The MARLEDA Manual
version 0.8

Matthew Alden
`mealden@uw.edu`

FILES

`examples/*`	Example MARLEDA parameter files.
`src/*`	MARLEDA source files, makefile, and `chi2.dat` (a required data file).
`COPYING`	The GNU Public License (v3).
`README.pdf`	This document.

INSTALLATION

MARLEDA is written in C++ and has been compiled/tested with numerous versions of the GNU C++ compiler through version 4.3.3. The `src` directory contains the source code for the MARLEDA executable as well as a GNU `make` makefile for building the MARLEDA executable. Appropriate `make` targets are `all` and `marleda`.

When executed, MARLEDA requires the included data file `chi2.dat` (in the `src` directory) to be present in the current working directory. Apart from this restriction, the `marleda` executable and `chi2.dat` file may be copied to any location within the file system.

RUNNING MARLEDA

A single invocation of MARLEDA performs a single optimization experiment. The parameters of that experiment may be specified in a parameter file, as command line arguments to the MARLEDA executable, or a combination of both.

Parameters specified on the command line take the form "*`-name value`*". For example,

```
marleda -task onemax -genes 20 -popSize 100 -generations 10
```

Parameter files may contain one or more sets of parameters. A parameter file containing a single set of parameters consists of lines of the form "*`name value`*" with no blank lines in between. Lines beginning with the # character denote comments and are ignored. Parameter values may be surrounded by double quotes. A single set of such quotes will be stripped and not included as part of the value. For example,

```
# all experiment parameters
task onemax
genes 20
popSize 100
generations "10"
```

Parameters are loaded from a parameter file by specifying the file's name on the MARLEDA command line. For example, if the above parameter file was named `OneMax.param`,

```
marleda OneMax.param
```

A parameter file containing multiple sets of parameters must provide a name for each set (placed within `[ ]` brackets) and separate each set with a blank line. For example,

```
[common]
popSize 200
generations 50

[exp1]
task onemax
genes 20

[exp2]
task rosenbrock
genes 32
```

A single set of parameters is loaded from a parameter file by specifying both the file's name and the set name (separated by a colon) on the MARLEDA command line. Reading multiple sets of parameters requires multiple command line arguments, but the parameter file's name can be omitted from subsequent arguments. For example, if the above parameter file was named `experiments.param`,

```
marleda experiments.param:common :exp2
```

Parameters can be loaded from multiple files or specified directly on the command line in any combination. For example,

```
marleda OneMax.param experiments.param:common -genes 150
```

The following table describes the current parameters of the MARLEDA system.

Name	**Description**	**Required**	**Default**
`task`	Specifies the optimization problem for the experiment. Included optimization problems are `onemax`, `trapmax`, `uniform`, `rosenbrock`, and `spin`.	✓	
`genes`	Specifies the number of genes in each chromosome for the duration of the experiment. Individual optimization problems, such as the included Ising spin glass problem, may set this parameter based on other problem-specific parameters, thus this parameter may not always need to be specified explicitly.	✓	
`popSize`	Specifies the number of chromosomes in the population (steady-state).	✓	
`popFile`	Specifies a file from which to load the initial population for the experiment. If this parameter is not specified, a random initial population will be generated. If this parameter is specified, the `genes` and `popSize` parameters will be automatically determined from the initial population, and therefore shouldn't be specified explicitly.		
`modelFile`	Specifies a file from which to load the initial Markov random field neighborhood system. If this parameter is not specified, a trivial (empty) initial neighborhood system will be generated.		

