

mPOEMS in JAVA

Thomas Kremmel

September 18, 2008

Abstract

This paper is the documentation of the generic JAVA implementation of an evolutionary algorithm called *Multiobjective Prototype Optimization with Evolved Improvement Steps* (mPOEMS). The framework was designed to provide optimisation problem engineers with an interface to use mPOEMS, without detailed understanding of the algorithm. All needed methods and fields to use the framework are presented.

The *mPOEMS in JAVA* framework is an implementation of the evolutionary algorithm mPOEMS, presented in [2]. Main goal of the framework implementation was the creation of a generic framework, for which problem-dependent parts could be easily attached. The framework was created in close collaboration with Jiri Kubalik ¹, the inventor of mPOEMS, under the observation of Stefan Biff ².

Problem engineers have to know only three classes of the mPOEMS in JAVA framework, to be able to work with it. Figure 1 shows these three classes, whereas these classes serve as the interface of the framework. The diagram shows only the methods and fields, which a developer will need to work with the framework.

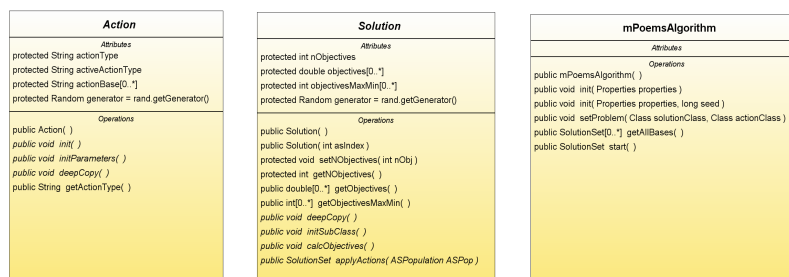


Figure 1: mPOEMS in JAVA Interface

Please note that in the interface diagram, abstract classes and methods are written in italic font style. The implementation of the abstract *Action* and

¹Department of Cybernetics, Czech Technical University in Prague

²Institute of Software Technology and Interactive Systems, Vienna University of Technology

Solution classes require the greatest part of the implementation effort, whereas the *mPOEMSAlgorithm* class serves as the main class to start, initialise and communicate with the framework.

mPOEMS uses so called actions to modify a selected prototype, and consequently create new solutions, which are merged with the initial solution base, resulting in a solution base with the best solutions from the initial and the created solutions. To merge these differing sets of solutions, a method based on the non-domination concept and the crowding distance, which is as well used in NSGA-II [1], is utilised. Solutions are complete encodings of the problem at hand, presenting, for example, the selected knapsack items in the knapsack problem, together with their calculated objective values. Further details of the algorithm, are presented in [2].

In order to use the mPOEMS in JAVA framework, one has to implement two classes, which are derived from the abstract solution and action classes, found in the mPOEMS in JAVA framework.

The following sections presents the available methods and fields from the abstract classes, which may and has to be used by the derived classes, like as well the methods which has to be called to start and communicate with the framework.

1 Action Class

This section presents the fields and methods of the abstract action class.

1.1 Fields

mPOEMS.Action.actionBase is a String array which holds the identifier of the actions used in the optimisation problem. This field has to be initialised in the `mPOEMS.Action.init()` method. Please note that the first action to defined in the actionbase has to be the NOP action. Thus `actionBase[0]` is String NOP.

mPOEMS.Action.actionType is a String which defines the actual type of the action. Actions may change their type in the evolutionary algorithm from an active action type, to a so called NOP (no operation) action. Each action has certain parameters, and modify a solution in a certain, problem dependent way. In the evolutionary algorithm a set of actions is applied to a solution, whereas such sets are called chromosomes. In order to implement a variable length of these chromosomes, called action sequences in terms of mPOEMS, NOP actions are used. In order to remember, to which action type an action was set in the initialisation of the object, the `mPOEMS.Action.activeActionType` is used.

