A C++ APPLICATION PROGRAMMING INTERFACE FOR BIASED RANDOM-KEY GENETIC ALGORITHMS

RODRIGO F. TOSO AND MAURICIO G.C. RESENDE

Abstract. In this paper, we describe `brkgaAPI`, an efficient and easy-to-use object oriented application programming interface for the algorithmic framework of biased random-key genetic algorithms. Our cross-platform library automatically handles the large portion of problem-independent modules that are part of the framework, including population management and evolutionary dynamics, leaving to the user the task of implementing a problem-dependent procedure to convert a vector of random keys into a solution to the underlying optimization problem. Our implementation is written in the C++ programming language and may benefit from shared-memory parallelism when available.

1. Introduction

A biased random-key genetic algorithm (BRKGA) is a general search metaheuristic for finding optimal or near-optimal solutions to hard combinatorial optimization problems (Gonçalves and Resende, 2010). It is derived from the random-key genetic algorithm of Bean (1994), differing in the way solutions are combined to produce offspring. BRKGAs have three key features that specialize genetic algorithms:

- A fixed *chromosome* encoding using a vector of n random keys or *alleles* over the interval $[0, 1)$, where the value of n depends on the instance of the optimization problem;
- A well-defined evolutionary process adopting parameterized uniform crossover (Spears and DeJong, 1991) to generate *offspring* and thus evolve the population;
- The introduction of new chromosomes called *mutants* in place of the mutation operator usually found in genetic algorithms.

Such features simplify and standardize the metaheuristic with a set of self-contained tasks from which only one is problem-dependent: that of *decoding* a chromosome, i.e., using it to construct a solution to the underlying optimization problem, from which the objective function value or *fitness* can be computed. Analogously, the decoder in a BRKGA is used to indirectly map the continuous chromosome space $[0, 1)^n$ onto the space containing valid solutions of the optimization problem. Contrary to genetic algorithms in general, feasibility can be easily maintained from one generation to another thanks to this standardized encoding, since all offspring and mutants introduced in the evolutionary process are also vectors of random keys.

Date: August 16, 2011.
Key words and phrases. Biased random-key genetic algorithms, genetic algorithms, application programming interface, combinatorial optimization, parallel programming.
AT&T Labs Research Technical Report.

Table 1. Parameters necessary to specify a BRKGA.

Parameter	Description
n	Number of alleles per chromosome
p	Number of chromosomes in population
p_e	Size of the elite set in population
p_m	Number of mutants to be introduced in population at each generation
ρ_e	Probability that an allele is inherited from the elite parent

At a glance, BRKGAs start with and then evolve a population containing exactly p chromosomes, each having n random keys, for a number of generations until some stopping criterion is met. In a given generation k, this evolutionary process is guided by the following steps:

(1) The fitness of each chromosome is decoded to reveal the objective value of the underlying optimization problem. A chromosome when associated with its fitness is called an *individual*.
(2) The population is partitioned into two groups: an elite set containing p_e individuals with the best fitness values and a non-elite set with the remaining individuals.
(3) The population at generation $k+1$ will then consist of:
   (a) p_e individuals from elite set of the current generation;
   (b) p_m randomly generated chromosomes called mutants;
   (c) $p_o = p - p_m - p_e$ evolved chromosomes (offspring) that are produced by mating two parents from the current generation, selected at random, with replacement, one from the elite set and another from the non-elite set. To define the j-th allele of offspring i, a uniform random number $r_{ij} \in [0, 1)$ is generated: if $r_{ij} \leq \rho_e$, with $\rho_e > 0.5$, then offspring i inherits the j-th allele of its elite parent; otherwise, it inherits the j-th allele of the non-elite parent.

To summarize the discussion above, the BRKGA framework is illustrated in Figure 1. We highlight the box that performs decoding and observe that the chromosomes can be decoded in parallel, one independently of the other. In Table 1 we list the parameters that are necessary to specify a biased random-key genetic algorithm. Advice regarding such parameter settings can be found in Gonçalves and Resende (2010).

A common ingredient found in many implementations of the BRKGA framework is the adoption of multiple populations that are evolved independently (Resende et al., 2011; Gonçalves and Resende, 2012). During the evolutionary process, a few elite individuals are periodically selected from each population and added to the others, replacing their worst individuals. This additional feature is mainly used to accelerate the convergence of the algorithm.

With respect to implementations of genetic algorithms in general, there exists a wide range of application programming interfaces (APIs) designed to reuse code and simplify the life of the user, see e.g., Keijzer et al. (2002) and Meffert (2011). We note that these are usually complex since the encoding must be selected by the user, together with both the evolution and mutation operators, as well as the

