A BIASED RANDOM-KEY GENETIC ALGORITHM FOR THE STEINER TRIPLE COVERING PROBLEM

M.G.C. RESENDE, R.F. TOSO, J.F. GONÇALVES, AND R.M.A. SILVA

Abstract. We present a biased random-key genetic algorithm (BRKGA) for finding small covers of computationally difficult set covering problems that arise in computing the 1-width of incidence matrices of Steiner triple systems. Using a parallel implementation of the BRKGA, we compute improved covers for the two largest instances in a standard set of test problems used to evaluate solution procedures for this problem. The new covers for instances A_{405} and A_{729} have sizes 335 and 617, respectively. On all other smaller instances our algorithm consistently produces covers of optimal size.

1. Introduction

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 $J = \{1, \ldots, n\}$. A subset $J^* \subseteq J$ is called a *cover* if $\cup_{j \in J^*} P_j = I$. The *set covering problem* is to find a cover of minimum cardinality. 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

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

where e_k denotes a vector of k ones and x is a binary n-vector such that $x_j = 1$ if and only if $j \in J^*$. The set covering problem is NP-hard (Garey and Johnson, 1979).

Fulkerson et al. (1974) introduced a class of computationally difficult set covering problems that arise in computing the 1-width of incidence matrices of Steiner triple systems. In this paper, we will refer to this problem as the *Steiner triple covering* problem. The A matrix in the integer programming formulation of this problem has exactly three ones per row. Furthermore, for every pair of columns j and k there is exactly one row i for which $A_{i,j} = A_{i,k} = 1$. Steiner triple system A is said to contain triple $\{i, j, k\}$ if there exists a row q such that $A_{q,i} = A_{q,j} = A_{q,k} = 1$. Let A_3 be the 1×3 matrix of all ones. A recursive procedure can generate Steiner triple systems for which $n = 3^k$ or $n = 15 \cdot 3^{k-1}$, $k = 1, 2, \ldots$ (Hall, 1967; Fulkerson et al., 1974; Avis, 1980). Starting from A_3 this recursive procedure generates Steiner triple systems A_9, A_{27}, A_{81}, A_{243}, A_{729}, ... Fulkerson et al. (1974) introduced A_{15} and A_{45}, two Steiner triple systems for which $n \neq 3^k$. From A_{15}, the recursive procedure generates Steiner triple systems A_{45}, A_{135}, A_{405}, ...

Fulkerson et al. (1974) solved A_9, A_{15}, and A_{27} to optimality, but were unable to solve A_{45}. Avis (1980) reported that A_{45} was finally solved optimally in 1979 by

Date: October 20, 2010.

Key words and phrases. Steiner triple covering, set covering, genetic algorithm, biased random-key genetic algorithm, random keys, combinatorial optimization, heuristics, metaheuristics.

AT&T Labs Research Technical Report.

TABLE 1. Steiner triple covering instances. For each instance, the table lists its name (instance), cardinality of J (n), cardinality of I (m), best known solution value (BKS), indication if best solution value is optimal (opt), and reference where instance was solved optimally if optimal or where best known solution was found if optimality is unknown (reference).

instance	n	m	BKS	opt	reference
stn9	9	12	5	yes	Fulkerson et al. (1974)
stn15	15	35	9	yes	Fulkerson et al. (1974)
stn27	27	117	18	yes	Fulkerson et al. (1974)
stn45	45	330	30	yes	Ratliff (1979)
stn81	81	1080	61	yes	Mannino and Sassano (1995)
stn135	135	3015	103	yes	Ostrowski et al. (2009; 2010)
stn243	243	9801	198	yes	Ostrowski et al. (2009; 2010)
stn405	405	27270	335	?	This paper.
stn729	729	88452	617	?	This paper.

H. Ratliff. A_{81} was solved to optimality by Mannino and Sassano (1995). Recently, A_{135} and A_{243} were both solved to optimality by Ostrowski et al. (2009; 2010). The solution of A_{135} required slightly over 10 million CPU seconds (126 days) while A_{243} was solved in just over 51 hours.

Several heuristic approaches for the Steiner triple covering problem have been proposed. Feo and Resende (1989) proposed a GRASP heuristic with which they found a cover of size 61 for A_{81}. This cover was later shown to be optimal by Mannino and Sassano (1995). Karmarkar et al. (1991) proposed an interior point method with which they found a cover of size 105 for A_{135}. In the same paper, they used a GRASP to produce a better cover of size 104. Mannino and Sassano (1995) also found a cover of this size. In 1998, Odijk and van Maaren (1998) produced a cover of size 103. This value was recently shown to be optimal by Ostrowski et al. (2009; 2010). The GRASP of Feo and Resende (1989) as well as the interior point method of Karmarkar et al. (1991) produced covers of size 204 for A_{243}. Karmarkar et al. (1991) used the GRASP of Feo and Resende (1989) to improve the best known solution of A_{243} at that time to 203. Mannino and Sassano (1995) improved it further to 202. Odijk and van Maaren (1998) produced a cover of size 198, recently shown to be optimal by Ostrowski et al. (2009; 2010). A_{405} and A_{729} were recently introduced by Ostrowski et al. (2009; 2010). To date, no computational results have been reported for A_{405}. Ostrowski et al. (2010) report that the best solution produced for A_{729} by CPLEX (v9) after two weeks was 653. Using their enumerate-and-fix scheme as a heuristic, Ostrowski et al. (2010) were able to find a better cover, of size 619. The test problem instances as well as the sizes of the best covers found over the years are summarized in Table 1.

