

CICC Framework User Manual

Justin Maltese

January 10, 2016

Contents

1	Overview	1
1.1	Motivation	1
1.2	Feature Summary	2
2	Installation	2
3	Architecture	3
3.1	Basic Components	3
3.2	XML Schemas	3
3.3	Fitness Evaluation	4
3.4	Codeword Compatibility	5
3.4.1	Generating Compatibility Matrices	5
4	Experimentation	6
4.1	Graphical User Interface	6
4.2	ExecutionHandler	11
5	FAQ	13

1 Overview

1.1 Motivation

The Computational Intelligence for Covering Codes (CICC) framework is a tool for developers to implement custom computational intelligence algorithms for the purpose of finding minimal covering codes. Currently, no frameworks exist for this purpose, thus the primary motivation for the CICC framework is to fill this gap. At its core, the CICC framework serves to compare all types of computational intelligence algorithms on minimal covering problems, running comparisons as quick as possible and saving results in an easy-to-read format. Algorithm implementation aims to be simple, representing a plug-and-play style framework. It is our

hope that developers can utilize the CICC framework to avoid writing boilerplate code, instead focussing on implementing creative new computational intelligence algorithms.

1.2 Feature Summary

The CICC framework provides a variety of different features, which include the following:

- Sleek graphical interface for users to select algorithms and minimal covering code problems for comparison purposes in a step-by-step fashion
- Fast, easy comparison of algorithms on the minimal covering code problem
- Live updates of algorithm performance
- Automatic saving of best-ever-found minimal covering codes. Upon finding a new best-ever code for a problem, the code is written out as a .code file to the *Codes* folder.
- Ability to easily implement new CI algorithms along with respective parameter sets in a plug-and-play fashion
- Developers can setup algorithm comparisons directly in code rather than using a graphical interface
- Ability to automatically write comparison results into nicely-formatted LaTeX tables

2 Installation

Upon downloading the executable version of the CICC framework, the package should contain the files shown below:

Name	Date modified	Type
AlgorithmSchemas	10/01/2016 3:49 PM	File folder
Codes	10/01/2016 3:49 PM	File folder
Compatibility	10/01/2016 3:57 PM	File folder
Latex	10/01/2016 3:57 PM	File folder
CICCFrameworkInterface.jar	10/01/2016 4:05 PM	Executable Jar File
matrixGenerator.bat	10/01/2016 4:02 PM	Windows Batch File
MatrixGeneratorInterface.jar	10/01/2016 3:48 PM	Executable Jar File
startup.bat	10/01/2016 4:00 PM	Windows Batch File

To run the graphical user interface, simply execute the **startup.bat** file. Note that if either the CICCFrameworkInterface.jar file or the AlgorithmSchemas folder is missing, the program will not execute. Please ensure that both of these items are present.

3 Architecture

3.1 Basic Components

The basic components of the CICC framework are **solutions**, **algorithms** and **problems**. Within the CICC framework, all algorithms evolve small “seeds” which are fed into a greedy algorithm to produce full covering codes. This has the advantage of keeping candidate solution representation small and compact, minimizing epistasis. Each solution represents a “seed”, possessing a set of decision variables which are analogous to codewords present in the seed. All codewords are represented as integers, stored in memory as the corresponding base-2 representation.

Algorithms within the CICC framework extend the `Algorithm` class. Each algorithm typically possesses a set of solutions, formally referred to as a `SolutionSet`, which are modified in some way during the execution of the algorithm. All operations performed by the algorithm should be placed into a method with the signature `public Solution execute()`. This method should return the overall best solution found by the algorithm during execution. Algorithms can print an output message using the `println(String)` method.

Problems represent minimal covering code problems, formally referred to as $K_q(n, r)$. Thus, a problem requires a value for q , n and r . Additionally, since all algorithms within the CICC framework are *greedy closure algorithms*, each problem requires a *seed_size* integer parameter which corresponds to the size of the seeds (candidate solutions) evolved by each algorithm.

3.2 XML Schemas

When defining a new algorithm in the CICC framework, an XML file must be provided for the algorithm in the *AlgorithmSchemas* folder which defines its properties and parameters. The root element should be

algorithm, possessing the following attributes:

- **id** - abbreviation which uniquely identifies the algorithm
- **name** - full name of the algorithm
- **driverClass** - full path (including package prefixes) to the algorithm class containing the `execute()` method

Nested within the `algorithm` element should be the `parameterSet` element. Each parameter of the algorithm should be given as a `parameter` child element within the `parameterSet` element. Parameters should possess the following attributes:

- **id** - string which uniquely identifies the parameter
- **label** - label of the parameter, displayed to the user via the GUI
- **type** - type of the parameter. Possible values are Integer, Double, Float, Long, Boolean
- **defaultValue** (optional) - default value of the parameter