mPOEMS.Action.activeActionType is a String, used to remember the first setting of the active action type. This is important, since tests have shown that mPOEMS performs better, when actions are reset to their initial action type, instead of choosing a new actionType by chance. Problem engineers do only use this variable, in setting or resetting the parameters, based on the activeActionType. The activeActionType is set by the framework, and must not be changed in the problem dependent part of the optimisation implementation.

mPOEMS.Action.generator is a java.util.Random RandomGenerator provided from the mPOEMS framework. This generator has to be used, if the problem at hand requires some created random values. The initialisation of the mPOEMS framework with a random seed, and the use of the same generator in all derived and non-derived classes, ensure the creation of identical results in different runs, given the same seed and same problem instance was used.

1.2 Methods

mPOEMS.Action.Action() is the constructor of the super class, which initialise an Action object, and in turn will call the init() method. This super class constructor has to be called out of the constructor of the derived class, with the command *super()*;

mPOEMS.Action.init() initialise the action object. The String array mPOEMS.Action.actionBase must be created, with all available action IDs, including the NOP ID. Other problem dependent objects can be initialised in this method. Probably each optimising problem has the need for a parameter object, which may be created in this method.

mPOEMS.Action.initParameters() is used to initialise the parameter values, dependent on the activeActionType. One can rely that each time when this method is called, the activeActionType is already set by the mPOEMS framework. Parameters could be set by random, or following some problem dependent procedure.

mPOEMS.Action.deepCopy() cuts all references to problem dependent objects, by replacing them with new objects, including the same values. The method is called automatically by the mPOEMS framework, and ensures that no action object will reference to the same object. This is especially important, since actions undergo a lot of modifications and mutations.

mPOEMS.Action.getActionType() returns the actionType of an action object.

2 Solution Class

This section presents the fields and methods of the abstract solution class.

2.1 Fields

mPOEMS.Solution.nObjectives is an int value, indicating the number of objectives to optimise in the optimisation problem. The number of objectives is set in the configuration file of the framework. The framework is using the configuration, and automatically sets the nObjectives value of each solution to the value stated in the configuration file.

mPOEMS.Solution.objectives is a double array, which holds the calculated objectives for the solution. Based on the solutions objectives, the quality of the solution is determined.

mPOEMS.Solution.objectivesMaxMin is an int array, of length nObjectives. Each entry states whether an objective is to be maximised, or to be minimised, whereas value 1 means the objective has to be maximised, and value 0 means the objective has to be minimised. The optimisation method for the objectives is declared in the configuration file.

mPOEMS.Solution.generator has the same purpose as mPOEMS.Action.generator. Please refer to this section for more detail.

2.2 Methods

mPOEMS.Solution.Solution() is the constructor of the super class.

mPOEMS.Solution.Solution(int asIndex) is the constructor of the super class, which takes the argument asIndex. asIndex is the index of the applied action sequence. This method is mainly for testing / track back purposes. Most developers have to use the mPOEMS.Solution.Solution() constructor.

mPOEMS.Solution.setNObjectives(int nObj) is used to set the number of objectives in a created solution. The mPOEMS framework select a prototype to be modified by the actions. The modification results in the creation of new, hopefully better, solutions. This solutions have to be initialised by setting the solution values to the same values like the prototype solution. Setting the number of objectives is one of these values.

mPOEMS.Solution.getNObjectives() returns the number of objectives to be optimised.

mPOEMS.Solution.getObjectives() returns the objectives array of the solution.

mPOEMS.Solution.getObjectivesMaxMin() returns the objectivesMaxMin array, which states whether an objective is to be maximised, or to be minimised.

mPOEMS.Solution.deepCopy() is an abstract method which has to be implemented in a deriving class. It serve the same purpose as mPOEMS.Action.deepCopy(). Please refer to this section for more details.