In this paper, we propose a new effective heuristic for the Steiner triple covering problem. The heuristic is based on the algorithmic framework of biased random-key genetic algorithms (BRKGA). We show that our heuristic finds the best known solutions for all instances A_k, for $k = 9, 15, 27, 45, 81, 135, 243, 405, 729$. For A_{405}

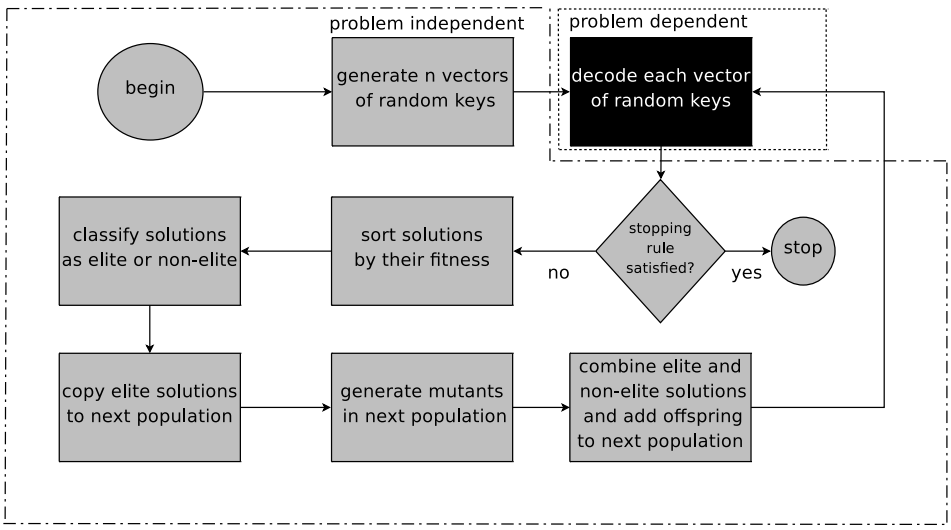

FIGURE 1. Flowchart of biased random-key genetic algorithm with problem independent and problem dependent components.

we produce covers of size 335, while for A_{729} we found an improved cover of size 617.

The paper is organized as follows. In Section 2 we describe biased random-key genetic algorithms. This is followed by a description of solution encoding and random-key vector decoding in Section 3. In Section 4 we present some implementation details of our multi-population parallel genetic algorithm. Computational results are given in Section 5 and concluding remarks are made in Section 6.

2. Biased random-key genetic algorithms

Genetic algorithms with random keys, or *random-key genetic algorithms* (RKGA), were first introduced by Bean (1994) for solving combinatorial optimization problems involving sequencing. 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 solution of the combinatorial optimization problem for which an objective value or fitness can be computed.

A RKGA evolves a population of random-key vectors over a number of iterations, called *generations*. The initial population is made up of p vectors of random-keys. Each component of the solution vector is generated independently 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 individual of the population of generation k are copied without modification to the population of generation $k + 1$. RKGAs implement mutation by introducing *mutants* into the population. A mutant is simply a vector of random keys generated in the same way that an element of the initial population is generated. At each generation, a small number

(p_m) of mutants is 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.

Bean (1994) selects two parents at random from the entire population to implement mating in a RKGA. A *biased random-key genetic algorithm,* or BRKGA (Gonçalves and Resende, 2010), 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 in the current population and one from the non-elite partition. 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 and DeJong, 1991) is used to implement mating in BRKGAs. Let $\rho_e > 0.5$ be the probability that an offspring inherits the vector component of its elite parent. 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 $e(i)$ of the elite parent e with probability ρ_e and the value of the i-th component $\bar{e}(i)$ of the non-elite parent $\bar{e}$ with probability $1 - \rho_e$.

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 1 shows a flow diagram of the BRKGA framework with a clear separation between the problem dependent and problem independent components of the method.

A BRKGA searches the solution space of the combinatorial optimization problem indirectly by searching 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.

To describe a BRKGA 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 a BRKGA for Steiner triple covering.

3. A BRKGA heuristic for Steiner triple covering

In this section we show how solutions are encoded to a vector of random keys and how solutions are decoded from a vector of random keys.

3.1. **Encoding a solution to a vector of random keys.** A solution is encoded as a vector $\mathcal{X} = (\mathcal{X}_1, \ldots, \mathcal{X}_n)$ of size $n = |J|$, where $\mathcal{X}_j$ is a random number in the interval $[0, 1)$, for $j = 1, \ldots, n$. The j-th component of $\mathcal{X}$ corresponds to the j-th element of J (i.e. the j-th column of matrix A).

3.2. **Decoding a solution from a vector of random keys.** A decoder takes as input the vector of random keys $\mathcal{X}$ and returns a cover $J^* \subseteq J$ corresponding to the indices of the columns of A selected to cover the rows of A. To describe the decoding procedure, let the cover be represented by a binary vector $\mathcal{Y} = (\mathcal{Y}_1, \ldots, \mathcal{Y}_n)$ of size $n = |J|$, where $\mathcal{Y}_j = 1$ if and only of $j \in J^*$.