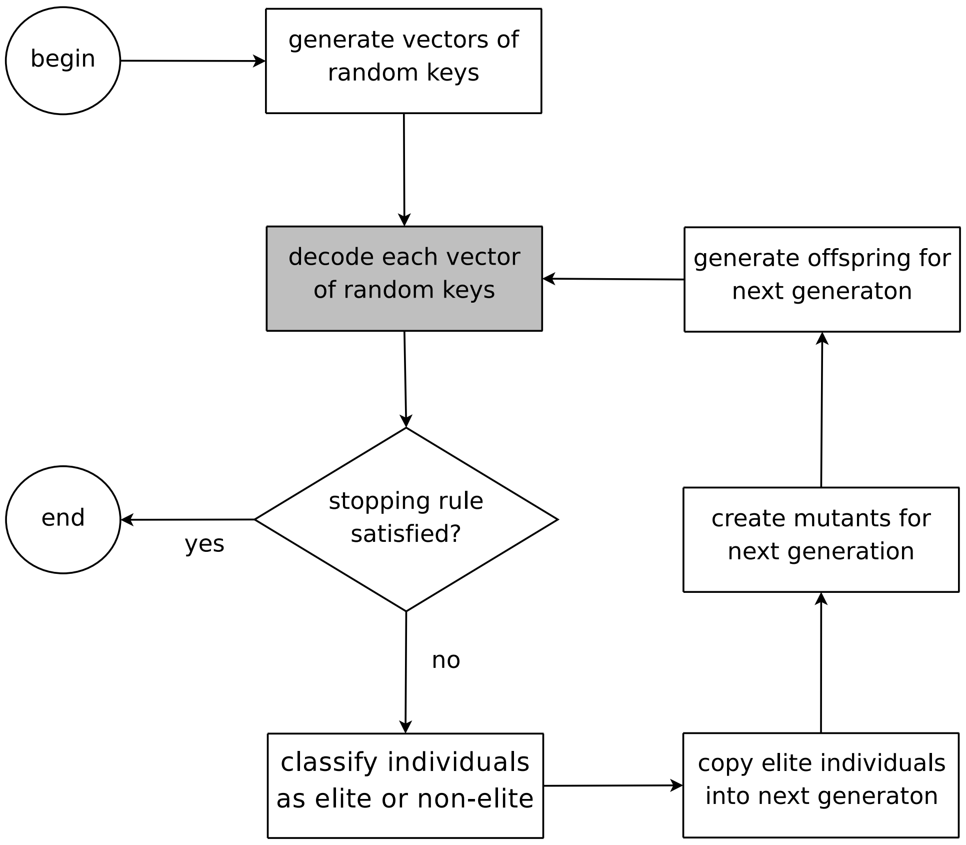
begin
generate vectors of random keys
decode each vector of random keys
generate offspring for next generaton
stopping rule satisfied?
end
yes
no
create mutants for next generation
classify individuals as elite or non-elite
copy elite individuals into next generaton

Figure 1. The BRKGA framework.

decoding process. On the other hand, as aforementioned, designing a BRKGA only entails the implementation of the decoder – everything else is problem-independent.

Despite the recent success cases obtained with the application of BRKGAs to problems such as covering (Resende et al., 2011), network planning (Buriol et al., 2010), scheduling (Gonçalves et al., 2009; 2010), and telecommunications (Resende, 2011; Noronha et al., 2011; Reis et al., 2011), we have not found in the literature any attempt towards a reusable implementation of the BRKGA framework. Therefore, in this paper we propose `brkgaAPI`, an API for BRKGAs that implements all the key components of the framework and includes support for evolving multiple independent populations with periodic exchange of elite individuals. When implemented by the compiler and available in the computing environment, calls to decode the $p - p_e$ chromosomes at each generation are dispatched in parallel, in an attempt to speed up the computational time. The implementation efforts of the user are mainly directed toward the problem-dependent decoder.

The paper is organized as follows. In Section 2 we describe `brkgaAPI` and argue in support of the design choices made. Experiments are shown in Section 3. Concluding remarks are made in Section 4.

2. An API for the BRKGA framework

We propose `brkgaAPI`, a simple but efficient application programming interface that implements the BRKGA framework described in Section 1. This library is written in the C++ programming language and relies on its standard template library (STL) (see, e.g., Stroustrup (2000)). When implemented by the compiler, it also makes use of OpenMP (OpenMP, 2011), a multi-platform API for shared-memory parallel programming. Henceforth, we assume the reader is familiar with C++, especially with the STL, templates, and const-correctness.

With simplicity as a goal, our design comprises two classes, `Population` and `BRKGA`: the first simply defines, stores, and provides easy access to a population of individuals, while the second corresponds to the algorithm itself, allowing the user to manipulate a set of independent populations in accordance with the guidelines of the framework detailed in Section 1.

2.1. Class `Population`.

This class encapsulates a set of p individuals labeled $\{0, 1, \dots, p-1\}$, each one containing a chromosome with n alleles and its corresponding fitness value. Here, a chromosome is defined as an STL `vector` of double-precision floating points representing the random-keys, i.e., `std::vector< double >`.

To the user, this class is only useful when accessing information about a population that is being evolved by the algorithm, i.e. to obtain the chromosome that encodes the best solution and its corresponding fitness. However, for operations like crossover, it plays an essential role in organizing the individuals to be manipulated by the algorithm (see Section 2.2 below). The following API is available to the user:

- `unsigned getN() const`: returns the size of each chromosome;
- `unsigned getP() const`: returns the size of the population;
- `double getBestFitness() const`: returns the best fitness value;
- `double getFitness(unsigned i) const`: returns the fitness value of the i-th best chromosome. Letting $i = 0$ results in `getBestFitness`;
- `const std::vector< double >& getChromosome(unsigned i) const`: returns a read-only reference to the i-th best chromosome.

All methods above run in constant time. For efficiency, none of them is range-checked and thus runtime errors may occur if the user tries to access information of a chromosome outside the set of indices $\{0, 1, \dots, p-1\}$.

