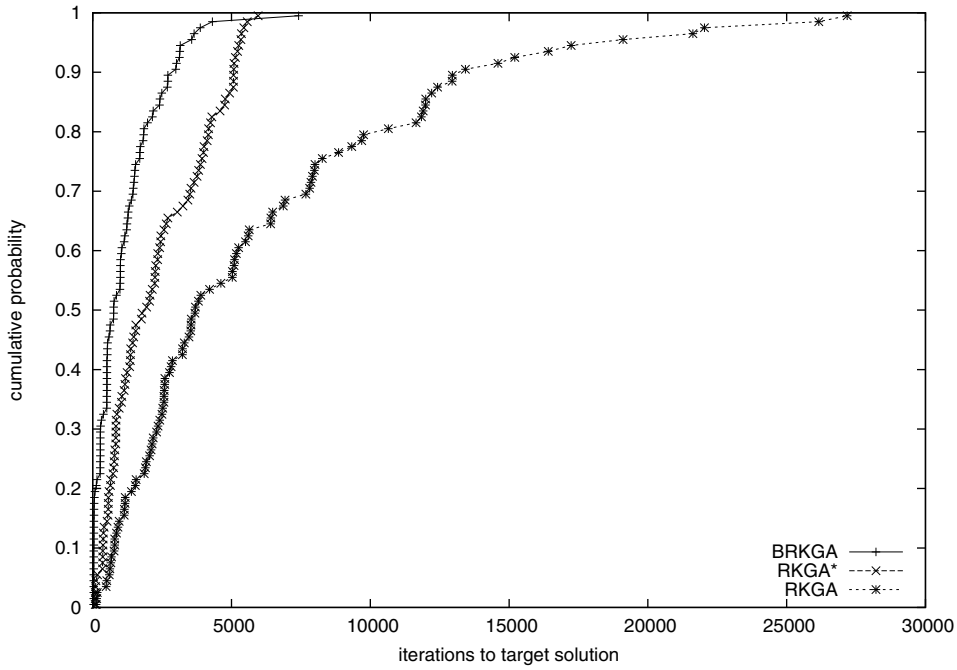

Figure 15 – Iteration count distributions for BRKGA, RKGA*, and RKGA on instance `n220-i0-m220-b220` with target solution 62.

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.

- Visual inspection of Figures 5 to 16 shows a clear domination of BRKGA over RKGA, with the exception of instance `stn135`. This is backed up by the probabilities shown in Table 4, where $0.733 \leq \Pr(t_{BRKGA} \leq t_{RKGA}) \leq 0.999$ for all but instance `stn135` where $\Pr(t_{BRKGA} \leq t_{RKGA}) = 0.485$.

- Visual inspection of Figures 5 to 16 shows a clear domination of BRKGA over RKGA*, with the exception of instances `stn135` and `scp45-11`. This is backed up by the probabilities shown in Table 4, where $0.591 \leq \Pr(t_{BRKGA} \leq t_{RKGA*}) \leq 0.975$ for all but instances instances `stn135` and `scp45-11`, where $\Pr(t_{BRKGA} \leq t_{RKGA*}) = 0.496$ and 0.547, respectively.

- Visual inspection of Figures 5 to 16 shows a clear domination of RKGA* over RKGA, with the exception of instance `scp41`. This is backed up by the probabilities shown in Table 4, where $0.588 \leq \Pr(t_{BRKGA} \leq t_{RKGA}) \leq 0.975$ for all but instances `scp41` and `stn135`, where $\Pr(t_{BRKGA} \leq t_{RKGA}) = 0.489$.