mPOEMS.Solution.initSubClass() is called automatically by the framework, and initialise all problem dependent variables. Thus at least the solution representation has to be initialised. This can be done by change, or following some heuristic. The solution representation is in the case of the knapsack problem, the vector with the length of available items, where each index of the vector can be set to zero or one, to indicate whether an item is selected or not.

mPOEMS.Solution.calcObjectives() is on the one hand called automatically by the framework in creating the first solution base, and on the other hand called manually in the mPOEMS.Solution.applyActions(ASPopulation ASPop) method. The calcObjectives() method calculates the objectives, based on the current setting of the solution representation, while adhering to all problem dependent restrictions. Thus this method is as well the starting point to call repair algorithms, to check all restrictions, and in the end calculate the objectives. In the knapsack problem, the calcObjectives() method check whether capacity constraints are violated, remove items based on their weight/item ratio, and calculates for the valid solution representation the profit for both knapsack.

mPOEMS.Solution.applyActions(ASPopulation ASPop) applies the actions in the argument ASPop to the solution. This method is called automatically by the framework. The method is called each time when a prototype is selected, and new solutions are created based on the application of actions to the prototype. Based on the actionType, the corresponding parameters, the clones of the prototype are modified, and objectives re-calculated. Each created solution is saved in a created mPOEMS.SolutionSet and will be returned consequently.

After presenting the methods and variables of the abstract classes, the starting interface for the mPOEMS framework is presented in the next section.

3 mPOEMSAlgorithm Class

The mPOEMSAlgorithm class provides the methods to start the mPOEMS in JAVA framework. The methods have to be called in the given order.

mPOEMS.mPoemsAlgorithm.mPoemsAlgorithm() is the constructor of the mPoemsAlgorithm class.

mPOEMS.mPoemsAlgorithm.setProblem(Class solutionClass, Class actionClass) sets the problem dependent classes. This is necessary in order that the algorithm is able to work and initialise this classes.

mPOEMS.mPoemsAlgorithm.init(Properties properties) initialise the algorithm with the given properties. The java.util.Properties argument properties, has to include all configuration values, presented in section *Configuration*. This configuration file has to include as well a seed value.

mPOEMS.mPoemsAlgorithm.init(Properties properties, long seed) is an alternative to the mPOEMS.mPoemsAlgorithm.init(Properties properties) method. It basically serves the same purpose, but does separately provide the user with the ability to provide a specific seed. This is especially useful if a certain number of subsequent runs with different seeds should be conducted with a batch file, without re-writing the properties file for each run.

mPOEMS.mPoemsAlgorithm.start() does start the mPOEMS algorithm, with the given properties, for the given problem. The method returns a mPOEMS.SolutionSet, which contains the final set of solutions. This set of solutions may be used for further analysis.

mPOEMS.mPoemsAlgorithm.getAllBases() returns all solutions in an array of mPOEMS.SolutionSet, which were kept for further analysis. Which sets have to be kept, is configured in the configuration value *analysisIterations*.

4 Configuration

This framework can be configured with various configuration values. The configuration is separated into three main areas. These areas are *mPOEMS Framework*, *MOEA*, *Optimisation Problem*, and *Analysis*.

4.1 mPOEMS Framework

The configuration values for the mPOEMS framework are presented subsequently. Table 1 shows the basic working principle of mPOEMS. The configuration values presented below, refer to some parts of this outline.

nIterations is the termination condition of mPOEMS, found at line seven in the mPOEMS outline . This configuration value defines the number of iterations conducted by the mPOEMS framework.

```

1  generate(SolutionBase)
2  repeat
3    Prototype ← choosePrototype(SolutionBase)
4    ActionSequences ← MOEA(Prototype,SolutionBase)
5    NewSolutions ← applyTo(ActionSequences,Prototype)
6    SolutionBase ← merge(NewSolutions,SolutionBase)
7  until(termination condition is fulfilled )
8  return(SolutionBase)

```

Table 1: Outline of the mPOEMS Framework

sbSize defines the size of the solution base.

pActive defines the percentage of actions, which will be initialised as active actions. Thus this configuration value set to 75, would lead to 25 percent non-active actions in a mPOEMS run.

seed defines the seed configuration value for the random number generator. The use of a seed value ensures exactly the same sequence of random values, and therefore to the creation of exactly the same results in different runs, given that the configuration values are not changed.