Name	**Description**	**Required**	**Default**
`generations`	Specifies the duration of the experiment in terms of number of population generations. Only one of `generations` or `evaluations` is required.	✓	
`evaluations`	Specifies the duration of the experiment in terms of fitness function evaluations. The actual number of fitness function evaluations performed may exceed this limit in order to evaluate all new chromosomes within the final population. Only one of `generations` or `evaluations` is required.	✓	
`selectionBasis`	Specifies the proportion of high-fitness chromosomes within the population that will contribute to the next generation.		1.0
`tournamentSize`	Specifies the number of chromosomes that compete in a single round of tournament selection. If the value of this parameter is less than 2, tournament selection is disabled.	✓	
`mutation`	Specifies the per-gene probability of mutation.		0.0
`elitists`	Specifies the proportion of high-fitness chromosomes that will be directly copied to the next generation.		0.0
`modelAddTries`	Specifies the number of randomly selected non-neighbor gene pairs to test for neighbor status in the Markov random field neighborhood system.	✓	
`modelAddThresh`	Specifies the minimum confidence level necessary to change the status of two genes from non-neighbors to neighbors within the Markov random field neighborhood system.	✓	
`modelSubTries`	Specifies the number of randomly selected neighbor gene pairs to test for non-neighbor status in the Markov random field neighborhood system.	✓	
`modelSubThresh`	Specifies the maximum confidence level necessary to change the status of two genes from neighbors to non-neighbors within the Markov random field neighborhood system.	✓	
`monteIters`	Specifies the number of iterations conducted in the Markov chain Monte Carlo process used to generate new chromosomes.	✓	
`snapshot`	Specifies the location and/or prefix for saving population/model files during the course of the experiment. The file name for population/model snapshots is generated by appending the generation number and extension `pop` to the value of this parameter. If this parameter is not specified, no population snapshots will be saved.		
`snapshotMin`	Specifies the first generation for which a population snapshot will be saved. If `snapshotMin` is less than zero, then a snapshot will only be taken of the final generation, provided the `snapshot` parameter is also defined.		-1
`snapshotFreq`	Specifies the frequency (in generations) of population snapshots. If population snapshots are enabled, the final population of the experiment will always be saved, regardless of the values of `snapshotMin` and `snapshotFreq`.		1
`fitnessLog`	Specifies the location for saving a summary of population statistics calculated every generation. The special file name ”`-`” may be used to specify the standard output stream.		
`rngSeed`	Specifies a seed value for the random number generator used by MARLEDA. If this parameter is not specified, the random number generator is seeded from the local system time.		

Individual optimization problems may utilize additional parameters. For the included TrapMax problem,

Name	**Description**	**Required**	**Default**
`trapSize`	Specifies the number of bits in each trap. The value of this parameter should evenly divide `genes`.	✓	

For the included uniform strings problem,

Name	**Description**	**Required**	**Default**
`alleles`	Specifies the size of the alphabet strings are drawn from.	✓	

For the included Ising spin glass problem,

Name	**Description**	**Required**	**Default**
`spinFile`	Specifies the file from which coupling coefficients are read. The `genes` parameters is automatically set based on the size of the spin glass.	✓	

ADDING OPTIMIZATION PROBLEMS

MARLEDA represents optimization problems using C++ classes that inherit from the provided class `EvolutionTask`. When an optimization experiment is performed, a single object (of the appropriate class) is used to evaluate all chromosomes generated in the course of the experiment. Included in this distribution are five common optimization problems, OneMax, TrapMax, uniform strings, the 2D Rosenbrock function, and Ising spin glasses, to serve as guides for creating new optimization problem classes.

An optimization problem class encapsulates all domain specific aspects of the problem:

- The set of alleles (gene values) appropriate for the problem.
- Instance parameters (if applicable), e.g. spin glass coupling coefficients.
- A fitness function for evaluating chromosomes.

All data (alleles and parameters) must be available via class fields.

An optimization problem class must provide three methods, declared in the in the `EvolutionTask` class:

- `bool initAlleles(const ParameterSet& ps, AllelePool& ap)`
- `bool init(ParameterSet& ps, const AllelePool& ap)`
- `double evaluate(const Chromosome& chromosome)`

The first two methods are used to initialize the optimization problem object before the optimization experiment begins, while the last method is the fitness function used during an experiment.

