

**χ -ary Extended Compact Genetic Algorithm in
C++**

**Luis de la Ossa, Kumara Sastry, and
Fernando G. Lobo**

**IlliGAL Report No. 2006013
March, 2006**

Illinois Genetic Algorithms Laboratory
University of Illinois at Urbana-Champaign
117 Transportation Building
104 S. Mathews Avenue Urbana, IL 61801
Office: (217) 333-2346
Fax: (217) 244-5705

χ -ary Extended Compact Genetic Algorithm in C++

Luis de la Ossa¹, Kumara Sastry¹, Fernando G. Lobo²

¹Illinois Genetic Algorithms Laboratory (IlliGAL)
Department of Industrial and Enterprise Systems Engineering
University of Illinois at Urbana-Champaign, Urbana IL 61801

² DEEI-FCT, University of Algarve
Campus de Gambelas, 8000-117 Faro, Portugal

March 27, 2006

Abstract

This report provides documentation for the χ -ary extended compact genetic algorithm (χ ECGA) that solves problems with χ -ary alphabets. The source code is an extension of the original binary-coded extended compact genetic algorithm (ECGA) (Harik, 1999) and its previous implementation in C++ (Lobo & Harik, 1999; Lobo, Sastry, & Harik, 2006). Each decision variable in the current implementation can be of different cardinalities and the χ ECGA finds linkage groups among the decision variables.

1 Introduction

In this report we briefly describe how to download compile and run the χ -ary extended compact genetic algorithm (χ ECGA). The source code is an extension of the original binary ECGA described in Harik's paper (Harik, 1999) and its implementation in C++ (Lobo & Harik, 1999; Lobo, Sastry, & Harik, 2006). It also explains how to modify the objective function that comes with the distribution of the code. The source is written in C++ but a knowledge of the C programming language is sufficient to modify the objective function so that you can try the χ ECGA on your own problems.

With χ ECGA, each decision variable can have a different user-specified cardinality. That is, we are not restricted to optimizing binary variables.

2 How to download the code?

The code is available from `ftp://ftp-illigal.ge.uiuc.edu/pub/src/ECGA/chiECGA.tgz`. After downloading it, uncompress and untar the file by typing

```
tar zxvf chiECGA.tgz
```

At this point you should have in your directory the following files:

DISCLAIMER	chromosome.cpp	inputfile	objfunc.cpp	random.hpp
CHANGES	chromosome.hpp	intlist.cpp	objfunc.hpp	subset.cpp
Makefile	ecga.cpp	intlist.hpp	parameter.hpp	subset.hpp
README	ecga.hpp	main.cpp	population.cpp	utility.cpp
cache.cpp	gene.cpp	mpm.cpp	population.hpp	utility.hpp
cache.hpp	gene.hpp	mpm.hpp	random.cpp	

3 How to compile the code?

We assume you have a C++ compiler properly installed on your computer. We have compiled the code using GNU C++ and tested under the Linux operating systems. For windows operating systems, we have used the GNU C++ compiler included in MinGW (<http://www.mingw.org>). To compile the code, type `make` on the UNIX shell prompt. After that, you should have an executable file called `chiEcga`.

4 How to run the code?

The executable `chiEcga` needs two arguments: the name for an input file and the name for an output file. The χ ECGA reads its parameters from the input file and stores the results of the run in the output file. A sample input file is provided as an example with the distribution of this code. The filename is called `inputfile` and its contents (along with line numbers) is shown below:

```

1 #
2 # Sample parameter file.
3 # Don't change the order of the lines in this file.
4 #
5 BEGIN
6 chromosome_length           20
7 values_per_gene             3 4 2 5 4 4 3 2 5 2 3 2 4 4 4 3 2 5 3 5
8 seed                         0.254534
9 population_size              4000
10 probability_crossover        1
11 tournament_size              16
12 learn_MPM                    on
13 stop_criteria                 allele_convergence
14 stop_criteria_argument        0
15 #
16 # reporting flags
17 #
18 report_pop                    off
19 report_string                  on
20 report_fitness                 on
21 report_MPM                     on
22 END

```

The `chiEcga` skips all the lines until it reaches the word `BEGIN` (line 5 in the example above). Then it starts reading the parameters in a predefined order. The program doesn't do any fancy parsing on the input file. This means that after the word `BEGIN`, **you should not change the order of the lines in the input file** because otherwise the `chiEcga` will get totally confused. The input file is straightforward to understand. Below is an explanation of each of its lines:

Line 6 indicates that the problem length (# of variables) is **20**.