Our decoder has three phases. In the first phase, the values of $\mathcal{Y}$ are initially set according to

$$\mathcal{Y}_j = \begin{cases} 1 & \text{if } \mathcal{X}_j \geq 0.5 \\ 0 & \text{otherwise.} \end{cases}$$

The indices implied by the binary vector $\mathcal{Y}$ can correspond to either a feasible or infeasible cover J^*. If J^* is a feasible cover, then the second phase is skipped. If J^* is not a valid cover, then the second phase of the decoding procedure builds a valid cover with a greedy algorithm for set covering (Johnson, 1974), starting from the partial cover J^* defined by $\mathcal{Y}$. This greedy algorithm proceeds as follows. While J^* is not a valid cover, select the smallest index $j \in J \setminus J^*$ for which the inclusion of j in J^* covers a maximum number of yet-uncovered elements of I. The third phase of the decoder attempts to remove superfluous elements from cover J^*. While there is some element $j \in J^*$ such that $J^* \setminus \{j\}$ is still a valid cover, then such element having the smallest index is removed from J^*.

4. Implementation details

The algorithm described in this paper was implemented using the *BRKGA framework*, a C++ framework for biased random-key genetic algorithms (Resende and Toso, 2010). We observed in Section 2 that a BRKGA consist of a problem-dependent phase (the decoder) and a general-purpose problem independent phase (see Figure 1). We designed the BRKGA framework as an object-oriented, multi-threaded, general-purpose framework which implements all problem independent components and provides a simple hook for chromosome decoding. This enables seamless interaction with any problem-specific decoder.

To the user, the BRKGA framework consists of a template class (Stroustrup, 1997) `BRKGA< class Decoder, class RNG >` that is initialized with the hyper-parameters n, p, p_e, p_m, ρ_e, together with the desired number of threads to perform the decoding in parallel. Moreover, the framework evolves π populations simultaneously, exchanging the best t chromosomes from each population at every q generations, whenever there is no repeated chromosomes in the target populations. Classes `Decoder` and `RNG` are also required to perform decoding and random number generation tasks, respectively, each having to implement the following methods:

- `double Decoder::decode(vector< double >& chromosome)` returns the fitness (`double`) of the vector of random keys $\mathcal{X}$, implemented as a vector of doubles of size n.
- `double RNG::rand()` returns a double precision random deviate in range $[0.0, 1.0)$.
- `unsigned long int RNG::randInt()` returns an unsigned long integer in the range $[0, \texttt{std} :: \texttt{numeric\_limits} < \texttt{unsigned long} >:: \texttt{max}()]$, where the last command indicates the maximum `unsigned long` supported by the architecture.

The BRKGA framework does decoding in parallel. It uses OpenMP (OpenMP, 2010), a multi-platform application programming interface (API) for shared-memory parallel programming in C, C++, and FORTRAN. In the computational experiments, we make use of a parallel version of the biased random-key genetic algorithm.

In our implementation, 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

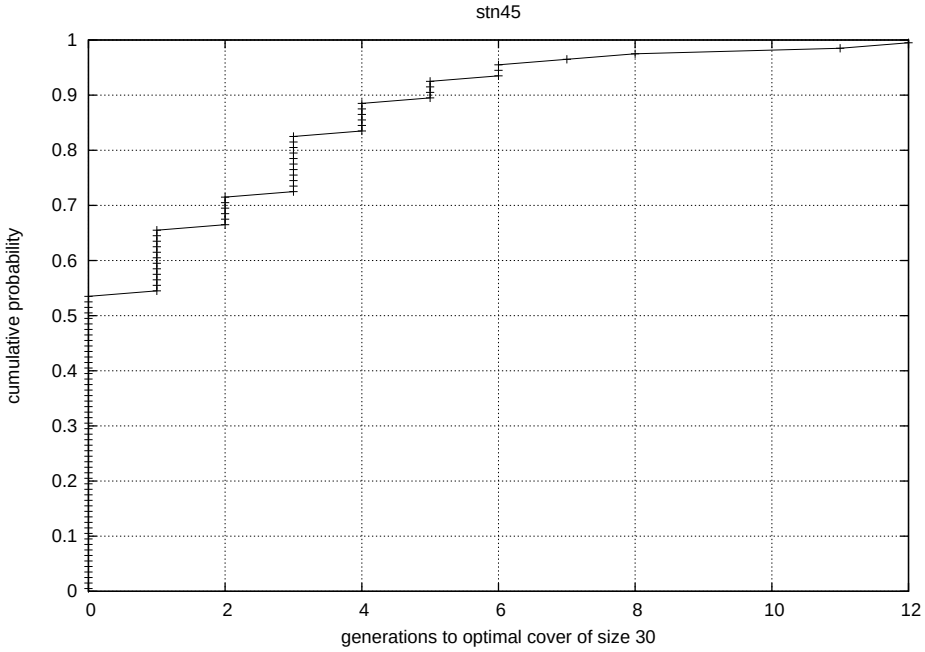

FIGURE 2. Distribution of generations to optimal cover for *stn45*. 100 independent runs were done, each stopping when a cover with optimal value (30) was found.

application of only the first phase of the decoder. To do this we reset $\mathcal{X}$ as follows:

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

5. Experimental results

The objective of this computational experiment is threefold. We first investigate the effectiveness of our BRKGA to find the optimal solutions for the test problems in the literature with known optima. Secondly, for the two problems for which optimal solutions are unknown, we seek to produce smaller covers than those previously produced. Finally, we investigate the effectiveness of our parallel implementation.

