Single and Multiobjective Genetic Algorithm Toolbox in C++

Kumara Sastry

IlliGAL Report No. 2007016
June 2007

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

Single and Multiobjective Genetic Algorithm Toolbox in C++

Kumara Sastry
Illinois Genetic Algorithms Laboratory (IlliGAL),
Materials Computation Center, and
Department of Industrial and Enterprise Systems Engineering
University of Illinois at Urbana-Champaign, Urbana IL 61801

June 5, 2007

Abstract

This report provides documentation for the general purpose genetic algorithm toolbox. The toolbox provides different selection, recombination, mutation, niching, and constraint-handling operators. Problems with single and multiple objectives can be solved with the toolbox. Moreover, the toolbox is easily extensible and customizable for incorporating other operators and for solving user-defined search problems.

1 Introduction

This document briefly describes how to download, compile, and run the GA toolbox. Interested user should refer to the extensive comments in the code for an explanation of the more intricate GA-implementation and the code structure details. This report 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 toolbox for your own problems.

2 How to download the code?

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

```
tar zxvf GAtoolbox.tgz
```

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

```
DISCLAIMER      README          Makefile           chromosome.cpp
chromosome.hpp  crossover.cpp   crossover.hpp      ga.cpp
ga.hpp          globalSetup.cpp globalSetup.hpp    individual.cpp
individual.hpp  localsearch.cpp localsearch.hpp    nsgapopulation.cpp
population.cpp  population.hpp  random.cpp         random.hpp
selection.cpp   selection.hpp   userDefinables.cpp
```

3 How to compile the code?

To compile the code, a C++ compiler needs to be installed on your computer. The code has been compiled using GNU C++ and tested under Linux and Microsoft Visual C++ compiler version 6.0 and GNU C++ compiler distributed under MinGW (`www.mingw.org`) for Windows. To compile the code type `make` on the Linux/Unix shell prompt. After that, you should have an executable file called `GAtbx`.

4 How to run the code?

The executable `GAtbx` needs one argument: the name for an input file. The toolbox reads its parameters from the input file and outputs the results onto the screen (`stdout`). Four sample input files—`input_sga_maxSpec`, `input_sga_minSpec`, `input_nsga_maxSpec`, and `input_nsga_minSpec`—are provided as examples with the distribution of this code.

Two objective functions that comes with the distribution of the code; one is a single objective problem with two constraints and the other is a multiobjective problem with two objectives. The single objective test function is the constrained Himmelblau's function (Rekalaitis, Ravindran, & Ragsdell, 1983). Himmelblau's function is a minimization problem with two decision variables. The objective value for this case is:

$$f(x_1, x_2) = ({x_1}^2 + x_2 - 11)^2 + (x_1 + {x_2}^2 - 7)^2, \tag{1}$$

with the following two constraints

$$\begin{aligned} (x_1 - 5)^2 + x_2^2 &\leq 26, &(2)\\ 4x_1 + x_2 &\leq 20. &(3) \end{aligned}$$

The multiobjective test function is the Kurosawe test function (Deb, Pratap, Agrawal, & Meyarivan, 2002):

$$\begin{aligned} f_1(x_1, x_2, x_3) &= \sum_{i=1}^{2} \left[-10 * \exp\left(-0.2\sqrt{x_i^2 + x_{i+1}^2}\right)\right], &(4)\\ f_2(x_1, x_2, x_3) &= \sum_{i=1}^{3} \left[|x_i|^{0.8} + 5 \sin x_i^3\right]. &(5) \end{aligned}$$

The steps required to write your own fitness function are described in section 7.

To run the GA toolbox, at the command prompt, type

```
GAtbx <input file name>
```

For example to use `input_sga_maxSpec` as the input file type `GAtbx input_sga_maxSpec`.

Relevant statistics are displayed, depending on the input file specifications, on the screen at the end of each generation. For example, for single-objective optimization problem with niching the output is the population. For multiobjective optimization problem the output is the objective function values of candidate solutions in the population. To change the output, you can modify the function

```
std::ostream &operator<< (std::ostream &out, const Population &pop)
```

implemented in the file `population.cpp`.

5 Description of GA toolbox Capabilities

This section presents an overview of the operators and components implemented in the GA toolbox. The toolbox can solve both single- and multi-objective problems with or without constraints. The multiobjective genetic algorithm implemented is the non-dominated sorting GA II (NSGA-II) (Deb, Pratap, Agrawal, & Meyarivan, 2002).