2.2. Class `BRKGA`.

In this class we encapsulate a multi-population biased random-key genetic algorithm having K independent populations and provide methods for initialization, evolution, exchange of elite individuals among the independent populations, as well as re-initialization of the populations (useful when one wants to restart the algorithm). The adopted architecture instantiates two `Population` objects for each independent population: one to hold the current generation and another for the subsequent generation. This way, the evolution step is done in place and no new memory is allocated throughout the generations – a simple swap of pointers is performed when the population of the current generation is evolved into the subsequent generation.

The framework as depicted in Figure 1 actually requires two external blocks: one is the decoder, discussed extensively beforehand; the other is a random number generator. In C++ there are basically two ways to bind the user-implemented tools inside an API: dynamically (during execution), using object-oriented polymorphism through virtual methods defined in a base class, or statically (at compile-time), using templates in the API. In short, the former uses a virtual method table for runtime method-binding, while the latter relies on the compiler for method binding. Efficiency wise, the latter runs faster while compiling slower. Based on this discussion, we implemented the key class of the API as a C++ template class with signature

```
template< class Decoder, class RNG > class BRKGA.
```

Therefore, to be able to use the API, the user must implement the interfaces `Decoder` and `RNG`. Before we delve into the API itself, we next discuss the requirements for each of these classes. Failures to comply with these requirements are reported by the compiler.

2.2.1. *Interface: `Decoder`.* Exactly one of the following two methods must be implemented in the class serving as the decoder:

- `double decode(std::vector< double >& chromosome) const`: enables read-write access to a chromosome; or
- `double decode(const std::vector< double >& chromosome) const`: enables read-only access to the chromosome.

When compiling and linking the class `BRKGA` with multi-threading support, the `decode` method must be thread-safe, which is guaranteed by const-correctness in both declarations above. The main difference between the above signatures is that the first can be used to change the chromosome, for example to reflect into it a solution found by the decoder (see e.g. the discussion on chromosome correction in Resende et al. (2011)).

2.2.2. *Interface: `RNG`.* The following methods must be implemented in the class serving as the uniform (pseudo) random number generator:

- `double rand()`: must return a random floating-point in range $[0, 1)$;
- `unsigned long randInt(unsigned N)`: must return a random integer in range $[0, N]$.

An implementation of the *Mersenne Twister* random number generator (Matsumoto and Nishimura, 1998) is currently distributed with our library.

2.2.3. *The `BRKGA` API.* We next describe the API to initialize and manipulate a biased random-key genetic algorithm. Below is a list of all the methods available to the user, with pertinent details regarding their implementation.

- `BRKGA(unsigned n, unsigned p, double pe, double pm, double rho, const Decoder& refDecoder, RNG& refRNG, unsigned K = 1, unsigned MAXT = 1)`: constructor to instantiate a `BRKGA` object. Below is a brief description of each parameter:
  - `n`: number of alleles in each chromosome;
  - `p`: number of elements in each population;
  - `pe`: fraction of population made up of elite items;
  - `pm`: fraction of population made up of mutants introduced in the population at each generation;
  - `rho`: probability that an offspring inherits the allele of its elite parent;

Table 2. Running times and number of calls to interfaced methods. The running times do not include the time spent in decoding and random number generation.

Method	Running time	`decode`	`rand`	`randInt`
`BRKGA`	$\Theta(np + p \log p)$	p	np	0
`evolve`	$\Theta(np + p \log p)$	$p - p_e$	$n(p - p_e)$	$p - p_e - p_m$
`reset`	$\Theta(np + p \log p)$	p	np	0

- `refDecoder`: const reference to the class implementing the `Decoder` interface (discussed in Section 2.2.1);
- `refRNG`: reference to the class implementing the `RNG` interface (discussed in Section 2.2.2);
- `K` (optional; defaults to 1): number of independent populations;
- `MAXT` (optional; defaults to 1): maximum number of threads used to perform parallel decoding.

Notice that const-correctness is used here to help guarantee that the `decode` method implemented in the `Decode` interface is thread-safe. In doing so, we make sure that calls from `BRKGA` to `decode` are not allowed to change the internal state of the `Decoder`, thus avoiding race conditions in a multi-threaded environment.

- `void evolve(unsigned generations = 1)`: evolves the populations following the guidelines of the algorithm. The user may optionally provide the number of generations.
- `void reset()`: resets all populations with randomly generated chromosomes.
- `void exchangeElite(unsigned M)`: exchanges elite individuals among the populations; for each population, selects the top M elite individuals and copies them to the remaining populations, substituting their worst individuals.
- `const Population& getPopulation(unsigned k = 0) const`: returns a const reference to the k-th population. Refer to Section 2.1 for the API available in the class `Population`.
- `double getBestFitness() const`: returns the best overall fitness.
- `const std::vector< double >& getBestChromosome() const`: returns a const reference to the chromosome with best overall fitness.

Operation counts for the key methods discussed above are provided in Table 2. Here, for simplicity, we assume that $K = 1$ for all methods and that `evolve` is called with `generations = 1` as default.

2.2.4. *Sample decoder and biased random-key genetic algorithm.* We conclude this section with sample code for implementation of a simple decoder and a BRKGA driver whose C++ source codes are listed in Figures 2 and 3, respectively. Two tasks are done in the decoder. First, the fitness is computed as

$$\texttt{myFitness} = \sum_{i=0}^{n-1} (i+1) \cdot \texttt{chromosome}[i].$$

Then, a permutation is generated by means of pairing each allele with its position in the chromosome and sorting the pairs in increasing order of their allele values – the sorted sequence of pairs stores a permutation of $\{0, 1, \ldots, n-1\}$ as the second member of each of the pairs. Such a permutation is stored in a `std::list` object named `permutation`. The BRKGA runs for 1,000 generations, evolving three populations, each with 1,000 random-key vectors of size 100. 200 elements of each population are classified as elite, while 100 mutants are inserted in each population at each generation. Random-key vectors are decoded with the `SampleDecoder` shown in Figure 2. The probability that an offspring solution inherits the allele of the elite parent is 70%, Every 100 generations the three populations exchange their two top solutions. The cost of the best solution found is printed at the end.