4.2 MOEA

The multi-objective evolutionary algorithm (MOEA), used in the mPOEMS framework, is outlined in table 2. It can be configured with the following configuration values.

Input: Prototype, SolutionBase
Output: Population of Evolved ActionSequences

```

1  generate(OldPop)
2  evaluate(OldPop)
3  repeat
4    NewPop ← evolutionaryCycle(OldPop)
5    evaluate(NewPop)
6    OldPop ← merge(OldPop,NewPop)
7  until(EA termination condition is fulfilled )
8  return(oldPop)

```

Table 2: Outline of the MOEA used in MPOEMS

nGenerations is the MOEA termination condition, found on line seven in the outline of the MOEA.

popSize defines the size of the population of ActionSequences, created in step one of the MOEA.

maxGenes defines the number of actions included in one ActionSequence.

pM defines the probability of mutation for each ActionSequence, selected by Tournament Selection.

pC defines the probability of crossover for the ActionSequences, selected by Tournament Selection.

pBitflip defines the probability of mutation for each Action of an Action-Sequence, which was selected for mutation.

nTournament defines the number of candidates, which take part in the Tournament Selection.

4.3 Optimisation Problem

The configuration values for the problem to solve are presented in this section.

nObjectives defines the number of objectives for the optimisation problem.

ObjectiveN defines whether an objective has to be maximised or minimised. N has to be exchanged with the number of the objective. For each objective a configuration value has to be defined. Therefore N ranges from one till $nObjectives$.

4.4 Analysis

Some configuration values are available to configure the output of the framework.

analysisIterations defines the SolutionBases which should be returned by the `mPOEMSAAlgorithm.start()` method. This is done by calculating number of actual iteration modulo this configuration value. Thus if this configuration value is set to fifteen, each fifteenth SolutionBase is saved. This configuration value is mainly for testing purposes. Especially to observe how the SolutionBase evolve over time. If only the last SolutionBase should be saved, set the configurationValue to the same value, like the *nIterations* configuration value. Besides the SolutionSets selected by this configuration value, the randomly generated, the first and the last SolutionBase will be returned. Attention should be given

to a possible memory overflow, if a great number of SolutionBases are kept for further analysis.

analysisCoverageMetric defines whether the saved SolutionSets should be analysed using the coverage metric. Setting this configuration value to 1 means, the analysis should be conducted. 0 means the analysis should not be conducted. This metric could be used to observe whether your SolutionBase evolve over time, respectively how fast it converges. The results are printed out to the standard output, or to the log file defined in the log4J.xml. The analysis compares each saved SolutionSet with the final SolutionBase.

5 Conclusion

mPOEMS in JAVA provides an interface to use the state-of-the-art evolutionary algorithm mPOEMS, without the need to know the algorithm in detail. It is relatively easy to use, and ensures fast creation of the problem dependent part of an multi-objective optimisation problem.

This framework is free to use for education and research purposes. If you use the framework for academic purposes, please cite my work. For use which is beyond academic purposes, please refer to the licence, which can be found at <http://www.thomaskremmel.com/mpoems/mpoems-licence.html>.

References

- [1] Deb K., Pratap A., Agarwal S., Meyarivan T., *A Fast and Elitist MultiObjective Genetic Algorithm: NSGA-II*. in Journal IEEE Trans Evol Computat, Volume 6, pp. 182–197, 2002.
- [2] Kubalik J., Mordinyi R. and Biffi S., *Multiobjective Prototype Optimization with Evolved Improvement Steps*,. Lecture Notes in Computer Science, Springer, 2008.