The decision variables of the problem are encoded as real numbers or as integers within their specified ranges. This encoding procedure permits the decision variables to be binary, discrete, χ-ary alphabet or a real number. The decision variable type needs to be specified in the input file and the valid options are `double` or `int`.

The following genetic operators are supported by GAtbx:

Selection The following selection procedures are available in GA toolbox:

1. Tournament selection with replacement (Goldberg, Korb, & Deb, 1989; Sastry & Goldberg, 2001)
2. Tournament selection without replacement (Goldberg, Korb, & Deb, 1989; Sastry & Goldberg, 2001)
3. Truncation selection (Mühlenbein & Schlierkamp-Voosen, 1993)
4. Roulette-Wheel selection (Goldberg, 1989)
5. Stochastic universal selection (SUS) (Baker, 1985; Baker, 1987; Grefenstette & Baker, 1989; Goldberg, 1989)

Recombination/Crossover The recombination procedures implemented in the GA toolbox are:

1. One point crossover (Goldberg, 1989; Sastry, Goldberg, & Kendall, 2005)
2. Two point crossover (Goldberg, 1989; Sastry, Goldberg, & Kendall, 2005)
3. Uniform crossover (Goldberg, 1989; Sastry, Goldberg, & Kendall, 2005)
4. Simulated binary crossover (SBX) (Deb & Agarwal, 1995; Deb & Kumar, 1995)

Mutation Different mutation procedures implemented in the GA toolbox are:

1. Selective mutation, where a randomly selected gene is replaced by a *uniformly distributed* random value in the interval $[x_{i,min}; x_{i,max}]$ with probability p_m.
2. Genewise mutation, where every gene is mutated using Gaussian mutation with probability p_m.
3. Polynomial mutation (Deb & Agarwal, 1995; Deb & Kumar, 1995; Deb, 2001).

Niching Niching methods implemented in the GA toolbox are:

1. Fitness sharing (Goldberg & Richardson, 1987; Goldberg, 1989)
2. Deterministic crowding (Mahfoud, 1992)
3. Restricted tournament selection (Harik, 1995)

Scaling Three scaling procedures are implemented in the GA toolbox:

1. Linear ranking (Baker, 1985; Goldberg, 1989),
2. Sigma scaling (Forrest, 1985; Goldberg, 1989), and

3. Default scaling which is only used if any one of the following conditions hold true: (a) The selection procedure is roulette-wheel or SUS, and the minimum fitness value is negative, or (b) the niching method is fitness sharing and the minimum fitness value is negative.

Constraint handling Three constraint handling methods are implemented in the GA toolbox:

1. Tournament method (Deb, 2001)
2. Linear penalty method, where a weighted sum of the constraint violations of an individual is added to or subtracted from its objective value depending on whether we are minimizing or maximizing the objective function.
3. Quadratic penalty method, which is similar to linear penalty, except that the penalty value is computed as a weighted sum of the square of constraint violation values of a candidate solution.

It is important to note that the constraint handling methods use the constraint violation value and not constraint function value. For example, if the constraint is

$$10x_1 + x_2 \leq 20,$$

$x_1 = 2$, and $x_2 = 2$, the constraint violation value is 2. However, if $x_1 = 1$, $x_2 = 7$, then the constraint violation value is 0.

Local search The local search method implemented in the GA toolbox is the *Nelder-Mead* algorithm (Nelder & Mead, 1965; Press, Flannery, Teukolsky, & Vettering, 1989). Local search is applied to an individual with probability p_{ls}. The solution obtained through the local search is incorporated using the *Baldwinian strategy*—where the local search solution replaces the individual, but the fitness of the individual is not modified—with probability, p_b, and *Lamarckian strategy*—where both the fitness and the individual are replaced with local search solution and its fitness—with probability, $1 - p_b$.

Elitist Replacement In GA toolbox, both the old and new population is sorted according to their fitness and constraint violation. A user specified proportion, p_e, of top individuals are retained and the rest, $n(1 - p_e)$ individuals are replaced by the top individuals in the new population. Here n is the population size.