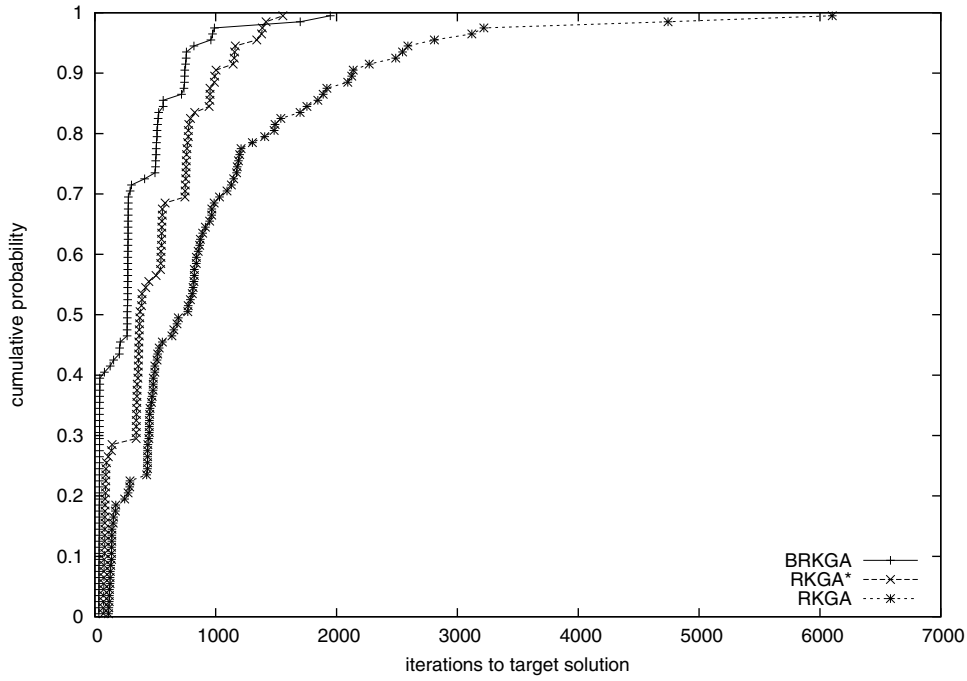

Figure 16 – Iteration count distributions for BRKGA, RKGA*, and RKGA on instance `n190-i9-m190-b95` with target solution 37.

5 CONCLUDING REMARKS

This paper studies runtime distributions for three types of random-key genetic algorithms: the algorithm of Bean (1994) – RKGA, a slight modification of Bean's algorithm in which the best fit of the two parents selected for mating has higher probability of passing along its keys to its offspring – RKGA*, and the biased random-key genetic algorithm of Gonçalves & Resende (2011) – BRKGA. We use the methodology described in Aiex et al. (2007) to generate plots of iteration count distributions for BRKGA, RKGA*, and RKGA on problem instances from four classes of set covering problems: general-cost covering, Steiner triple covering, general cost set k-covering, and unit-cost covering by pairs. In each class, the experiment considers three instances. On each, the heuristics are run 100 times, independently, and stop when a given target solution value (in most cases the best known solution value) is found.

To obtain runtime distributions from the iteration count distributions we multiply iteration counts for each algorithm/instance pair by the corresponding average time per iteration obtained by running all algorithm/instance pairs 10 times for 100 iterations on the same computer.

With the sets of runtimes from each distribution on hand, we use the iterative method of Ribeiro et al. (2012) to compute precise probabilities (having error at most 0.001) that one method is faster than another.

We conclude that Bean's algorithm (RKGA) can be improved by simply making the best fit of the two parents chosen for mating have a higher probability of passing along its keys to it offspring than the other parent. The resulting algorithm is denoted RKGA*. For 11 of the 12 instances $\Pr(t_{RKGA*} \leq t_{RKGA}) \geq 0.588$, where $t_{\mathcal{A}}$ is the random variable time-to-target solution of heuristic $\mathcal{A}$. In 9 of the 12 instances $\Pr(t_{RKGA*} \leq t_{RKGA}) \geq 0.7$. In 4 of the 12 instances $\Pr(t_{RKGA*} \leq t_{RKGA}) \geq 0.854$. In 2 of the 12 instances $\Pr(t_{RKGA*} \leq t_{RKGA}) \geq 0.943$. The biased random-key genetic algorithm (BRKGA) does even better. In 11 of the 12 instances $\Pr(t_{BRKGA} \leq t_{RKGA}) \geq 0.733$. In 9 of the 12 instances $\Pr(t_{BRKGA} \leq t_{RKGA}) \geq 0.841$. In 3 of the 12 instances $\Pr(t_{BRKGA} \leq t_{RKGA}) \geq 0.917$. Despite the fact that BRKGA has to order the population by fitness at each iteration, in addition to the same work done by RKGA and RKGA*, it is still faster per iteration, on average, than both RKGA and RKGA*. This is mainly due to the fact that it is not as random as RKGA and RKGA*. Combining time per iteration with number of iterations shows that BRKGA is faster than RKGA*, too. In 11 of the 12 instances $\Pr(t_{BRKGA} \leq t_{RKGA*}) \geq 0.547$. In 9 of the 12 instances $\Pr(t_{BRKGA} \leq t_{RKGA*}) \geq 0.643$. In 7 of the 12 instances $\Pr(t_{BRKGA} \leq t_{RKGA*}) \geq 0.721$. In 2 of the 12 instances $\Pr(t_{BRKGA} \leq t_{RKGA*}) \geq 0.96$.

