Y. Tsoy

EVOLUTIONARY COMPUTATION TOOLKIT LIBRARY ECWORKSHOP (ECW)

Version 0.2

11-25-2007

Development of the ECW is still in progress therefore the structure of the library and some working concepts are subjects to change. For questions, please, mail me to qai@mail.ru

Yury Tsoy

CONTENTS

1. General Description

Title: ECWorkshop

Purpose: Toolkit classes library for evolutionary computation

Programming language: C++

Operating systems: Windows.

WWW: http://qai.narod.ru/ecw

2. Library Structure

General structure scheme of the developed toolkit library is shown in fig. 1. Main modules are [Tsoy, 2007]:

1) Generative module.

2) Evolutionary algorithm (EA).

3) Problem environment.

4) Results processing module.

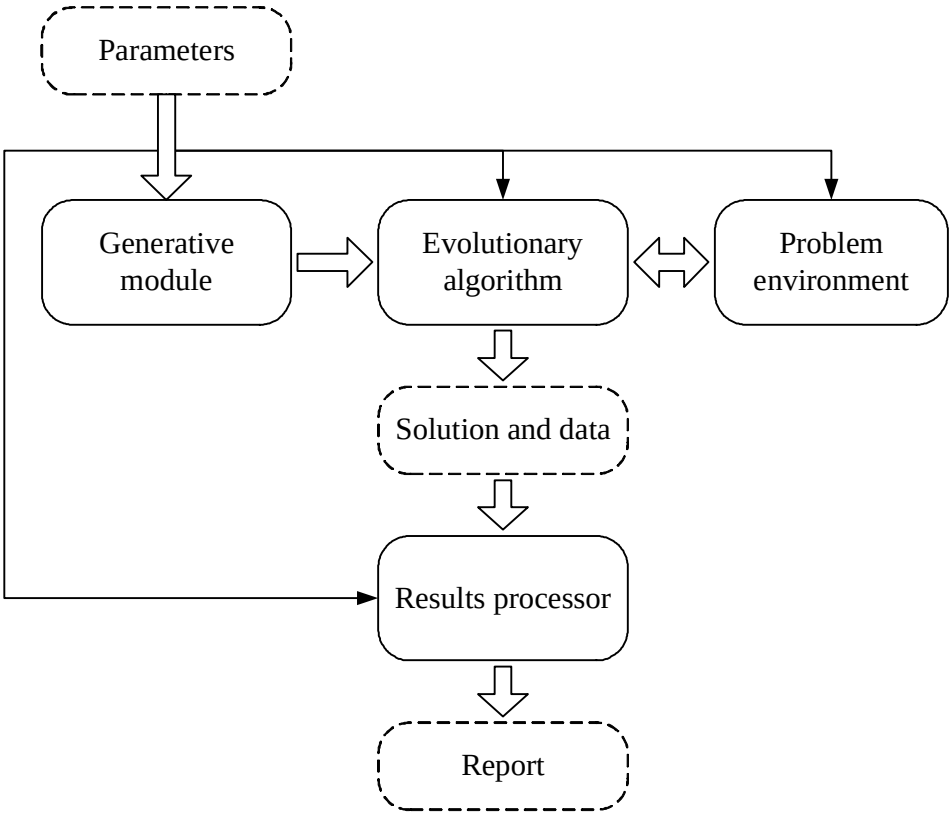

Fig. 1. General scheme of library modules relations

All the input *parameters* are written into the structure `QEAParameters`, which contains fields presented in table 1, that are available for reading for all the library modules and operators (see also Section 7).

The *Generative module* is implemented using the Factory pattern [Gamma et al., 1995] and is used to create all the operators and data structures for EA. Necessity of Factory pattern use arises from the fact that functioning of evolutionary operators under use depend on the selected encoding type. Such an approach allows simplification of EA source code due to involved abstraction from the used encoding and also helps to avoid mismatch between operators and genetically encoded data structures.