5.1 Statistics

Various statistics are collected from the population every generation. These not only aid the user with the progress of genetic search, but are also used in many operators and functionalities. Statistics are also used to determine if the genetic search should be terminated or not. The statistics collected are the mean, minimum, and maximum fitness and objective values. Other statistics include, fitness variance and objective variance in the population. For the multiobjective case, the statistics are collected separately for each objective.

5.2 Stopping Criteria

GA toolbox provides the following stopping criteria for the GA runs

1. **Number of Function Evaluations**: This option terminates the run after certain number of function evaluations have been performed by the GA.

2. **Fitness Variance**: This option stops the once the fitness variance of the population reaches below a specified value.

3. **Best Fitness**: This criterion terminates the run once the fitness of the best individual in the population exceeds a specified value.

4. **Average Fitness**: This option stops the run once the average fitness of the population exceeds a specified value.

5. **Average Objective**: This option stops the run once the average objective function value of the population exceeds a specified value.

6. **Change in Best Fitness**: This option terminates the run once the change in fitness of the best individual in the population over the previous generation is below a specified value.

7. **Change in Average Fitness**: This option terminates the run once the change in average fitness of the population over the previous generation is below a specified value.

8. **Change in Fitness Variance**: This option terminates the run once the change in fitness variance of the population over the previous generation is below a specified value.

9. **Change in Best Objective**: This option terminates the run once the change in change in objective value of the best individual in the population over the previous generation is below a specified value.

10. **Change in Average Objective**: This option terminates the run once the change in change in average objective value the population over the previous generation is below a specified value.

11. **Number of Fronts**: This option is available only for NSGA. It terminates the run after the number of non-dominated fronts in the population goes below a specified value.

12. **Number of guys in first front**: This option terminates the run when the number of individuals in the first exceeds a specified value. It is only available only for NSGA.

13. **Change in number of fronts**: This options stops the run when the change in the number of fronts over the previous generating is below a specified value.