The `initAlleles` method is the first initialization method called. This method should add alleles appropriate for the optimization problem to the passed `AllelePool` object. The `AllelePool` object manages the mapping from the internal representation of an allele (an integer) to the serializable form used when reading or writing population files (a string). `Note`: due to current constraints on the serialization process, alleles should not contain white space. The `AllelePool` object also conveys the set of legal alleles to the rest of the MARLEDA system. For example, an optimization problem over binary chromosomes such as OneMax would add alleles for `0` and `1`.

```
class OneMax : public EvolutionTask {
public:
        bool initAlleles(const ParameterSet& ps, AllelePool& ap) {
                ap.newAllele("0");
                ap.newAllele("1");
                return true;
        }
        ...
```

The `initAlleles` method has access to the full set of parameters passed to the MARLEDA invocation (via the command line or parameter files), should those parameters affect the set of legal alleles.

The `init` method is called after the `initAlleles` method and should initialize the optimization problem object's internal storage of the legal alleles plus any instance data. Continuing the example of OneMax, the `OneMax` class requires only two allele fields:

```
class OneMax : public EvolutionTask {
private:
        Allele OFF, ON;

public:
        bool initAlleles(const ParameterSet& ps, AllelePool& ap) {
                ap.newAllele("0");
                ap.newAllele("1");
                return true;
        }

        bool init(ParameterSet& ps, const AllelePool& ap) {
                try {
                        OFF = ap.get("0");
                        ON = ap.get("1");
                } catch (invalid_argument& ex) {
                        return false;
                }
                return true;
        }
        ...
```

The `init` method may modify the set of global parameters used by the MARLEDA system. This is useful in situations where MARLEDA parameters depend on instance parameters, e.g. setting the MARLEDA parameter `genes` parameter to match the size of a spin glass read from a file (see the `SpinGlass` class).

Lastly, the `evaluate` method fills the role of fitness function. The return value indicates the quality of the evaluated chromosome, which MARLEDA seeks to maximize. Fitness scores are only used to rank chromosomes, thus the specific range of fitness scores for an experiment will not affect MARLEDA's performance. Finishing the OneMax example:

```
class OneMax : public EvolutionTask {
private:
        Allele OFF, ON;

public:
        bool initAlleles(const ParameterSet& ps, AllelePool& ap) {
                ap.newAllele("0");
                ap.newAllele("1");
                return true;
        }

        bool init(ParameterSet& ps, const AllelePool& ap) {
                try {
                        OFF = ap.get("0");
                        ON = ap.get("1");
                } catch (invalid_argument& ex) {
                        return false;
                }
                return true;
        }

        double evaluate(const Chromosome& c) {
                double sum = 0;
                for (size_t i = 0; i < c.size(); i++)
                        if (c[i] == ON)
                                sum++;
                return sum;
        }
}
```

For convenience, new optimization problem classes for problems over binary chromosomes may inherit from the provided class `BinaryEvolutionTask` instead of class `EvolutionTask`. The `BinaryEvolutionTask` class supplies allele fields and implementations of the `initAlleles` and `init` methods appropriate for binary chromosomes (as shown in the OneMax example). The `OneMax` class above can be simplified by utilizing the `BinaryEvolutionTask` class as follows:

```
class OneMax : public BinaryEvolutionTask {
public:
        double evaluate(const Chromosome& c) {
                double sum = 0;
                for (size_t i = 0; i < c.size(); i++)
                        if (c[i] == ON)
                                sum++;
                return sum;
        }
}
```

To help support environments where fitness evaluations could be more efficiently performed in batches, the `EvolutionTask` class provides an additional method to act as a hook for custom evaluation of a set of chromosomes. The `evaluate(Population` `pop)` method is called to evaluate the entire set of new chromosomes created every generation. This method may be overridden to, for example, serialize the chromosomes and evaluate them on a distributed computing cluster.

To fully integrate a new optimization problem into the MARLEDA executable, the problem must be selectable via the MARLEDA parameter `task`. In the MARLEDA source file `main.cc`, the function `selectTask` controls the mapping between `task` arguments and optimization problem objects. This function should be augmented to return an object of the new optimization problem class when an appropriate `task` argument is specified.