Table 1. Description of `QEAParameters` structure

Field name	Field type	Description
`eaType`	`QEAType`	Type of used EA.
`encodingType`	`QEncodingType`	Encoding type.
`permutation`	`bool`	Denotes whether permutation encoding is used.
`order`	`QOrder`	Gene sorting order for permutation encoding.
`chromosomeLength`	`int`	Chromosome length (number of genes).
`arity`	`int`	Power of the alphabet for string encoding.
`geneSize`	`int`	Number of bits per gene.
`geneOffset`, `genePrecision`	`double`, `double`	Step size and offset respectively for gene encoding/decoding operations for integer encoding. The following formula is used: g_{real} = `genePrecision`*g_{int} + `geneOffset` here g_{real} and g_{int} – are real and integer representation of the gene.
`initializationType`	`QInitializationType`	Initialization type.
`unitsRatio`	`double`	Fraction of units for initialization for integer encoding (from 0 to 1)

Field name	Field type	Description
populationSize	int	Initial population size.
minPopSize, maxPopSize	int, int	Minimal and maximal population size respectively.
sizingStep, sizingDirection	int, int	Correspondingly population sizing step and step direction ("1" means increase, "-1" means decrease) for steady population sizing operator.
generationsNumber	int	Number of generations for evolutionary search.
errorLevel	double	Target value of the objective function.
nichingType	QNichingType	Type of niching strategy
sharingSigma	double	Sigma parameter value for the sharing niching strategy
selectionType	QSelectionType	Selection operator type.
tournamentSize	int	Initial tournament size for tournament selection.
eliteCount	int	Number of elite individuals.
parentSelectorType	QParentSelectorType	Type of parents selection operator for single crossing operation.
parentNumber, childrenNumber	int, int	Respectively number of parent individuals participating in single crossing and number of produced offspring.
xType, xRate	QCrossoverType, double	Type and initial rate of crossover operator respectively.
xAlpha	double	Alpha parameter value for BLX and SBX operators
mutationType, mRate	QMutationType, double	Type and initial rate of mutation operator respectively.

Field name	Field type	Description
`populationSizerType`	`QPopulationSizerType`	Type of population sizing operator.
`removeClones`	`bool`	Enables (“true”) removal of duplicate-individuals when next generation population is formed.
`problem`	`QProblem*`	Pointer to the object that present problem environment (is used by EA to evaluate fitness).
`ea`	`QEvolutionaryAlgorithm*`	Pointer to the EA under use
`Inputs`	`int`	Number of inputs for ANN and RN
`outputs`	`int`	Number of ouputs for ANN and RN
`untilFirstHit`	`bool`	Makes EA to stop (if “true”) when the first solution is obtained
`seed`	`unsigned`	Random numbers generator seed.

Produced by generative module data structures and operators are used in the module implementing *evolutionary algorithm* to present evolutionary search. Fitness evaluation of individuals is made by calling the *problem environment* module from the evolutionary algorithm module. With this call the pointer to the individual being evaluated is passed.

Multiple runs of EA can be made within evolutionary algorithm module. Both produced temporal and resulting data are stored inside the structure `QRunSummary` presented in table 2. Note that contents of this structure are available for reading to all the evolutionary operators.

Table 2. Description of structure `QRunSummary`

Field name	Field type	Description
`generation`	`int`	Current generation number.
`feCount`	`int`	Number of fitness evaluation calls

Field name	Field type	Description
		made from the EA beginning.
means, devs	vector<double>, vector<double>	Correspondingly arrays to store mean fitness and fitness deviation dynamics for every generation.
bests, worsts	vector<double>, vector<double>	Correspondingly arrays to store best and worst fitness dynamics for every generation.
popSizes	vector<int>	Array to store population size dynamics information.
startClock, finishClock, firstHitClock	clock_t, clock_t, clock_t	Correspondingly time of EA's start, EA's finish and the time when the first solution was found.
time, firstHitTime	double, double	Correspondingly time of EA run and time necessary to find the first solution in seconds.
firstHitFECount, firstHitGeneration	int, int	Correspondingly number of fitness evaluation calls and number of generations necessary to find the first solution.
firstHitSolution	QIndividual*	First solution found during the EA run
convergenceGeneration	int	Number of generation when population convergence was occurred.
currentBest	QIndividual*	The best individual found so far.