3. Experiments

In this section, we evaluate our implementation under scenarios that include running times and parallel efficiency, both taking into account different decoding complexities. Since initializing the algorithm differs from evolving the population only by an extra few calls to the decoder (p_e, to be exact) plus memory allocation, from this point on we only focus on the method `evolve`. For all the experiments, we fixed the values of the following parameters: $p_e = \lceil 0.15p \rceil$, $p_m = \lceil 0.10p \rceil$, and $\rho_e = 0.7$. The tradeoffs between the remaining parameters, i.e. n, p, and number of threads for parallel decoding, as well as the complexity of the decoder, are the main subject of our investigation.

We compiled the BRKGA API with the `gcc` C++ compiler `g++` version 4.1.2 and carried out the experiments on a machine running Linux 2.6.18 on 32 2.3 GHz AMD Opteron CPUs with access to 128 Gb of RAM.

3.1. Single-threaded decoding.

Assuming that generating a random number runs in $O(1)$, the evolution of a population in one generation entails filling in np memory cells with random keys in $\Theta(np)$-time, followed by decoding $p - p_e$ chromosomes, where each chromosome is decoded in $\Omega(n)$ time, followed by sorting p fitness values in $O(p \log p)$ time. We next investigate a throughput measure – number of calls to the decoder per second – obtained when evolving a single population for 100 generations. For this purpose, we implemented a simple decoder that copies the length-n chromosome into a local `vector` object and returns its first allele as the fitness value, thus running in $\Theta(n)$.

In the plot displayed in Figure 4, we varied the size of the population p for four fixed values of n. Though for larger values of n such as 1,000 or 10,000 the throughput does not change when p is increased, it does decrease for smaller values of n, such as 10 or 100. For the case of larger values, filling in the genes for the next population and decoding seem to dominate the running time, whereas for case of smaller values, sorting the fitness values impacts most the performance of the algorithm. Similar experiments with slower, heavy-duty decoders (such as those with running time $\Theta(n \log n)$, $\Theta(n^2)$, and so on) exhibit the same trend found in populations with larger chromosomes since decoding dominates the other tasks.

The lesson learned in this experiment is compatible with the theoretical bounds for `evolve` that were given in Table 2: $\Theta(np)$ alleles are written to form the next generation, plus $p - p_e$ calls are made to a decoder that runs in $\Theta(n)$, totaling $\Theta(np)$. Sorting the individuals by their fitness runs in $O(p \log p)$ and no chromosome is moved in the process, so it is predictable that the size of the population does not

```
#ifndef SAMPLEDECODER_H
#define SAMPLEDECODER_H
#include <vector>

class SampleDecoder {
public:
    SampleDecoder();
    ~SampleDecoder();

    double decode(const std::vector< double >& chromosome) const;

SampleDecoder::SampleDecoder() :  internal(123456789) { }

SampleDecoder::~SampleDecoder() { }

double SampleDecoder::decode(const std::vector< double >& chromosome) const {
    // Here we store the fitness of ‘chromosome’
    double myFitness = 0.0;

    std::vector< std::pair< double, unsigned > > ranking(chromosome.size());

    // Here we compute the fitness as the inner product of the random-key vector,
    // i.e.  the ‘chromosome’, and the vector of indices [0,1,2,...,n-1]
    for (unsigned i = 0; i < chromosome.size(); ++i) {
        ranking[i] = std::pair< double, unsigned >(chromosome[i], i);
        myFitness += (double(i+1) * chromosome[i]);
    }

    // Here we sort ‘ranking’, which will then produce a permutation of [n]
    // in pair::second:
    std::sort(ranking.begin(), ranking.end());

    // Here we obtain the permutation of [n]:
    std::list< unsigned > permutation;
    for(std::vector< std::pair< double, unsigned > >::const_iterator i = ranking.begin();
            i != ranking.end(); ++i) {
        permutation.push_back(i->second);
    }

    // Here we return the fitness of ‘chromosome’:
    return myFitness;
}

#endif
```