The standard set of test problems consists of instances *stn9*, *stn15*, *stn27*, *stn45*, *stn81*, *stn135*, *stn243*, *stn405*, and *stn729*. The instances can be downloaded from `http://www.research.att.com/~mgcr/data/steiner-triple-covering.tar.gz`. Table 1 shows some characteristics of these instances.

The experiments were conducted on a server with four 2.4 GHz Quad-core Intel Xeon E7330 processors with 128 Gb of memory running CentOS 5 Linux. The code was compiled with the `g++` compiler (version 4.1.2 20080704) with flags `-O3 -fopenmp`. Random numbers were generated with the *Mersenne Twister* (Matsumoto and Nishimura, 1998).

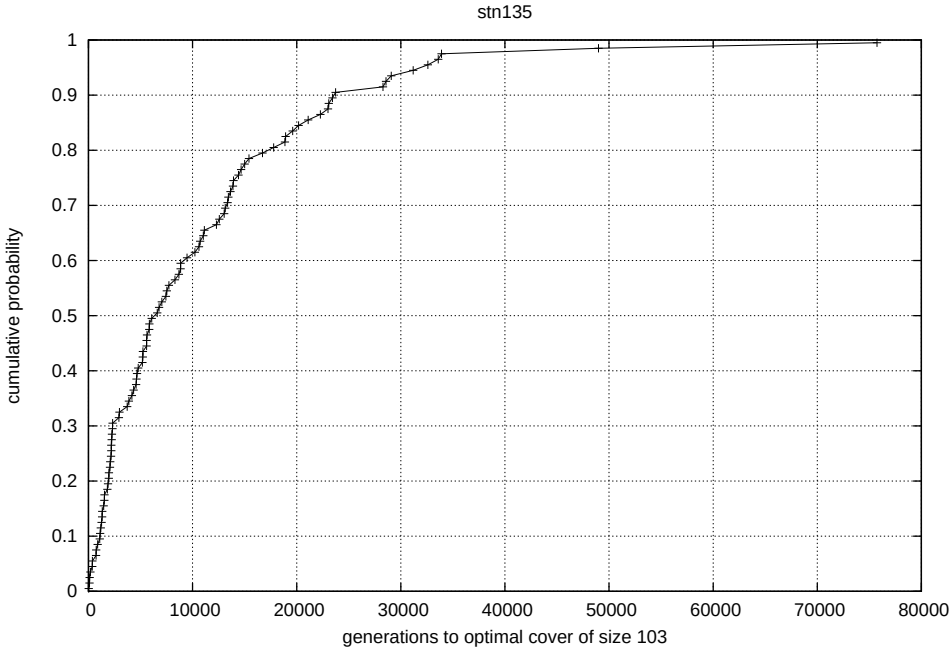

FIGURE 3. Distribution of generations to optimal cover for *stn135*. 100 independent runs were done, each stopping when a cover with optimal value (103) was found.

We used the following parameters in the BRKGA. The population size was set to $p = 10n$ of which $p_e = \lceil 1.5n \rceil$ solutions are considered elite and $p - p_e = \lfloor 8.5n \rfloor$ are non-elite. At each iteration, $p_m = \lfloor 5.5n \rfloor$ mutants are generated. The probability that a child inherits the random key of its elite parent is $\rho_e = 60\%$. The multi-population scheme evolves $\pi = 3$ populations. Once every $q = 100$ generations, each population potentially exchanges its $t = 2$ best solutions with the other populations. These parameter settings are somewhat different from those used in the literature (Gonçalves and Resende, 2010) because of the flat nature of the solution space landscape in this problem. To explore the neighborhood, more randomness had to be added to this BRKGA than what is customary (common settings are $p_e \approx 0.15p$, $p_m \approx 0.15p$, and $\rho_e \approx 70\%$).

We use different stopping criteria in the different experiments, which we describe below.

In the rest of this section, we first describe the experiments on instances *stn9*, *stn15*, *stn27*, *stn45*, *stn81*, *stn135*, and *stn243*, for which optimal cover values are known. Then we consider instances *stn405* and *stn729*, both of which have unknown optimal cover values. Finally, we report on some experiments with the parallel implementation using up to 16 processors.

5.1. **Experiments on instances with known optimal covers.** For the instances with known optimal solution, we ran the genetic algorithm independently 100 times, using a different random number generator seed for each run. Each run was done in parallel using 16 processors and stopped when an optimal cover was

TABLE 2. Several covers of size 335 were found for *stn405*. The indices of the $405 - 335 = 70$ zeroes of three of these solutions are listed here.