Obtained in result of evolutionary algorithm module multiple runs structures of `QRunSummary` type are processed inside the *processing module* performing the primary statistical calculations to make further comparison of EAs with different parameters setting possible.

3. Data Structures Description

Table 3 contains information about abstract (base) classes for data structures for genetic encoding used in the library.

Table 3. Description of base classes for genetic encoding

Class name	Brief description
QGene	Base class for single gene information.
QIndividual	Base classes for individual. Acts as a container for QGene class objects and contains methods for reading/writing of individual fitness and also additional methods to handle array of genes and the method for individual information printing into the prescribed output stream.
QPopulation	Base class for population. Acts as a container for QIndividual class objects and contains adiditonal methods to handle array of individuals and the method for population information printing into the prescribed output stream.

Definition of new encodings is made by creation of derivative classes from QGene, QIndividual and QPopulation classes. To simplify the addition of the new encoding the template classes are presented (see "template.txt" file in the ECW sources root). Below the template declaration for the new encoding gene class is shown. Square brackets "[]" denote source parts that should be replaced.

```
class QSomeGene : public QGene {
protected:
     [type] _value;
public:
     QSomeGene (void);
     QSomeGene (const QSomeGene& argGene);
     virtual ~QSomeGene (void);

     virtual QSomeGene& operator= (const QSomeGene& argGene);
     virtual QGene* clone () const;
     virtual QEncodingType getEncodingType () const;
     virtual bool equalsTo (const QGene* argGene) const;
     virtual int assign (const QGene* argGene);

     virtual [type] getValue () const;
     virtual int setValue (const [type] argValue);
};
```

Similar template classes are used for new encoding individual and population classes. All encoding classes should overload method getEncodingType which returns type of encoding defined inside the QEncodingType enumeration and method clone to create new instance of the object with the same encoding type that is used by the called object.

Classes for individual's representation should also overload toVectorDouble and toVectorInt methods that are used during fitness calculation to handle different encodings.

These methods are used to convert genetic representation into array of doubles or ints for numerical optimization problems. If the encoding can not be converted into such arrays (the example is graph encoding) the overloaded methods should return `Q_FAILURE`.

To store data about EA run parameters and run results structures `QEAParameters` and `QRunSummary` described in tables 1 and 2 respectively are used. Filling of the most fields of the `QRunSummary` structure is performed during EA run after individuals' evaluation stage (see also Section 6).

The following encoding types are supported:

- Integer encoding.
- Real-parameters encoding.
- String encoding (ordered).
- Permutation encoding (case of {***string***}).
- Regulatory network.

4. Evolutionary Operators Classes Description

All the classes that modify and alter population during the evolutionary search process are referred as evolutionary operators classes. These are the following base classes:

- `QInitializer` – base class for population initialization.
- `QSelection` – base class for selection.
- `QParentSelector` – base class for parents selection for crossing.
- `QCrossoverOperator` – base class for crossover operator.
- `QMutationOperator` – base class for mutation operator.
- `QPopulationSizer` – base class for population sizing operator.
- `QNichingOperator` – base class for niching operator.

Note that the list above can be extended by addition of new operators.

The base class for all the operators is abstract class `QOperator` which declaration is presented below:

```
class QOperator {
protected:
     const QEAParameters* _params;

public:
     QOperator (void);
     QOperator (const QOperator& argOperator);
     virtual ~QOperator (void);

     QOperator& operator= (const QOperator& argOperator);
```

```
    virtual int operate () = 0;
    virtual int setParameters (const QEAParameters& argParams);
};
```

The main method of the operator class is the `operate` method in which operator's functioning peculiarities should be implemented.