Figure 2. Sample code for a simple decoder. Two tasks are done in the decoder. The decode first computes the fitness of the solution and then generates a permutation by means of pairing each allele with its position in the chromosome and sorting the pairs in increasing order of their allele values. Such a permutation is stored in a `std::list` object named `permutation`.

```
#include <iostream>
#include "SampleDecoder.h"
#include "MTRand.h"
#include "BRKGA.h"
int main(int argc, char* argv[]) {
    const unsigned n = 100; // Size of chromosomes
    const unsigned p = 1000; // Size of population
    const double pe = 0.20; // Fraction of population to be the elite-set
    const double pm = 0.10; // Fraction of population to be replaced by mutants
    const double rhoe = 0.70; // Probability offspring inherits elite parent allele
    const unsigned K = 3; // Number of independent populations
    const unsigned MAXT = 2; // Number of threads for parallel decoding

    SampleDecoder decoder; // Initialize decoder

    const long unsigned rngSeed = 0; // Seed random number generator
    MTRand rng(rngSeed); // Initialize random number generator

    // Initialize BRKGA-based heuristic
    BRKGA< SampleDecoder, MTRand >
                algorithm(n, p, pe, pm, rhoe, decoder, rng, K, MAXT);

    unsigned generation = 1; // Current generation
    const unsigned X_INTVL = 100; // Exchange best individuals every 100 generations
    const unsigned X_NUMBER = 2; // Exchange top 2 best
    const unsigned MAX_GENS = 1000; // Run for 1000 generations

    // Iterations of the algorithm ...
    do {
        algorithm.evolve(); // Evolve the population for one generation

        if((++generation) % X_INTVL == 0) {
            algorithm.exchangeElite(X_NUMBER); // Exchange top individuals
        }
    } while (generation < MAX_GENS);

    std::cout << "Best solution found has objective value = "
                         << algorithm.getBestFitness() << std::endl;

    return 0;
}
```

FIGURE 3. Sample code for a BRKGA with 1,000 random-key vectors of size 100, 200 of which are elite, 100 mutants, with 70% probability that offspring solution inherits allele of its elite parent. Three populations are evolved, exchanging their two top solutions every 100 generations. The algorithm is run for a total of 1,000 generations. The cost of the best solution found is printed at the end. Random-key vectors are decoded with the `SampleDecoder` shown in Figure 2.

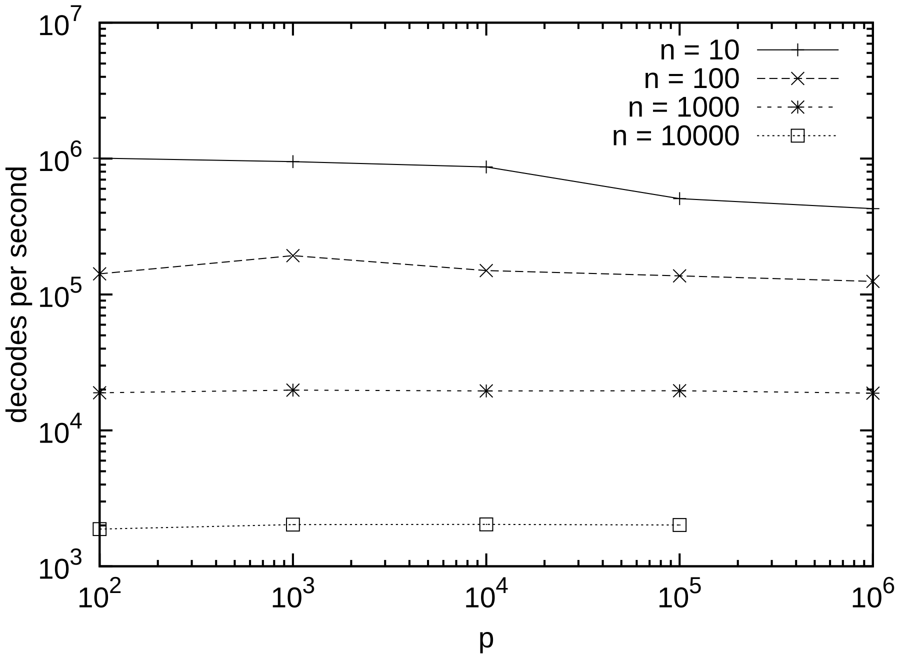

FIGURE 4. Effect of n and p in the runtime of `evolve`.

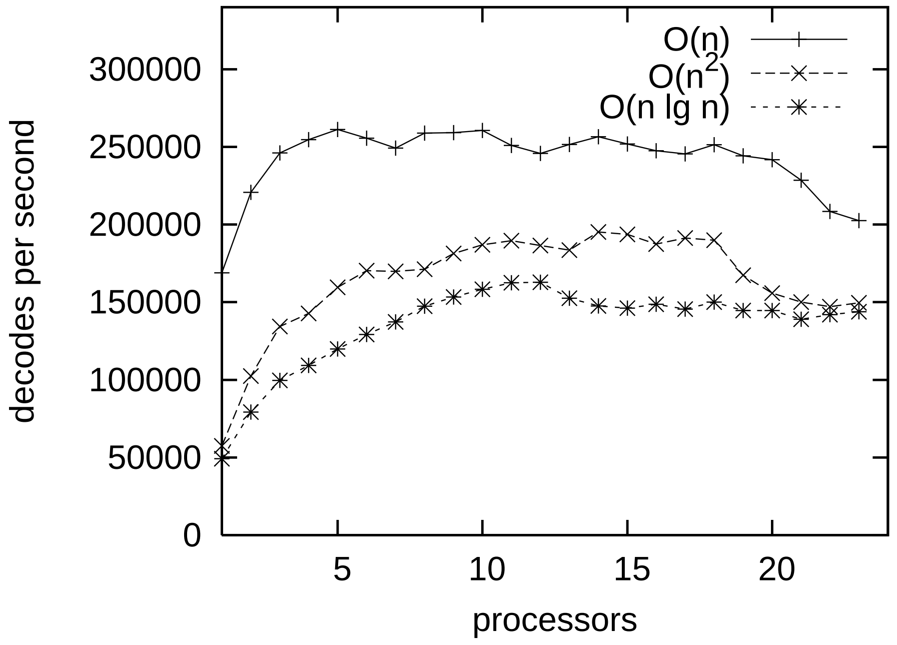

FIGURE 5. Parallel decoding throughput as a function of the number of processors for $p = 10,000$ and $n = 100$.