Solution 1

1	2	3	4	5	31	32
33	34	35	56	57	58	59
60	86	87	88	89	90	91
92	93	94	95	106	107	108
109	110	146	147	148	149	150
171	172	173	174	175	201	202
203	204	205	221	222	223	224
225	226	227	228	229	230	266
267	268	269	270	271	272	273
274	275	306	307	308	309	310

Solution 2

6	7	8	9	10	26	27
28	29	30	41	42	43	44
45	51	52	53	54	55	71
72	73	74	75	86	87	88
89	90	151	152	153	154	155
196	197	198	199	200	226	227
228	229	230	261	262	263	264
265	286	287	288	289	290	331
332	333	334	335	361	362	363
364	365	396	397	398	399	400

Solution 3

6	7	8	9	10	21	22
23	24	25	31	32	33	34
35	46	47	48	49	50	76
77	78	79	80	96	97	98
99	100	136	137	138	139	140
151	152	153	154	155	176	177
178	179	180	196	197	198	199
200	251	252	253	254	255	266
267	268	269	270	341	342	343
344	345	371	372	373	374	375

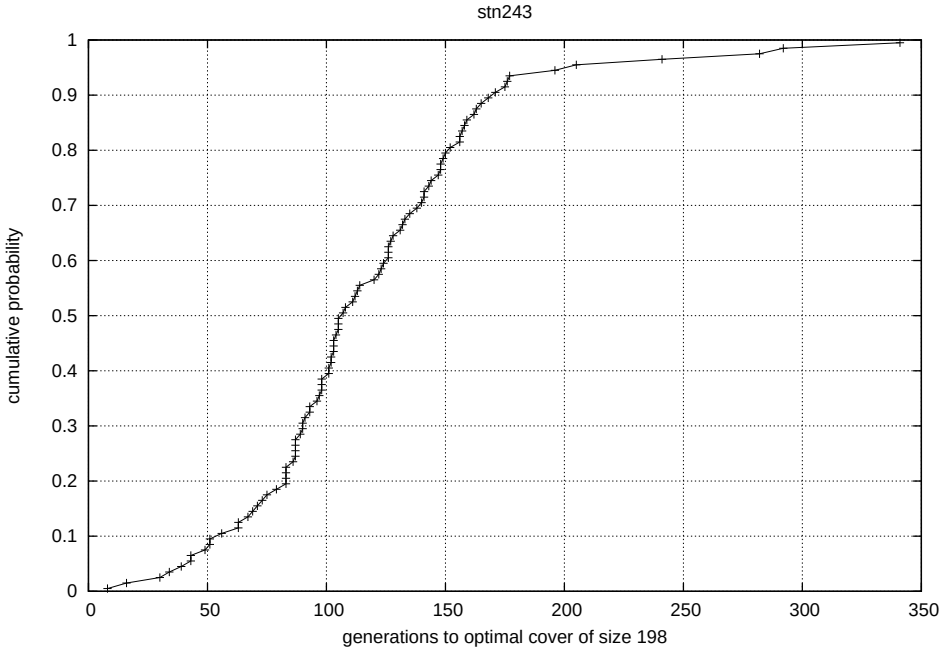

FIGURE 4. Distribution of generations to optimal cover for *stn243*. 100 independent runs were done, each stopping when a cover with optimal value (of 198) was found.

TABLE 3. A cover of size 617 was found for *stn729*. The indices of the $729 - 617 = 112$ zeroes of this solution are listed here.

3	5	11	12	27	36	39	43
52	54	56	63	70	73	74	85
94	121	128	142	159	166	167	176
177	181	197	200	201	214	215	220
225	230	237	239	245	252	255	263
264	277	279	283	288	291	299	309
313	322	323	331	333	334	343	344
355	357	364	365	377	382	390	392
400	405	410	430	437	446	470	483
497	509	520	535	548	550	560	561
565	567	570	578	580	590	591	599
600	608	614	621	627	629	632	639
648	652	661	663	669	673	680	682
693	697	699	705	709	712	717	723

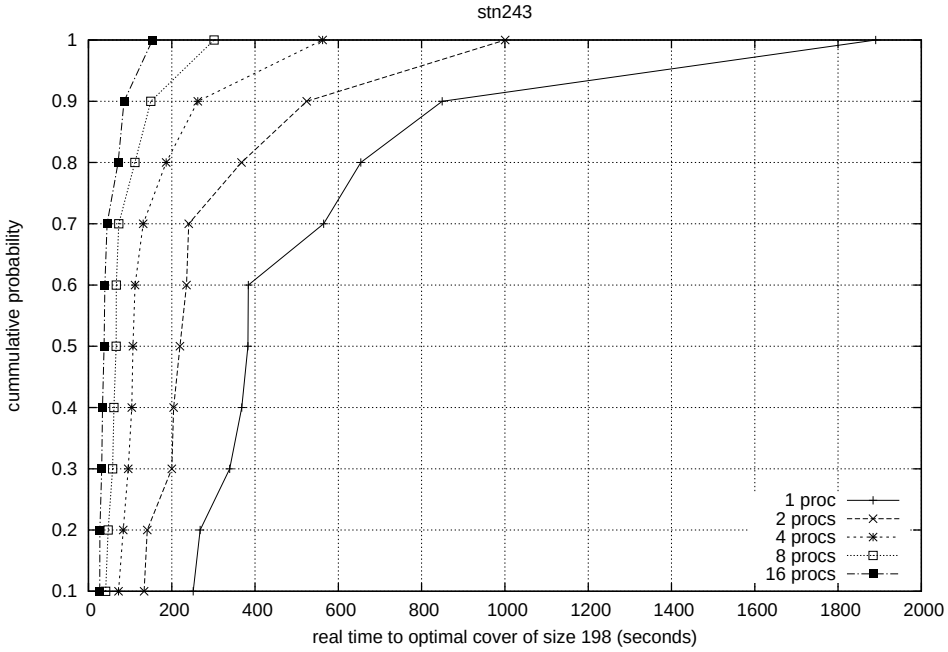

FIGURE 5. Runtime distributions to optimal cover of size 198 on *stn243* for 1, 2, 4, 8, and 16 processor runs. 10 independent runs were done for each processor configuration, each stopping when a cover with optimal value (of 198) was found.