Use of the most operators listed above is of traditional way. However the functioning of `QPopulationSizer` operator should to be clarified. The operator under consideration is used for resizing of population for the next generation. Since offspring population is formed by crossover operator (`QCrossoverOperator`) then to define the moment when offspring production should be stopped `QPopulationSizer` operator should be used. Basic variant of `QPopulationSizer` operator doesn't change population size leaving it untouched. Such a functionality is implemented inside `QConstPopulationSizer` class.

The example of use/interaction of different operators for genetic algorithm and their relation with data structures is depicted in figure 2 (see also Section 6). "EA Parameters; Results" block contains data about EA run parameters and run results obtained so far.

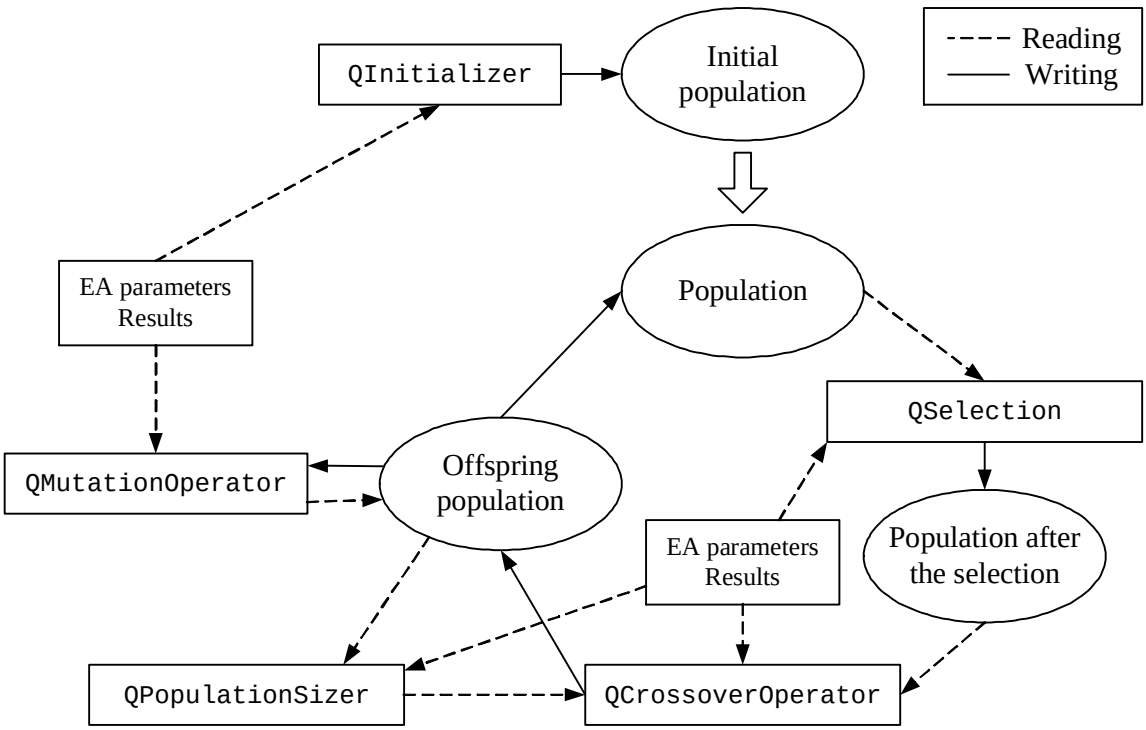

Fig. 2. The example of use of evolutionary operators inside the genetic algorithm. Dashed lines show reading operations while solid lines show writing operations. "EA parameters; Results" block is shown twice to simplify the scheme.

To provide modularity of the library the inner logic of the operators functioning doesn't depend on the other operators. In other words functioning of some operator has no influence on

the functioning of another operator (although its influence on overall EA results can not be excluded).

5. Problem Environment Description

To calculate individual's fitness the `QProblem` class is used. Such use of standalone class for fitness calculation allows to separate evolutionary search and problem specific properties [Tsoy, Spitsyn, 2004]. This provides one with possibility to use one and the same set of classes to solve different optimization problems.