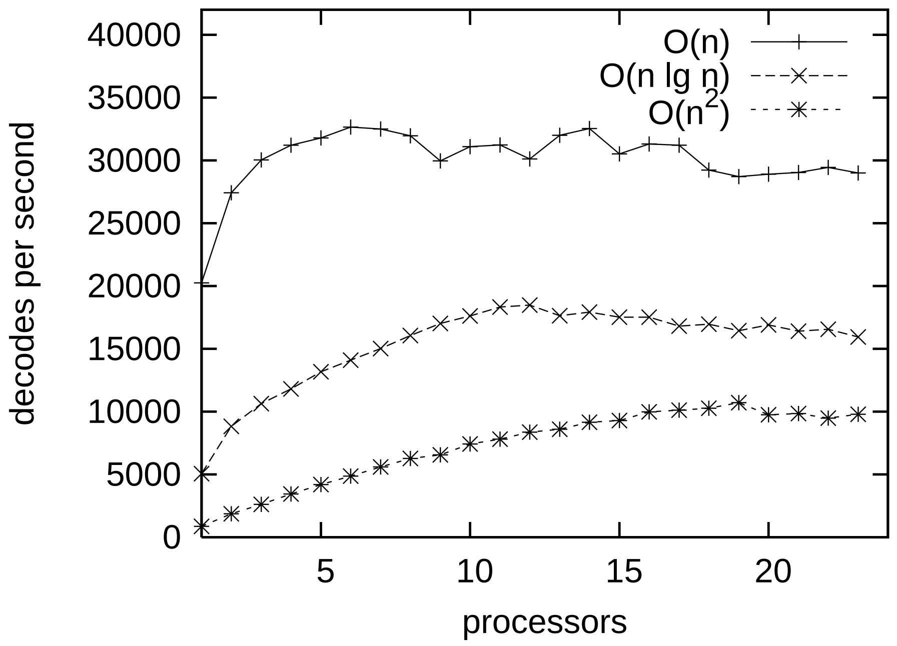

Figure 6. Parallel decoding throughput as a function of the number of processors for $p = 1,000$ and $n = 1,000$.

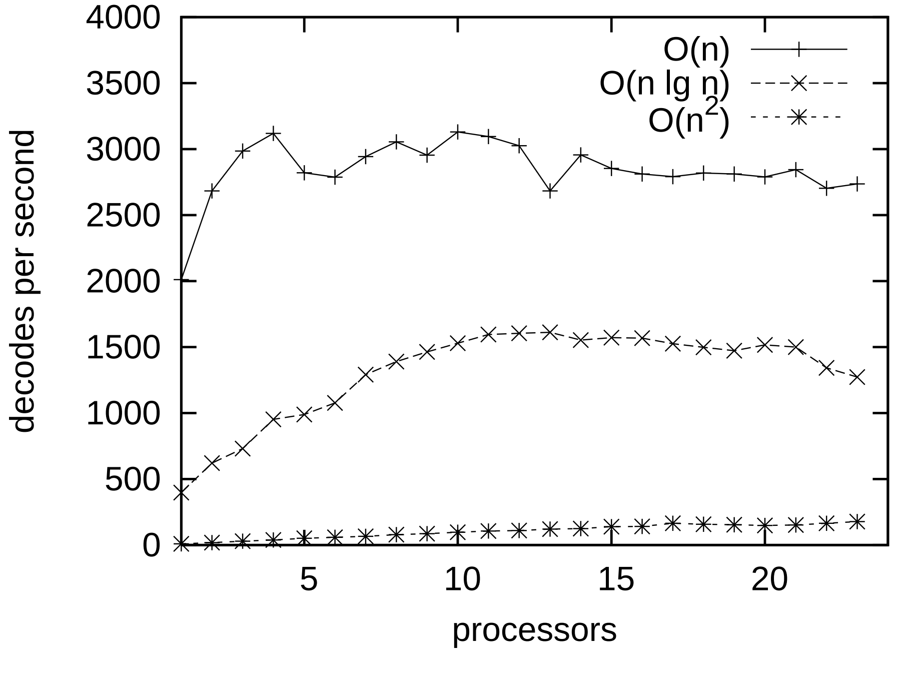

Figure 7. Parallel decoding throughput as a function of the number of processors for $p = 100$ and $n = 10,000$.