Line 7 indicates the cardinalities of the 20 variables is **3, 4, 2, 5, 4, 4, 3, 2, 5, 2, 3, 2, 4, 4, 4, 3, 2, 5, 3**, and **5** respectively. **Make sure that the number of entries in this line equals to the problem length specified in the above line (Line 6)**. The minimum value of the cardinality is 2, which means that a variable can have only two values (binary).

Line 8 indicates that **0.254534** is the seed to initialize the pseudo random number generator. **The value for the seed must be a number between 0 and 1.**

Line 9 indicates that the population size is **4000**. See (Goldberg, Deb, & Clark, 1992; Harik, Cantú-Paz, Goldberg, & Miller, 1999; Pelikan, Sastry, & Goldberg, 2003; Sastry & Goldberg, 2004) for guidelines for setting population sizing in χ ECGA.

Line 10 says that the probability of crossover is **1**. That is, the whole population is regenerated after each generation cycle.

Line 11 indicates that the tournament size is **16**. The only selection method implemented is the tournament selection without replacement (Goldberg, Korb, & Deb, 1989; Sastry & Goldberg, 2001).

Line 12 indicates that the χ ECGA learns the marginal product model (MPM) every generation. You can set this option **on** and **off**. If set to **on**, you get the normal χ ECGA. If set to **off**, you get the χ -ary compact GA (Harik, Lobo, & Goldberg, 1998).

Line 13 indicates that the χ ECGA stops when the population has fully converged. That is, when the population consists of n copies of the same individual, where n is the population size. Besides the `allele_convergence`, you can also choose the `max_generations` option.

Line 14 indicates the maximum number of generations in case you choose the `max_generations` option on the previous line (Line 12). If you choose the `allele_convergence` option, then this parameter is irrelevant.

Line 18 indicates that the population should not be stored in the output file at the end of each generation. You can set this option **on** or **off**. If set to **on**, you should be careful as the output file size can easily become quite large.

Line 19 indicates that the best chromosome of every generation is stored in the output file. you can set this option **on** or **off**.

Line 20 indicates that the best fitness and the average fitness of the population at the end of each generation is stored in the output file. you can set this option **on** or **off**.

Line 21 indicates that the MPM—including the greedy search steps taken in the construction of the MPM—for each generation is stored in the output file. you can set this option **on** or **off**.

Now you are ready to go ahead and run the χ ECGA. At the prompt, type

```
chiEcga inputfile outputfile
```

Population statistics are displayed on the screen at the end of each generation. The same information is also sent to the `outputfile`. In addition, the `outputfile` also shows the different MPM structures that the χ ECGA finds during its MPM search.

The objective function that comes with the distribution of the code is similar to a concatenated $m - k$ trap function. The test problem is a concatenation of 5 copies of a 4-alphabet trap function. The trap function has fitness u , where u is the sum of the variable values, except when the string is 0000, in which case the fitness is . Thus, for the first block of four variables, the global optima is at 0000, with fitness 15, and the local (deceptive) optima is at 2314, and has a fitness of 10. Therefore, for the overall problem, the optimal solution is the string with all zeros and has a fitness of 74.

5 How to plug-in your own objective function?

The code for the objective function is in the function `objective_func()` in the file `objfunc.cpp`. This is the only function that you need to rewrite in order to try your own problem. The function header is as follows:

```
double objective_func(char *chrom, int lchrom)
```

It takes as argument a character string of 1s and 0s, and the string length. The function returns a real number: the objective function of the string. In the current implementation, the `chiEcga` assumes that the decision variables are integers of user-specified cardinality. Note that if you want to use the cardinality of the variables in your objective function, then use `parameter::ranges`. That is, the cardinality i^{th} gene is stored in `parameter::ranges[i]`.

6 About the C++ code

The implementation of the χ ECGA doesn't use advanced features of the C++ language such as templates and inheritance. This means that you don't need to be a C++ expert in order to modify the code. In fact, you can modify the code and plug-in your own objective function using the C programming language alone. Next, we give brief description of the source files. Each `.cpp` file has a corresponding `.hpp` file, except `main.cpp`. The `.hpp` files are the header files and contain the definitions of the various classes. The `.cpp` files contain the actual implementation.