To define new fitness function it is necessary to create a class derivative from the `QProblem` class (see table 4) and overload pure virtual methods `getDefaultParameters`, `evaluateIndividual`, `evaluatePopulation` and `printProblemName`. Also `testIndividual` method exists (which is blank by default) for the case when found (candidate-)solutions are need to be tested on some data which differs from that of evaluation (for example, neuroevolution problems). The calculated fitness value is written into the variable `_fitness` inside `QIndividual` class. Availability of different genetic encodings involves necessity to handle different encodings in `evaluateIndividual`, `evaluatePopulation` and `testIndividual` methods.

The `getDefaultParameters` method is useful when there is a need to set some EA's parameters that depend on the problem and that are supposed to be unknown a priory (file/keyboard input etc.). In most cases just leave this function blank.

Table 4. Brief description of the `QProblem` class

Variable name	**Description**
`_params`	Pointer to the `QEAParameters` structure containing information about current EA run parameters.
Method name	**Description**
`getDefaultParameters`	Sets some fields of <***_params***> to default values according to the specific problem at hand.
`setParameters`	Transfers pointer to the `QEAParameters` structure with information about current EA run parameters into `QProblem` class.
`evaluateIndividual`	Method to calculate individual's fitness.
`evaluatePopulation`	Method to calculate fitness for every individual in population.
`testIndividual`	Method to test obtained solution (in case when evaluation and

	testing are different, which is common for ANNs).
`printProblemName`	Method to print problem name into prescribed output stream.

Note that current library implementation considers fitness *minimization* problem only therefore individual's fitness should decrease as corresponding solution quality grows.

Below an example for `QOneMaxProblem` class is shown considering well-known One-max problem where the string with maximum number of units is to be found.

```
class QOnemaxProblem : public QProblem {
public:
    QOnemaxProblem (void) {}
    virtual ~QOnemaxProblem (void) {}

    int getDefaultParameters (QEAParameters* argParams) {
        argParams->geneSize = 1;
        argParams->genePrecision = 1;
        argParams->geneOffset = 0;
        return Q_OK;
    }

    virtual int evaluatePopulation (QPopulation* argPopul) {
        int i, popSize = argPopul->getSize ();
        for (i=0; i<popSize; i++) {
            evaluateIndividual (argPopul->getIndividual( i ));
        }
        return Q_OK;
    }

    virtual int evaluateIndividual (QIndividual* argInd) {
        vector<int> params;
        // convert genetic representation into vector of ints
        if (argInd-> toVectorInt(params)) {
            int nParams = _params->chromosomeLength;
            int er = nParams;
            double x;

            for (int i=0; i<nParams; i++) {
                x = params[i];
                er -= x;
            }
            argInd->setFitness (er);
            return Q_OK;
        } else {
            cerr<<"\n\n[QOneMaxProblem::evaluatePopulation] error:\n"
                <<"\tIncorrect encoding type! (encoding ID = "
                <<argInd->getEncodingType()<<")\n\n";
            return Q_FAILURE;
        }
        return Q_OK;
    }

    virtual int printProblemName (ostream& out) {
        out<<”Onemax problem\n”;
```

```
        return Q_OK;
    }
};
```

6. Evolutionary Algorithm Class Description

For implementation of certain evolutionary search scheme the separate class derived from `QEvolutionaryAlgorithm` class should be used. The `QEvolutionaryAlgorithm` class provides base methods for evolutionary search including population and data structures initialization and also linking operators with correspondent data structures. The sequence of operators use depends on chosen EA and evolutionary search scheme.

Brief description of `QEvolutionaryAlgorithm` class main methods is presented in table 5.