A sample XML schema for a discrete particle swarm optimization algorithm is provided below.

```
<?xml version="1.0"?>
<algorithm id="DPSO" name="Discrete Particle Swarm Optimization" driverClass="com.ciccoFramework.algorithms.dpspo.DiscretePSOAlgorithm">
  <parameterSet>
    <parameter id="MAX_ITERATIONS" label="Maximum number of iterations" type="Integer" defaultValue="200"/>
    <parameter id="SWARM_SIZE" label="Swarm size" type="Integer" defaultValue="400"/>
    <parameter id="INERTIAL_WEIGHT" label="Inertial weight value" type="Double" defaultValue="0.7"/>
    <parameter id="COGNITIVE_WEIGHT" label="Cognitive weight value" type="Double" defaultValue="1.4"/>
    <parameter id="SOCIAL_WEIGHT" label="Social weight value" type="Double" defaultValue="1.4"/>
    <parameter id="GBEST_MUTATION_PROBABILITY" label="Probability of variable mutating to gBest" type="Double" defaultValue="0.15"/>
    <parameter id="PBEST_MUTATION_PROBABILITY" label="Probability of variable mutating to pBest" type="Double" defaultValue="0.05"/>
  </parameterSet>
</algorithm>
```

3.3 Fitness Evaluation

To ensure a plug-and-play style, fitness evaluation of solutions is extremely easy to perform when implementing an algorithm. Since each algorithm should possess a `SolutionSet` (see Section 3.1) containing the candidate solutions of the algorithm, the `doEvaluation()` method of the solution set can be called to

perform fitness evaluation of all solutions in the set. Note that fitness evaluations are distributed across all available cores to minimize the amount of time taken. Upon completing the `doEvaluation` method, the `fitness` property of each solution will be updated with its corresponding fitness value. Note that solutions which possess invalid codewords (boundary violations) are given a fitness of `Integer.MAX_VALUE`.

3.4 Codeword Compatibility

Computing the compatibility between codewords is an important task in the CICC framework. Since codewords are stored in memory as binary representations, problems possessing $q = 2$ utilize bit operations to swiftly calculate codeword compatibility. When $q > 3$, the integer representation of each word is converted into base q when computing distance. In this case, bit operations cannot be used and thus compatibility calculation is much slower. Consequently word compatibility is not computed at runtime, rather it is precomputed and accessed in $O(1)$ time via a matrix of bits. Such a matrix is deemed a “compatibility matrix”, defined as follows: Let $H_D(x, y)$ represent the Hamming distance between integer words x and y . If the bit at index $i, j = 1$, $H_D(i, j) \leq r$, therefore i covers j and vice-versa. If the index is 0, then neither i or j cover each other and thus are incompatible.

3.4.1 Generating Compatibility Matrices

Problems with $q > 3$ require pre-computed compatibility matrices, stored in `.matrix` files within the *Compatibility* folder. Using this strategy, large values of n and q can be tackled for as many runs as needed, since the matrix only needs to be loaded into memory once for a given problem. The **matrixGeneration.bat** file can be executed to bring up an interface which allows compatibility matrices to be generated up to a specified problem parameter set. To save memory, redundant information of compatibility matrices is removed. This includes the upper triangular portion of the matrix (since the matrix is mirrored) and the diagonal of the matrix (since $H_D(i, i) = 0$ trivially). An example compatibility matrix for the $K_3(3, 2)$ problem is shown in Figure 1.

```

1
11
111
1111
11111
111111
1111111
11111111
111100100
1110100101
11100100111
100111100111
0101110101111
00111100111111
100100111111111
0100101111111111
00100111111111111
111100100111100100
1110100101110100101
11100100111100100111
100111100100111100111
0101110100101110101111
00111100100111100111111
100100111100100111111111
0100101110100101111111111
00100111100100111111111111

```

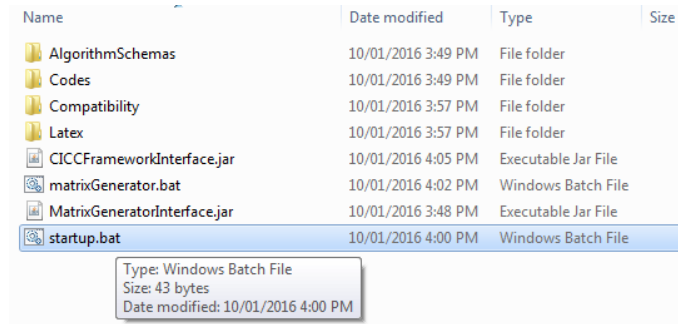
Figure 1: Compatibility matrix for the $K_3(3,2)$ is displayed.