`gene.cpp` contains the implementation of the class `gene`. A gene has a locus and an allele.

`chromosome.cpp` contains the implementation of the class `chromosome`. A chromosome is an array of genes.

`population.cpp` contains the implementation of the class `population`. A population is an array of chromosomes. Selection operators, population statistics, and stopping criteria are implemented here.

`objfunc.cpp` contains the code for the objective function. If you want to try the χ ECGA on your own problem, you should modify the function `objective_func()` contained in this file.

`utility.cpp` contains utility functions and procedures.

`intlist.cpp` implements a list of integers.

`subset.cpp` contains operations that can be done on a subset structure of an MPM.

`mpm.cpp` contains operations that can be done on an MPM.

`cache.cpp` implements a cache used for speedup-up the MPM search.

`random.cpp` contains subroutines related to the pseudo random number generator.

`ecga.cpp` contains the main loop of the χ ECGA.

`main.cpp` contains the `main()` function and the initialization procedures.

7 Disclaimer

This code is distributed for academic purposes only. It has no warranty implied or given, and the authors assume no liability for damage resulting from its use or misuse. If you have any comments or find any bugs, please send an email to kumara@illigal.ge.uiuc.edu.

8 Commercial use

For the commercial use of this code please contact Prof. David E. Goldberg at deg@uiuc.edu

Acknowledgments

This work was also sponsored by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under grant FA9550-06-1-0096, the National Science Foundation under grant ITR grant DMR-03-25939 at the Materials Computation Center. The U.S. Government is authorized to reproduce and distribute reprints for government purposes notwithstanding any copyright notation thereon.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Office of Scientific Research, the National Science Foundation, or the U.S. Government.

We thank Prof. David E. Goldberg for encouraging us to write this report.

References

- Goldberg, D. E., Deb, K., & Clark, J. H. (1992). Genetic algorithms, noise, and the sizing of populations. *Complex Systems*, 6, 333–362. (Also IlliGAL Report No. 91010).
- Goldberg, D. E., Korb, B., & Deb, K. (1989). Messy genetic algorithms: Motivation, analysis, and first results. *Complex Systems*, 3(5), 493–530. (Also IlliGAL Report No. 89003).
- Harik, G. (1999, January). *Linkage learning via probabilistic modeling in the ECGA* (IlliGAL Report No. 99010). Urbana, IL: University of Illinois at Urbana-Champaign.

- Harik, G., Cantú-Paz, E., Goldberg, D. E., & Miller, B. L. (1999). The gambler's ruin problem, genetic algorithms, and the sizing of populations. *Evolutionary Computation*, 7(3), 231–253. (Also IlliGAL Report No. 96004).
- Harik, G., Lobo, F., & Goldberg, D. E. (1998). The compact genetic algorithm. *Proceedings of the IEEE International Conference on Evolutionary Computation*, 523–528. (Also IlliGAL Report No. 97006).
- Lobo, F. G., & Harik, G. R. (1999, June). *Extended compact genetic algorithm in C++* (IlliGAL Report No. 99016). Urbana, IL: University of Illinois at Urbana-Champaign.
- Lobo, F. G., Sastry, K., & Harik, G. R. (2006, March). *Extended compact genetic algorithm in C++: Version 1.1* (IlliGAL Report No. 2006012). Urbana, IL: University of Illinois at Urbana-Champaign.
- Pelikan, M., Sastry, K., & Goldberg, D. E. (2003). Scalability of the Bayesian optimization algorithm. *International Journal of Approximate Reasoning*, 31(3), 221–258. (Also IlliGAL Report No. 2001029).
- Sastry, K., & Goldberg, D. E. (2001). Modeling tournament selection with replacement using apparent added noise. *Intelligent Engineering Systems Through Artificial Neural Networks*, 11, 129–134. (Also IlliGAL Report No. 2001014).
- Sastry, K., & Goldberg, D. E. (2004). Designing competent mutation operators via probabilistic model building of neighborhoods. *Proceedings of the Genetic and Evolutionary Computation Conference*, 2, 114–125. Also IlliGAL Report No. 2004006.