found. On all 100 runs for each instance, the algorithm found an optimal cover. On the smallest instances (*stn9*, *stn15*, and *stn27*) the optimal cover was always found in the initial population. On *stn81*, the optimal cover was also found in the initial population in 99 of the 100 runs. In the remaining run on *stn81*, the optimal cover was found in the second generation. Runtime distributions are shown for instances *stn45*, *stn135*, and *stn243* in Figures 2, 3, and 4, respectively.

As can be observed in Figure 2, an optimal cover for *stn45* for was found in the initial population on 54 of the 100 runs. The largest number of iterations over the 100 runs was 12. The time per 1000 generations on instance *stn45* was 4.70s (real), 70.55s (user), and 2.73s (system).

Figure 3 shows that *stn135* is the most difficult of the instances with known optimal covers. Though 9 of the 100 runs required less than 1000 generations, 39 of the 100 runs required over 10,000 generations to produce an optimal cover. No run required fewer than 23 generations. The largest number of iterations over the 100 runs was 75,741. The time per 1000 generations on instance *stn135* was 19.91s (real), 316.70s (user), and 0.85s (system).

Though larger than *stn135*, instance *stn243* appears to be much easier to solve. Figure 4 shows that 39 of the 100 runs took fewer than 100 generations to produce an optimal cover and 95% of the runs took fewer than 200 generations. The largest number of iterations over the 100 runs was 341. The time per 1000 generations on instance *stn243* was 68.60s (real), 1095.19s (user), and 0.79s (system).

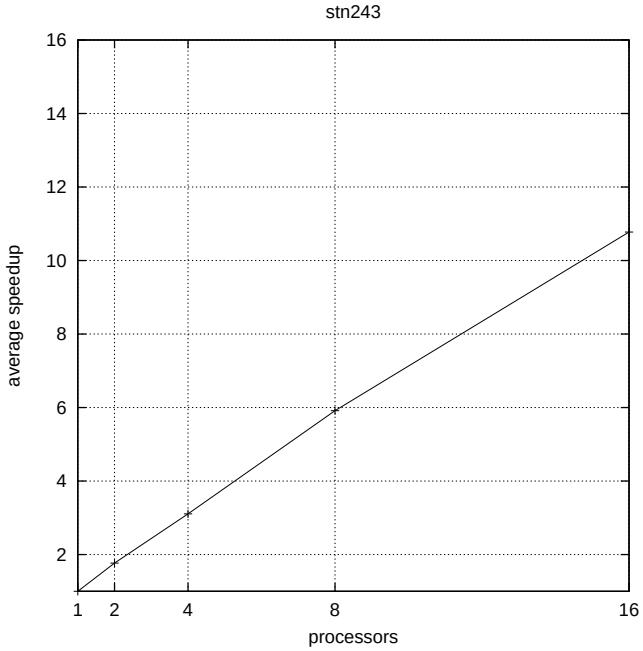

FIGURE 6. Average speedup with 1, 2, 4, 8, and 16 processors to find optimal cover of size 198 on *stn243*. 10 independent runs were done for each processor configuration, each stopping when a cover with optimal value (of 198) was found.

To show that the success of the BRKGA to consistently find covers of size 198 was not due to the decoder alone, we conducted 100 independent runs simulating a random multi-start method. Each run consisted of 1000 generations of the BRKGA with three populations of size 1000, each with an elite set of one, and a mutant set of size 999. This way, at each iteration, 2997 random solutions are generated, and each is evaluated with the three-phase decoder used in the genetic algorithm runs. The elite sets of one keep track of the incumbent solution in each population. Since the elite together with the mutants make up the entire population, mating never takes place (differentiating these runs from those of the BRKGA). The 100 runs generated a total of about 300 million solutions. The random multi-start heuristic was far from the optimal value of 198, finding covers of size 202 in 9 of the 100 runs and of size 203 in the remaining 91.

5.2. **Computing covers for the two largest instances.** For *stn405* and *stn729*, the two largest instances in the test set, we ran the BRKGA and stopped after 5000 generations without improvement. For both instances, we found improved solutions.

For *stn405* three covers of size 335 were found. The first run produced a cover after 203 generations. The second run, after 5165 gerations. The third run found the cover after 2074 generations. The complements of the three covers are shown in Table 2. The time per 1000 generations on instance *stn405* was 796.82s (real), 12723.40s (user), and 11.67s (system).