Table 5. Main methods for `QEvolutionaryAlgorithm` class

Method name	Description
`printName`	Prints algorithm name into the specified output stream
`clone`	Makes copy of the existing EA
`setParameters`	Writes run parameters into `QEAParameters` structure and creates operators and containers for data.
`initializeComponents`	Initialization of operators and data structures corresponding to the `QEAParameters` structure contents.
`evaluatePopulation`	Calculation of individuals' fitness and update of `QRunSummary` structure fields.
`selectParentIndividuals`	Selection of individuals by results of fitness calculation.
`selectElite`	Selection of elite individuals and their transfer into next generation.
`crossPopulation`	Crossing of individuals for formation of offspring population. If crossover rate is 0 then offspring population is created from random pairs of individuals chosen by selection.
`mutatePopulation`	Mutation of offspring population.
`nextGeneration`	Proceed to the next generation.
`finishWork`	Reallocation memory for data structures and operators.
`printResults`	Prints algorithm specific run results (if there are any) into the specified output stream. This method is called by the `QSummaryProcessor` object.

Method name	Description
run	Pure virtual method in which evolutionary operators calls should be implemented in definite order for evolutionary search.

Below the variant of implementation (overloading) of method for genetic algorithm `QGeneticAlgorithm` class functioning according to the scheme in figure 2 is presented:

```
#define   Q_FAILURE 0
#define   Q_OK      1

int QGeneticAlgorithm::run () {
    if (_params.generationsNumber > 0) {
        int i;
        QRunSummary summary;

        _summary.reset ();    // initialization of
                              // QRunSummary structure
        _summary.startClock = clock ();

        initializeComponents ();   // EA initialization

        for (i=0; i<_params.generationsNumber; i++) {
            evaluatePopulation ();      // individuals evaluation
            selectParentIndividuals (); // selection
            selectElite ();             // elite individuals
                                        // selection
            crossPopulation ();         // crossing
            mutatePopulation ();        // mutation
            nextGeneration ();          // next generation
            _summary.generation++;
        }
        _summary.finishClock = clock ();

        return Q_OK;
    } else {
        cerr<<"\n\n[QGeneticAlgorithm::run] error:\n"
            <<"\tGenerations number is undefined!\n\n";
        return Q_FAILURE;
    }
}
```

An example of use of `QGeneticAlgorithm` class, overloaded `run` method and also `QSummaryProcessor` class object to process runs results is shown below:

```
int main () {
    QEvolutionaryAlgorithm* ea = new QGeneticAlgorithm;
    QEAParameters params;
```

```
    QProblem* problem;
    QSummaryProcessor sumProcessor;

    /**
    Read run parameters,
    filling of QEAParameters structure
    and QProblem object creation
    */

    // Pass EA run parameters
    problem->getDefaultParameters (&params);
    problem->setParameters (&params);
    ea->setParameters (params);
    sumProcessor.setParameters (params);

    // Organization of 100 independent EA runs
    for (i=0; i<100; i++) {
        ea->run ();      // EA run
        // save run results
        sumProcessor.addSummary (ea->getSummary());
    }
    sumProcessor.operate ();  // results processing

    ea->finishWork ();   // reallocation of data structures
                         // and operators
    // reallocation of EA and problem objects
    delete problem;
    delete ea;

    return 0;
}
```

7. Configuration File Description (config.xml)

EA run parameters setting is performed via the configuration file «`config.xml`». Parameters values are defined inside «`parameter`» tags and should contain two embedded tags «`name`» and «`value`» to define parameter name and its value. Below an example for population size parameter setting is shown:

```
<parameter>
    <name>Population size</name>
    <value>50</value>
</parameter>
```

Parameters names used inside configuration file are presented in table 6 arranged in alphabetical order.

Table 6. Configuration file parameters.