6 Input File Format

The input file `input_sga_maxSpec` is shown (along with line numbers) below. As mentioned earlier, the `GAtbx` skips all empty lines and those that start with `#`. 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 **you should not change the order of the lines in the input file** and therefore the choice of operators and parameters will be incorrectly read. The input file is self explanatory and straightforward to understand.

```
1     #
2     # GA type: SGA or NSGA
3     #
4     SGA
```

```

#
# Number of decision variables
#
2

#
# For each decision variable, enter:
#   decision variable type, Lower bound, Upper bound
# Decision variable type can be double or int
#
double 0.0 6.0
double 0.0 6.0

#
# Objectives:
#   Number of objectives
#   For each objective enter the optimization type: Max or Min
#
1
Min

#
# Constraints:
#   Number of constraints
#   For each constraint enter a penalty weight
#
2
1.0
1.0
#
# General parameters: If these parameters are not entered default
#                     values will be chosen. However you must enter
#                     "default" in the place of the parameter.
#   [population size]
#   [maximum generations]
#   [replace proportion]
#
100
100
0.9

#
# Niching (for maintaining multiple solutions)
# To use default setting type "default"
#  Usage: Niching type, [parameter(s)...]
#  Valid Niching types and optional parameters are:
#   NoNiching
#   Sharing [niching radius] [scaling factor]
#   RTS [Window size]
#   DeterministicCrowding
#
#  When using NSGA, it must be NoNiching (OFF).
#
NoNiching

#
# Selection
#  Usage: Selection type, [parameter(s)...]
#  To use the default setting type "default"
#
#  Valid selection types and optional parameters are:
#   RouletteWheel
#   SUS
#   TournamentWOR [tournament size]
#   TournamentWR [tournament size]
#   Truncation [# copies]
#
#  When using NSGA, it can be neither SUS nor RouletteWheel.
#
TournamentWOR 2

#
# Crossover
#  Crossover probability
#  To use the default setting type "default"
#
#  Usage: Crossover type, [parameter(s)...]
#  To use the default crossover method type "default"
#  Valid crossover types and optional parameters are
#       OnePoint
#       TwoPoint
#       Uniform [genewise swap probability]
#       SBX [genewise swap probability][order of the polynomial]
#
0.9
SBX 0.5 10

#
# Mutation
#  Mutation probability
#  To use the default setting type "default"
#
#  Usage: Mutation type, [parameter(s)...]
#  Valid mutation types and the optional parameters are:
#       Selective
#       Polynomial [order of the polynomial]
#       Genewise [sigma for gene #1][sigma for gene #2]...[sigma for gene #ell]
#
0.1
Polynomial 20

#
# Scaling method
#  To use the default setting type "default"
#
#  Usage: Scaling method, [parameter(s)...]
#  Valid scaling methods and optional parameters are:
#       NoScaling
#       Ranking
#       SigmaScaling [scaling parameter]
#
NoScaling

#
# Constraint-handling method
# To use the default setting type "default"
#
# Usage: Constraint handling method, [parameters(s)...]
# Valid constraint handling methods and optional parameters are
#       NoConstraints
#       Tournament
#       Penalty [Linear|Quadratic]
#
Tournament

#
# Local search method
# To use the default setting type "default"
#
# Usage: localSearchMethod, [maxLocalTolerance], [maxLocalEvaluations],
#               [initialLocalPenaltyParameter], [localUpdateParameter],
#               [lamarckianProbability], [localSearchProbability]
#
# Valid local search methods are: NoLocalSearch and SimplexSearch
#
# For example, SimplexSearch 0.001000 20 0.500000 2.000000 0.000000 0.000000
NoLocalSearch

#
# Stopping criteria
# To use the default setting type "default"
#
# Number of stopping criterias
```

```
#
# If the number is greater than zero
#    Number of generation window
#    Stopping criterion, Criterion parameter
#
# Valid stopping criterias and the associated parameters are
#       NoOfEvaluations, Maximum number of function evaluations
#       FitnessVariance, Minimum fitness variance
#       AverageFitness, Maximum value
#       AverageObjective, Max/Min value
#       ChangeInBestFitness, Minimum change
#       ChangeInAvgFitness, Minimum change
#       ChangeInFitnessVar, Minimum change
#       ChangeInBestObjective, Minimum change
#       ChangeInAvgObjective, Minimum change
#       NoOfFronts (NSGA only), Minimum number
#       NoOfGuysInFirstFront (NSGA only), Minimum number
#       ChangeInNoOfFronts (NSGA only), Minimum change
#       BestFitness (SGA with NoNiching only), Maximum value
#
0

#
# Load the initial population from a file or not
# To use the default setting type "default"
#
# Usage: Load population (0|1)
#
# For example, if you want random initialization type 0
# On the other and if you want to load the initial population from a
# file, type
#       1 <population file name> [0|1]
#
# Valid options for "Load population" are 0/1
# If you type "1" you must specify the name of the file to load the
# population from. The second optional parameter which indicates
# whether to evaluate the individuals of the loaded population or not.
0

# Save the evaluated individuals to a file
#
# To use default setting type "default".
#
# Here by default all evaluated individuals are stored and you will be
# asked for a file name later when you run the executable.
#
# Usage: Save population (0|1)
# For example, if you don’t want to save the evaluated solutions type 0
# On the other and if you want to save the evaluated solutions
#       1 <save file name>
#
# Note that the evaluated solutions will be appended to the file.
#
# Valid options for "Save population" are 0/1
# If you type "1" you must specify the name of the file to save the
# population to.
1 evaluatedSolutions.txt

#END
```

---

7 How to plug-in your own objective function?

The code for the objective function is in the file `userDefinables.cpp`. This is the only file that you need to rewrite in order to try your own fitness function. The function header is as follows:

```
void globalEvaluate(double *x, double *objArray, double *constraintViolation,
                    double *penalty, int *noOfViolations)
```

It takes as argument an array `x`, whose ℓ elements contains the decision variables of a candidate solution whose fitness is being evaluated. Here, ℓ is the problem length (# of genes). The objective function value(s) is(are) returned in the array `objArray`, the constraint violation value(s) in the array `constraintViolation`, penalty to be added to the fitness function in `penalty` (used only if the linear or quadratic penalty methods are chosen) constraint handling methods are chosen)and number of constraints violated in `noOfViolations`.

8 About the C++ code

The implementation of the GA toolbox 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 `nsgapopulation.cpp` and `userDefinables.cpp`. The `.hpp` files are the header files and contain the definitions of the various classes. The `.cpp` files contain the actual implementation.

`chromosome.cpp` contains the implementation of the class `chromosome`. A chromosome is an array of genes. It also contains implementation of mutation operators.