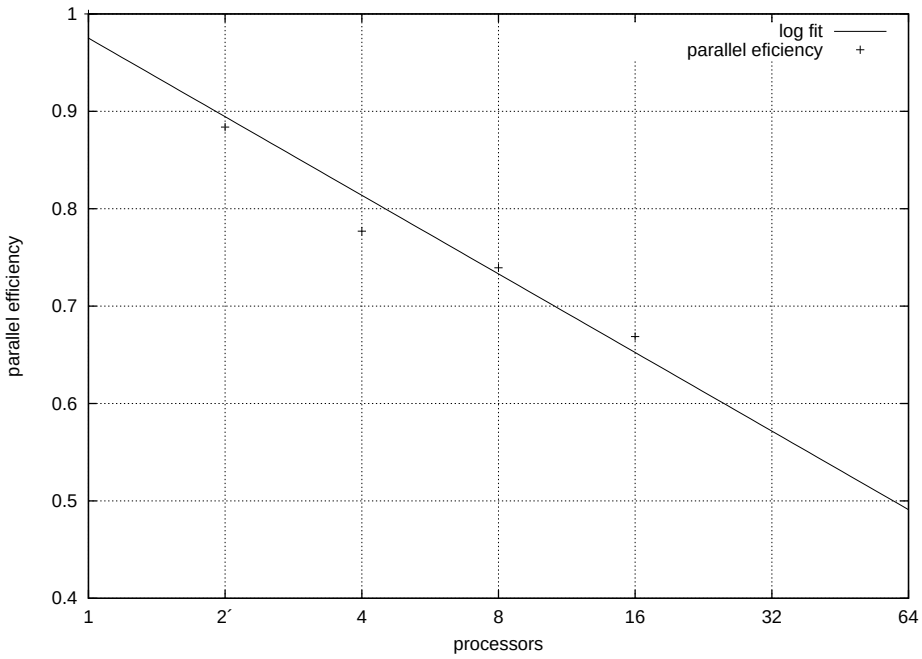

FIGURE 7. Parallel efficiency ($\frac{t_1}{p \cdot t_p}$), where p is the number of processors and t_p is average real (wall clock) time for the p processor runs) with 1, 2, 4, 8, and 16 processors to find optimal cover of size 198 on *stn243*. 10 independent runs were done for each processor configuration, each stopping when a cover with optimal value (of 198) was found. A log fit $0.975 - 0.1164 \log(p)$ is also plotted.

On the largest instance, *stn729*, a cover of size 617 were found in 1601 generations. The complement of this cover is shown in Table 3. The time per 1000 generations on instance *stn729* was 6099.40s (real), 93946.68s (user), and 498.00s (system).

5.3. **Computing covers with a parallel implementation.** The BRKGA proposed in this paper does decoding in parallel. Though decoding is the major computational bottleneck of this algorithm, there are several other tasks that are not done in parallel. These include: generation of random-key vectors for initial population and mutants at each iteration of the algorithm with corresponding calls to the random number generator; crossover operations at each iteration to produce offspring random-key vectors; periodic exchange of elite solutions among multiple populations; ordering of populations by fitness values; and copying elite solutions to next population at each iteration of the algorithm. Consequently, one cannot expect 100% efficiency (linear speedup) in a parallel implementation of this BRKGA. Nevertheless, we show that significant speedup is observed.

To illustrate the effectiveness of our parallel *OpenMP* implementation of the BRKGA, we carried out the following experiment on instance *stn243*. On each of five processor configurations (single processor, two, four, eight, and 16 processors), we carried out ten independent runs, stopping when the algorithm produced a

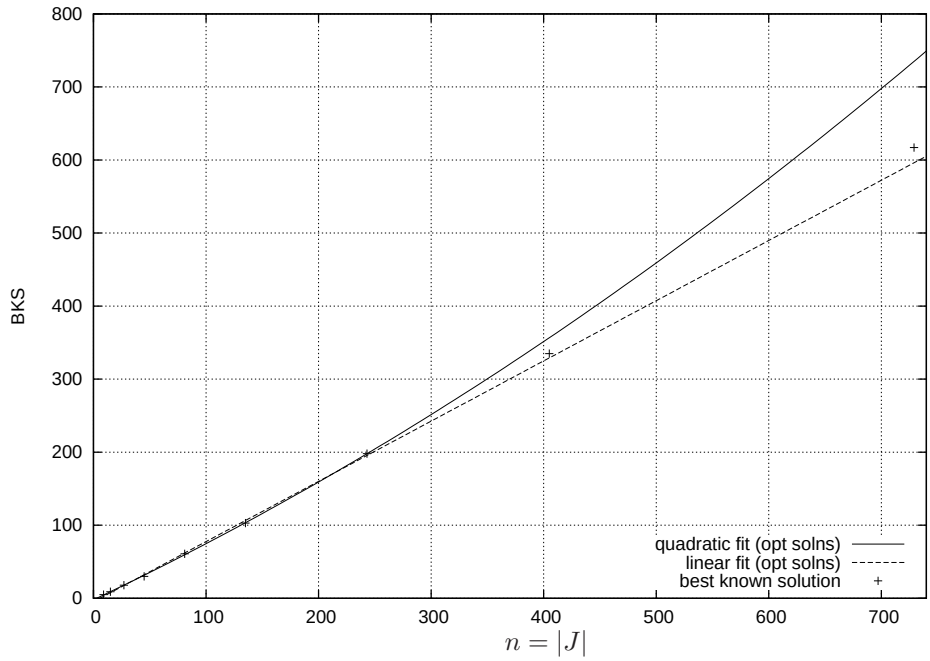

FIGURE 8. Value of best known cover as a function of $n = |J|$. A linear and a quadratic fit were computed using the optimal cover values and are plotted together with all best solution values.

cover of size 198. Figure 5 shows runtime distributions (in seconds) for the five configurations. Average speedup is shown in Figure 6. As can be observed in the figure, with 16 processors the speedup is still above tenfold. Figure 7 shows parallel efficiency, i.e. $t_1/(p \cdot t_p)$, where p is the number of processors and t_p is average real (wall clock) time for runs with p processors. A log fit is also plotted indicating that if parallel efficiency continues to decline at this rate, we still expect a speedup of about 32-fold when we use 64 processors.