[type] – denotes parameter value type,

{**type**} – denotes encoding type,

<**name**> – denotes configuration parameter name

Parameter name	Value	Description
Arity	[int]	Power of alphabet for {**string**} encoding.
Children number	2	Number of children produced by crossover. *Should be set to 2!*
Chromosome length	[int]	Number of genes inside the chromosome.
Crossover rate	[double]	Crossover rate.
Crossover type	1-point	1-point crossover operator for {**integer**} and {**string**} encodings.
	2-point	2-point crossover operator for {**integer**} and {**string**} encodings.
	uniform	Uniform crossover operator for {**integer**} and {**string**} encodings.
	1-point PGX	1-point per-gene crossover operator for {**integer**} encoding.
	2-point PGX	2-point per-gene crossover operator for {**integer**} encoding.
	arithmetic	Arithmetic crossover operator for {**real**} encoding.
	BLX	BLX crossover operator for {**real**} encoding. *Alpha parameter is set using <**xAlpha**> configuration parameter.*
	SBX	SBX crossover operator for {**real**} encoding. *Alpha parameter is set using <**xAlpha**> configuration parameter.*
	order	Order crossover operator for {**permutation**} encoding.
Elite count	[int]	Number of elite individuals. If 0 then no elitism used.
Encoding	integer	Integer based encoding. *Set encoding parameters using <**Gene size**>, <**Gene offset**> and <**Gene precision**> configuration parameters.*
	real	Real-code encoding.
	string	String based encoding. *Set encoding parameters using <**Arity**> configuration parameter.*
	permutation	Permutation encoding. Based on {**string**} encod-

		ing. *Set encoding parameters using <**Arity**> and <**Encoding order**> configuration parameters.*
	regulatory network	Regulatory network encoding.
Encoding order	no	No ordering for {**permutation**} encoding.
	ascending	Ascending ordering for {**permutation**} encoding.
	descending	Descending ordering for {**permutation**} encoding.
Error level	[double]	Target value of objective function.
Gene size	[int]	Number of bits per gene for {**integer**} encoding.
Gene offset	[double]	Minimal value of the encoded gene for {**integer**} encoding. *See <**Gene precision**> description for details.*
Gene precision	[double]	Gene encoding precision for {**integer**} encoding. Decoded gene value is calculated as follows: $G_d = G_P G_e + G_O$, where G_d and G_e – are decoded (phenotype) and encoded (genotype) gene values respectively; G_P – gene precision value; G_O – gene offset value.
Generations number	[int]	Number of generations for evolutionary search.
Initialization type	random	Random initialization.
	unmixed	Initialization for unmixed population for {**string**} encoding only.
Max population size	[int]	Maximum allowed population size parameter for dynamic population sizing.
Min population size	[int]	Minimum allowed population size parameter for dynamic population sizing.
Mutation rate	[double]	Mutation rate.
Mutation type	bit-flip	Bit-flip mutation for {**integer**} encoding.
	random flip	Random flip mutation for {**string**} encoding.
	gaussian	Gaussian mutation for {**integer**} and {**real**} encoding.
	basic	Basic mutation operator for the **{regulatory network}** encoding
Parents number	2	Number of parent individuals participating in single crossing. *Should be set to 2!*

Parents selection type	random	Type of <***Parents number***> parents selection from the parental subpopulation.
Population size	[int]	Initial population size.
Population sizer type	constant	Constant population size.
	steady	Constantly increasing/decreasing population size. *Set sizing parameters using <**Min population size**>, <**Max population size**>, <**Sizing direction**> and <**Sizing step**> configuration parameters.*
	simple	Simple population resizing mechanism using constant delta. *Set sizing parameters using <**Min population size**>, <**Max population size**> and <**Sizing step**> configuration parameters.*
	simple Fibonacci	Simple population resizing mechanism using Fibonacci sequence defined delta. *Set sizing parameters using <**Min population size**>, <**Max population size**> configuration parameters.*
Remove clones	yes/no	Defines whether to allow duplicates removal (“yes”) or not (“no”).
RNG seed	[int]	Random numbers generator seed. If 0 then seed is defined using system time.
Selection type	neutral	No selection preformed. All the individuals are allowed to cross.
	tournament	Tournament selection. *Set selection parameters using <**Tournament size**> configuration parameter.*
Sizing direction	no	No sizing direction for **steady** <**Population sizer type**>. *Set sizing amount using <**Sizing step**> configuration parameter.*
	up/down	Sizing using constantly increasing (“up”) or decreasing (“down”) population. *Set sizing amount using <**Sizing step**> configuration parameter.*
Sizing step	[int]	Sizing amount used for **steady** and **simple** <**Population sizer type**>.