`individual.cpp` contains the implementation of the class `individual`. An individual is a candidate solution for a given search and optimization problem.

`population.cpp` contains the implementation of the class `population`. A population is an array of chromosomes. Objective to fitness mappings and statistics computations are implemented here.

`nsgapopulation.cpp` contains the implementation of the class `nsgapopulation`. It builds on the class `population` and implements nondominated sorting and crowding distance computation required for NSGA-II.

`crossover.cpp` contains the implementation of the class `crossover`. Different crossover operators are implemented here.

`selection.cpp` contains the implementation of the class `selection`. Different selection operators are implemented here.

`localsearch.cpp` contains the implementation of the class `localsearch`. A local search operator is implemented here.

`ga.cpp` contains the implementation of the class `ga`. Different GA architectures (GA and NSGA-II) are implemented here.

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

`globalSetup.cpp` contains the implementation of the class `globalSetup`. Global data needed across different classes are implemented here.

`userDefinables.cpp` contains the code for the objective function. If you want to try the GAtbx on your own problem, you should modify the function `globalEvaluate()` contained in this file.

9 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`.

10 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.

I thank Prof. Duane D. Johnson and David E. Goldberg for encouraging me to write this report.

References

Baker, J. E. (1985). Adaptive selection methods for genetic algorithms. *Proceedings of the International Conference on Genetic Algorithms and Their Applications*, 101–111.

Baker, J. E. (1987). Reducing bias and inefficiency in the selection algorithm. *Proceedings of the Second International Conference on Genetic Algorithms*.

Deb, K. (2001). *Multi-objective optimization using evolutionary algorithms*. Chichester, UK: John Wiley and Sons.

Deb, K., & Agarwal, R. (1995). Simulated binary crossover for continuous search space. *Complex Systems*, *9*, 115–148.

Deb, K., & Kumar, A. (1995). Real-coded genetic algorithms with simulated binary crossover: Studies on multimodal and multiobjective problems. *Complex Systems*, *9*, 431–454.

Deb, K., Pratap, A., Agrawal, S., & Meyarivan, T. (2002). A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: NSGA-II. *IEEE Transactions on Evolutionary Computation*, *6*(2), 182–197. (Also KanGAL Report No. 2000001).

Forrest, S. (1985). *Documentation fo PRISONERS DILEMMA and NORMS programs that use genetic algorithms*. Unpublished manuscript, University of Michichan, Ann Arbor.

Goldberg, D. E. (1989). *Genetic algorithms in search optimization and machine learning*. Reading, MA: Addison-Wesley.

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).

Goldberg, D. E., & Richardson, J. J. (1987). Genetic algorithms with sharing for multimodal function optimization. *Proceedings of the Second International Conference on Genetic Algorithms*, 41–49.

Grefenstette, J. J., & Baker, J. E. (1989). How genetic algorithms work: A critical look at implicit parallelism. *Proceedings of the Third International Conference on Genetic Algorithms*, 20–27.

Harik, G. R. (1995). Finding multimodal solutions using restricted tournament selection. *Proceedings of the Sixth International Conference on Genetic Algorithms*, 24–31. (Also IlliGAL Report No. 94002).

Mahfoud, S. W. (1992). Crowding and preselection revisited. *Parallel Problem Solving from Nature*, *2*, 27–36. (Also IlliGAL Report No. 92004).

Mühlenbein, H., & Schlierkamp-Voosen, D. (1993). Predictive models for the breeder genetic algorithm: I. continous parameter optimization. *Evolutionary Computation*, *1*(1), 25–49.

Nelder, J. A., & Mead, R. (1965). A simplex method for function minimization. *The Computer Journal*, *8*, 308–313.

Press, W., Flannery, B., Teukolsky, S., & Vettering, W. (1989). *Numerical recipes in C*. Cambridge: Cambridge University Press.

Rekalaitis, G., Ravindran, A., & Ragsdell, K. (1983). *Engineering optimization: Methods and applications*. New York, NY: Wiley.

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., & Kendall, G. (2005). Genetic algorithms: A tutorial. In Burke, E., & Kendall, G. (Eds.), *Introductory Tutorials in Optimization, Search, and Decision Support Methodologies* (Chapter 4, pp. 97–125). Berlin: Springer. http://www.asap.cs.nott.ac.uk/publications/pdf/gxk_introsch4.pdf.