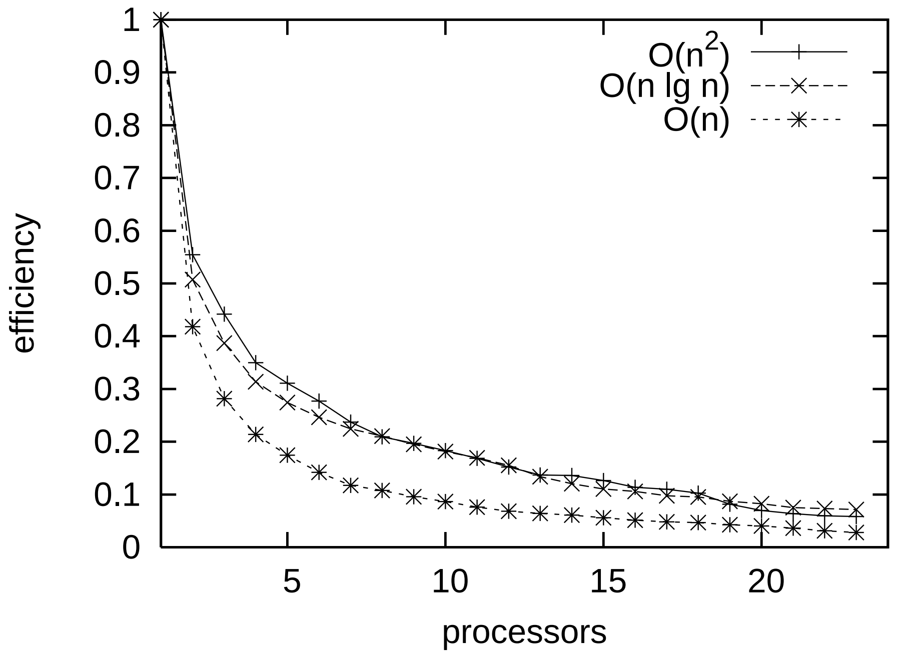

Figure 8. Parallel efficiency as a function of the number of processors for $p = 10,000$ and $n = 100$.

interfere in the throughput of the algorithm for sufficiently large n. To summarize, the size of the population should not interfere with the throughput of the algorithm in the vast majority of situations; in other words, in practice the increase in the running time incurred by the increase of p should have a close-to-linear relation.

3.2. **Multi-threaded decoding.** We next fix both n and p while varying the number of processors up to 24 to evolve a single population for 100 generations. We have considered three types of decoders: lightweight, fine grained, decoders with $\Theta(n)$ running time; sorting-based decoders (useful for generating permutations, for example) running in $\Theta(n \log n)$ time, and coarse-grained decoders with complexity $\Theta(n^2)$. The throughput for $n = 100$ and $p = 10,000$ is plotted in Figure 5; for $n = 1,000$ and $p = 1,000$ in Figure 6; and for $n = 10,000$ and $p = 100$ in Figure 7. In all three configurations of n and p, coarse-grained decoders benefited more from parallelism than did fine-grained decoders. We believe that in the former case, decoding the chromosome has more weight over bookkeeping tasks such as synchronization and scheduling. When the decoder is running too fast, it is better to run fewer threads – incurring less bookkeeping – to do the job.

To conclude this section, we move away from throughput to report the parallel efficiency

$$\text{efficiency}(P) = \frac{t_1}{P \cdot t_P}, \tag{1}$$

of our implementation, where t_i denotes the wall-clock execution time achieved with i processors under the same configuration adopted in the previous experiment. The

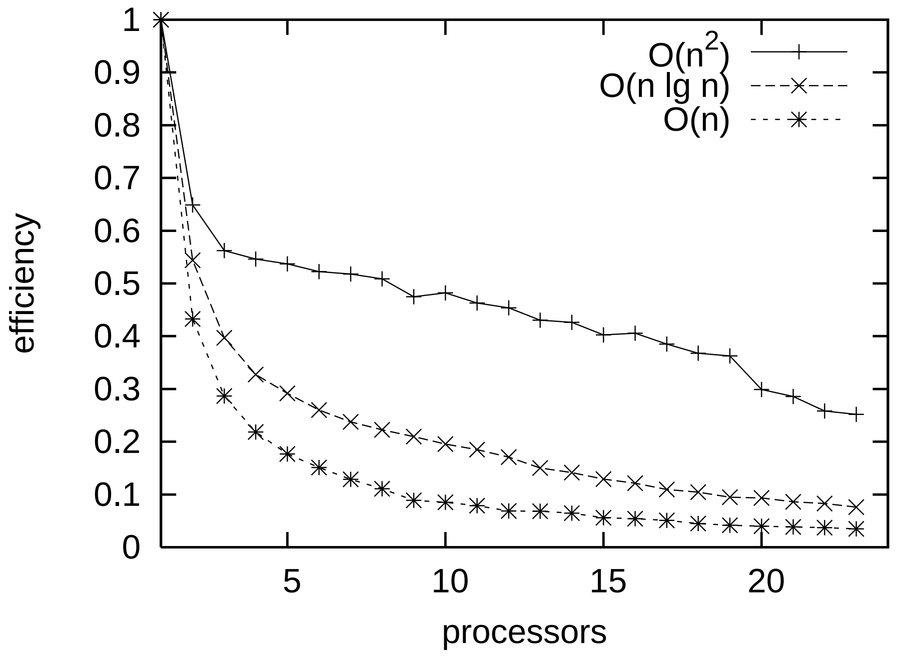

FIGURE 9. Parallel efficiency as a function of the number of processors for $p = 1,000$ and $n = 1,000$.