6. Concluding remarks

This paper introduced a biased random-key genetic algorithm for the Steiner triple covering problem, a computationally difficult set covering problem. Our parallel multi-population biased random-key genetic algorithm not only found optimal covers for all instances with known optimal solution, but also found new improved covers (best known solutions) of size 335 and 617, respectively, for the two largest instances, *stn405* and *stn729*, in the standard test set. Figure 6 shows how the best known solution increases with n, the size of the problem.

The parallel implementation achieved a speedup of 10.8 on 16 processors and is expected to achieve a speedup of about 32 on 64 processors.

The BRKGA described in this paper can be easily extended to solve other types of set covering problems.

Acknowledgement

José F. Gonçalves was partially supported by Fundação para a Ciência e Tecnologia (FCT) project PTDC/GES/72244/2006. Ricardo M.A Silva was partially supported by the Brazilian National Council for Scientific and Technological Development (CNPq) and the Foundation for Support of Research of the State of Minas Gerais, Brazil (FAPEMIG).

References

D. Avis. A note on some computationally difficult set covering problems. *Mathematical Programming*, 18:138–145, 1980.

J.C. Bean. Genetic algorithms and random keys for sequencing and optimization. *ORSA J. on Computing*, 6:154–160, 1994.

T.A. Feo and M.G.C. Resende. A probabilistic heuristic for a computationally difficult set covering problem. *Operations Research Letters*, 8:67–71, 1989.

D.R. Fulkerson, G.L. Nemhauser, and L.E. Trotter, Jr. Two computationally difficult set covering problems that arise in computing the 1-width of incidence matrices of Steiner triple systems. *Mathematical Programming Study*, 2:72–81, 1974.

M.R. Garey and D.S. Johnson. *Computers and intractability. A guide to the theory of NP-completeness*. WH Freeman and Company, San Francisco, Calif, 1979.

J.F. Gonçalves and M.G.C. Resende. Biased random-key genetic algorithms for combinatorial optimization. *J. of Heuristics*, 2010. DOI: `http://dx.doi.org/10.1007/s10732-010-9143-1`.

M. Hall. *Combinatorial theory*. Blaisdell Company, 1967.

D.S. Johnson. Approximation algorithms for combinatorial problems. *Journal of Computer and System Sciences*, 9:256–278, 1974.

N.K. Karmarkar, M.G.C. Resende, and K.G. Ramakrishnan. An interior point algorithm to solve computationally difficult set covering problems. *Mathematical Programming*, 52:597–618, 1991.

C. Mannino and A. Sassano. Solving hard set covering problems. *Operations Research Letters*, 18:1–5, 1995.

M. Matsumoto and T. Nishimura. Mersenne Twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, 8:3–30, 1998.

M.A. Odijk and H. van Maaren. Improved solutions to the Steiner triple covering problem. *Information Processing Letters*, 65:67–69, 1998.

OpenMP. `http://openmp.org`, 2010. Last visited on October 19, 2010.

J. Ostrowski, J. Linderoth, F. Rossi, and S. Smriglio. Solving large Steiner triple covering problems. Technical Report 1663, Computer Sciences Department, U. of Wisconsin, Madison, 2009.

J. Ostrowski, J. Linderoth, F. Rossi, and S. Smriglio. Solving Steiner triple covering problems. *Optima*, 83, July 2010.

M.G.C. Resende and R.F. Toso. BRKGA framework: A C++ framework for implementing biased random-key genetic algorithms. Technical report, AT&T Labs Research, October 2010.

W.M. Spears and K.A. DeJong. On the virtues of parameterized uniform crossover. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 230–236, 1991.

B. Stroustrup. *The C++ programming language*. Addison-Wesley, 1997.

(Mauricio G.C. Resende) Algorithms and Optimization Research Department, AT&T Labs Research, 180 Park Avenue, Room C241, Florham Park, NJ 07932 USA.
E-mail address, M.G.C. Resende: mgcr@research.att.com

(Rodrigo F. Toso) Department of Computer Science, Rutgers University, Piscataway, NJ 08854 USA.
E-mail address: rtoso@cs.rutgers.edu

(José Fernando Gonçalves) Faculdade de Economia do Porto / NIAAD, Rua Dr. Roberto Frias, 4200-464 Porto, Portugal.
E-mail address: jfgoncal@fep.up.pt

(R.M.A. Silva) Computational Intelligence and Optimization Group, Dept. of Computer Science, Federal University of Lavras, C.P. 3037, CEP 37200-000, Lavras, MG, Brazil.
E-mail address, R.M.A. Silva: rmas@dcc.ufla.br