Tournament size	[int]	Size of the tournament for tournament selection.
Units ratio	[double]	Fraction of units in chromosomes for {**integer**} encoded initialization.
Until first hit	yes/no	Set EA to run until the first solution is found (“yes”) or until the generation limit is reached (“no”)
xAlpha	[double]	Numerical parameter for {**real**} encoding cross-over operators.

Configuration file is divided into section («`section`» tags) but this division is made for convenience and do not affects reading.

Selection of the algorithm to use is performed via «`algorithm`» tag in configuration file. This tag should contain two embedded tags «`name`» and «`value`» where the «`name`» tag defines algorithm name (acronym is recommended) (see table 7) while the «`value`» tag contents doesn’t matter and this tag should be placed for correct configuration file reading only. Example for the Genetic algorithm selection is shown below:

```
<algorithm>
    <name>GA</name>
    <value>don't care</value>
</algorithm>
```

Table 7. Algorithms naming description

Algorithm’s name (acronym)	**Description**
GA	Genetic algorithm.
SARN	Self-Adaptive Regulatory Network.
SARN2	Self-Adaptive Regulatory Network (2^{nd} variant).

The selection of problem to solve is made using «`problem`» tag in configuration file. This tag should contain two embedded tags «`name`» and «`value`» where the «`name`» tag defines problem name (see table 8) while the «`value`» tag contents doesn’t matter and this tag should be placed for correct configuration file reading only. Example for the Sphere function selection is shown below:

```
<problem>
    <name>Sphere</name>
    <value>don't care</value>
</problem>
```

Table 8. Problems naming description

Problem's name	Description
Deceptive	4-ugly deceptive problem by D. Whitley.
Double sin	Sum of two sine waves prediction problem (taken from [Schmidhuber et al., 2005]) (for *SARN* and *SARN2* algorithms only)
Mixing-time	Mixing-time research setting.
Onemax	Onemax (units counting) problem.
Pointless	Pointless problem. Fitness values are assigned at random.
Rastrigin	Rastrigin's function. Multimodal, symmetric.
Rosenbrock	Rosenbrock's function. Multimodal, symmetric, have plateau.
Schwefel	Schwefel's function. Multimodal, asymmetric.
Sphere	Sphere function. Unimodal, easy.
XOR	XOR problem for ANN (for *SARN* and *SARN2* algorithms only)

8. Running EA and getting results

The compiled exe-file runs in console mode. The following command-line format is used:

```
ecw.exe [config-file] [r]
```

Here [`config-file`] is the name of the configuration file ("`config.xml`" is used by default) and [`r`] – is a number of EA's independent runs (default is 1). For example:

```
ecw.exe config_ga.xml 100
```

runs EA for 100 independent runs and makes `ecw` read parameters from the "`config_ga.xml`" configuration-file.

After the run is finished all results that were processed by `QSummaryProcessor` object are written into the "`summary.log`" file. Some additional results that could be written by `QEvolutionaryAlgorithm::printResults` method call are also written into this file.

Found solutions are written into the "`solution.log`" and candidate-solutions are written into the "`candidate.log`".

References

[Gamma et al., 1995] Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. – Addison-Wesley, 1995.

[Schmidhuber et al., 2005] Schmidhuber J., Wierstra D., Gomez F.J. Evolino: Hybrid Neuroevolution / Optimal Linear Search for Sequence Learning // Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI). – Edinburgh, 2005. – P. 853-858.

[Tsoy, Spitsyn, 2004] Tsoy Y.R., Spitsyn V.G. Use of Design Patterns for Design of the Software Environment for Researches in Genetic Algorithms // Proceedings of 8-th Korea-Russia International Symposium on Science and Technology KORUS-2004. – Tomsk, 2004. – Pp. 166-168.

[Tsoy, 2007] Tsoy Y.R. ECWorkshop – A Toolkit Library For Evolutionary Computation // Proceedings of International Conference on Artificial Intelligence Systems (AIS'07). – Moscow: Fizmatlit, 2007. – Pp. 94-101. (in Russian)