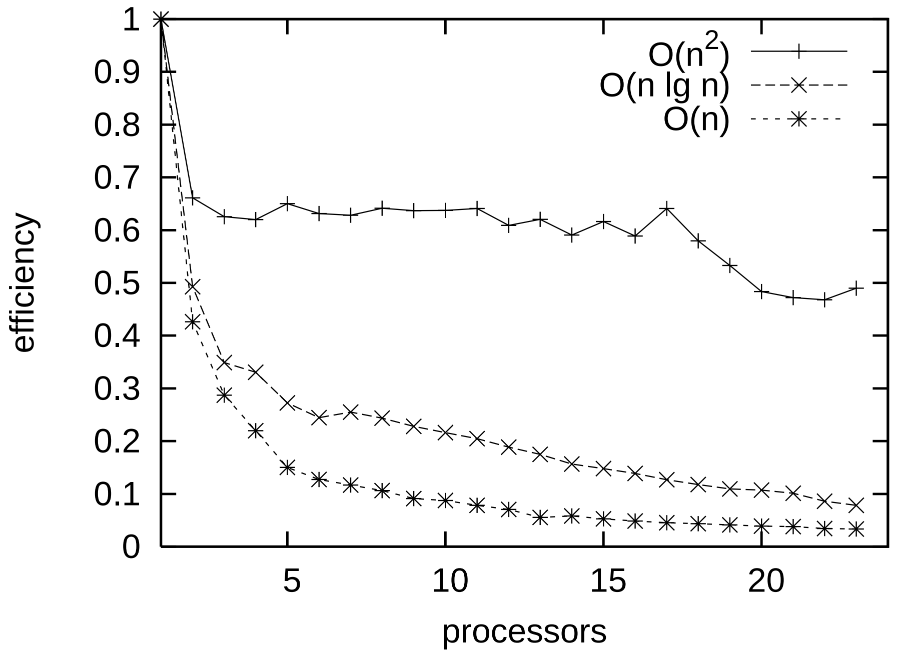

FIGURE 10. Parallel efficiency as a function of the number of processors for $p = 100$ and $n = 10,000$.

plots in Figures 8, 9, and 10 show efficiencies for fine-grained, sorting-based, and coarse-grained decoders, as a function of number of processors, for $\{p = 10,000, n = 100\}$, $\{p = 1,000, n = 1,000\}$, and $\{p = 100, n = 10,000\}$, respectively. These figures show that for fine-grained and sorting-based decoders, parallel efficiency is somewhat unaffected by n, while for coarse-grained decoders it grows with n. Figure 10 shows that, for the coarse-grained decoder with $n = 10,000$, efficiency is maintained above 50% up to the 24-processor configuration. Again, the greater the size of the chromosome and the slower the decoder, the better the efficiency obtained by our parallel implementation.

4. Concluding remarks

In this paper, we introduced `brkgaAPI`, an API that implements all the problem-independent components of the BRKGA framework, and provides the user with tools to implement a decoder that translates chromosomes – represented as a vector of random keys uniformly distributed over $[0, 1)$ – into corresponding solutions and objective values of the optimization problem to be solved.

The API is implemented in C++ and uses the OpenMP API to parallelize the calls to the decoding step into multiple threads. This library is easy to use and yet efficient. It is available at `http://www2.research.att.com/~mgcr/src/brkgaAPI`.

References

J.C. Bean. Genetic algorithms and random keys for sequencing and optimization. *ORSA Journal On Computing*, 2(6):154–160, 1994.

L.S. Buriol, M.J. Hirsch, T. Querido, P.M. Pardalos, M.G.C. Resende, and M. Ritt. A biased random-key genetic algorithm for road congestion minimization. *Optimization Letters*, 4:619–633, 2010.

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

J.F. Gonçalves and M.G.C. Resende. A parallel multi-population biased random-key genetic algorithm for a container loading problem. *Computers and Operations Research*, 29:179–190, 2012.

J.F. Gonçalves, J.J.M. Mendes, and M.G.C. Resende. A random key based genetic algorithm for the resource constrained project scheduling problems. *Computers and Operations Research*, 36:92–109, 2009.

J.F. Gonçalves, M.G.C. Resende, and J.J.M. Mendes. A biased random-key genetic algorithm with forward-backward improvement for the resource constrained project scheduling problem. *Journal of Heuristics*, 2010. doi: 10.1007/s10732-010-9142-2.

M. Keijzer, J.J. Merelo, G. Romero, and M. Schoenauer. Evolving objects: A general purpose evolutionary computation library. *Artificial Evolution*, 2310: 829–888, 2002.

M. Matsumoto and T. Nishimura. Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30, 1998.

K. Meffert. JGAP – Java genetic algorithms and genetic programming package. `http://jgap.sf.net`, 2011. [Online; accessed on 07/26/2011].

T.F. Noronha, M.G.C. Resende, and C.C. Ribeiro. A biased random-key genetic algorithm for routing and wavelength assignment. *Journal of Global Optimization*, 50:503–518, 2011.

OpenMP. `http://openmp.org`, 2011. [Online; accessed on 08/08/2011].

R. Reis, M. Ritt, L.S. Buriol, and M.G.C. Resende. A biased random-key genetic algorithm for OSPF and DEFT routing to minimize network congestion. *International Transactions in Operational Research*, 18:401–423, 2011.

M.G.C. Resende. Biased random-key genetic algorithms with applications in telecommunications. *TOP*, 2011. doi: 10.1007/s11750-011-0176-x.

M.G.C. Resende, R.F. Toso, J.F. Gonçalves, and R.M.A. Silva. A biased random-key genetic algorithm for the Steiner triple covering problem. *Optimization Letters*, 2011. doi: 10.1007/s11590-011-0285-3.

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 Professional, 3rd edition, 2000.

Acknowledgment

The research of Rodrigo F. Toso was mainly done while he was an intern at AT&T Labs Research in the summers of 2010 and 2011. The authors would like to thank Aaron Archer for interesting discussions that led to improvements in earlier versions of the API.

(Rodrigo F. Toso) Department of Computer Science, Rutgers University, 110 Frelinghuysin Road, Piscataway, NJ 08854-8019 USA.
E-mail address: `rtoso@cs.rutgers.edu`

(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`