Randomness is not always bad. On the Steiner triple covering instances, where the fitness landscapes are flat, it pays off to be more random. In fact, for the Steiner triple covering class, the parameter setting of BRKGA was more random than usual, with $p_m = \lceil .55p \rceil$ instead of the usual $\lceil .15p \rceil$ and $\rho_A = 0.65$ instead of the usual 0.7. BRKGA was faster than both RKGA and RKGA* on both `stn243` and `stn405`. On `stn135`, however, both RKGA and RKGA* were slightly faster than BRKGA, with $\Pr(t_{RKGA*} \leq t_{BRKGA}) = 0.504$ and $\Pr(t_{RKGA} \leq t_{BRKGA}) = 0.515$.

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 (Gonçalves & Resende, 2011).

ACKNOWLEDGMENT

This research has been 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.

REFERENCES

[1] AIEX RM, RESENDE MGC & RIBEIRO CC. 2007. TTTPLOTS: A perl program to create time-to-target plots. *Optimization Letters*, **1**: 355–366.

[2] BEAN JC. 1994. Genetic algorithms and random keys for sequencing and optimization. *ORSA J. on Computing*, **6**: 154–160.

[3] BEASLEY JE. 1990. OR-Library: Distributing test problems by electronic mail. *Journal of the Operational Research Society*, **41**: 1069–1072.

[4] BRESLAU L, DIAKONIKOLAS I, DUFFIELD N, GU Y, HAJIAGHAYI M, JOHNSON DS, RESENDE MGC & SEN S. 2011. Disjoint-path facility location: Theory and practice. In: *ALENEX 2011: Workshop on algorithm engineering and experiments*, January.

[5] GAREY MR & JOHNSON DS. 1979. Computers and intractability. A guide to the theory of NP-completeness. W.H. Freeman and Company, San Francisco, California.

[6] GONÇALVES JF & RESENDE MGC. 2011. Biased random-key genetic algorithms for combinatorial optimization. *J. of Heuristics*, **17**: 487–525.

[7] JOHNSON DS. 1974. Approximation algorithms for combinatorial problems. *Journal of Computer and System Sciences*, **9**: 256–278.

[8] PESSOA LS, RESENDE MGC & TOSO RF. 2011. Biased random-key genetic algorithms for set *k*-covering. Technical report, AT&T Labs Research, Florham Park, New Jersey.

[9] PESSOA LS, RESENDE MGC & RIBEIRO CC. 2013. A hybrid Lagrangean heuristic with GRASP and path relinking for set *k*-covering. *Computers and Operations Research*, **40**: 3132–3146.

[10] RESENDE MGC & RIBEIRO CC. 2010. Greedy randomized adaptive search procedures: Advances and applications. In: Gendreau and J.-Y. Potvin, editors, Handbook of Metaheuristics, pages 281–317. Springer, 2nd edition.

[11] RESENDE MGC, TOSO RF, GONÇALVES JF & SILVA RMA. 2012. A biased random-key genetic algorithm for the Steiner triple covering problem. *Optimization Letters*, **6**: 605–619.

[12] RIBEIRO CC & ROSSETI I. 2013. `tttplots`-compare: A perl program to compare time-to-target plots or general runtime distributions of randomized algorithms. Technical report, Department of Computer Science, Universidade Federal Fluminense, Rua Passo da Pátria 156, Niterói, RJ 24210-240, Brazil.

[13] RIBEIRO CC, ROSSETI I & VALLEJOS R. 2012. Exploiting run time distributions to compare sequential and parallel stochastic local search algorithms. *J. of Global Optimization*, **54**: 405–429.

[14] SPEARS WM & DEJONG KA. 1991. On the virtues of parameterized uniform crossover. In: *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 230–236.

[15] TOSO RF & RESENDE MGC. 2014. A C++ application programming interface for biased random-key genetic algorithms. *Optimization Methods and Software*. doi: 10.1080/10556788.2014.890197.

[16] VEMUGANTI RR. 1998. Applications of set covering, set packing and set partitioning models: A survey. In: D-Z. DU & P.M. PARDALOS, editors, *Handbook of Combinatorial Optimization*, **1**: 573–746. Kluwer Academic Publishers.