4 Experimentation

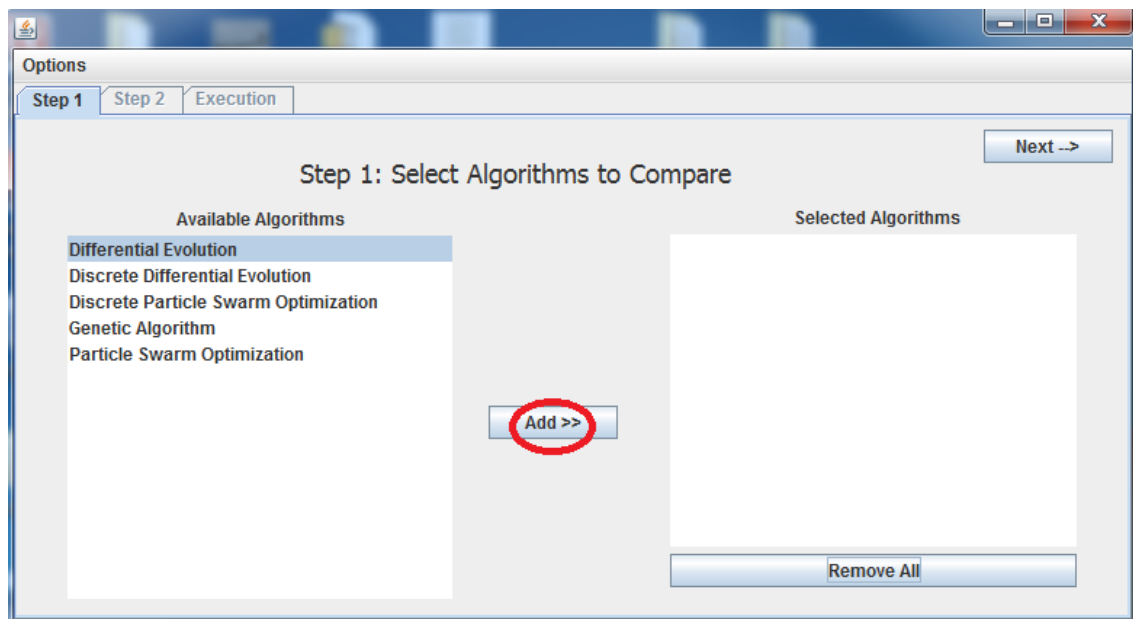
4.1 Graphical User Interface

Using the graphical user interface to perform experimentation is straightforward:

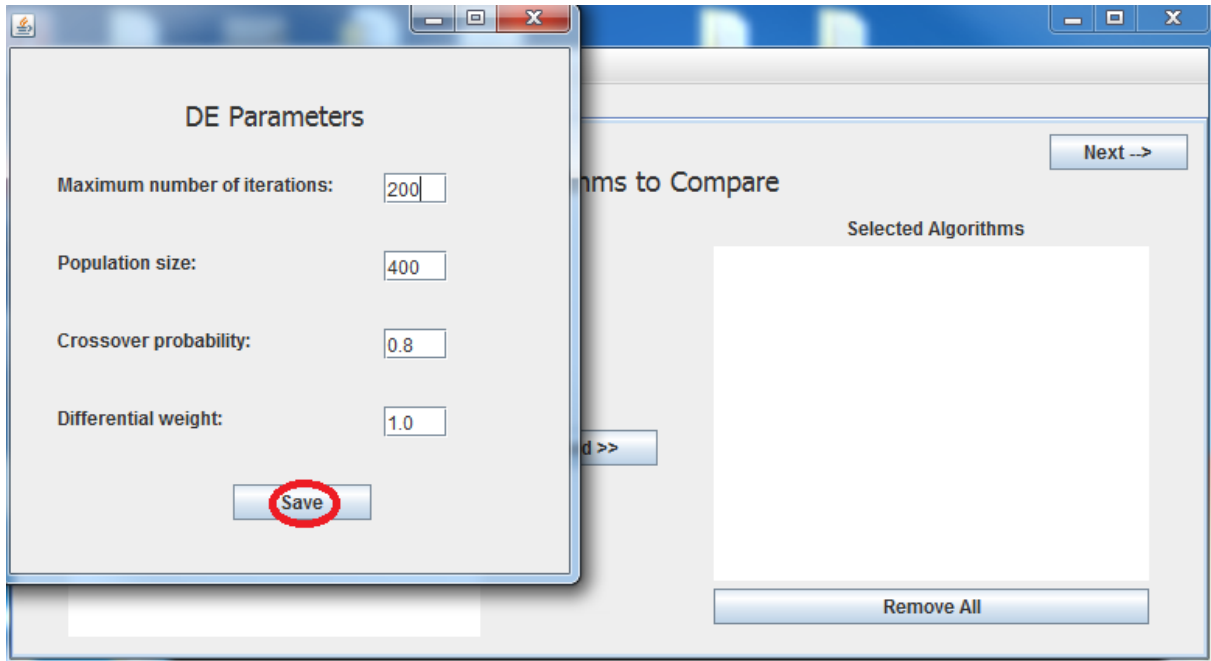
Step 1: Startup the graphical user interface via **startup.bat**



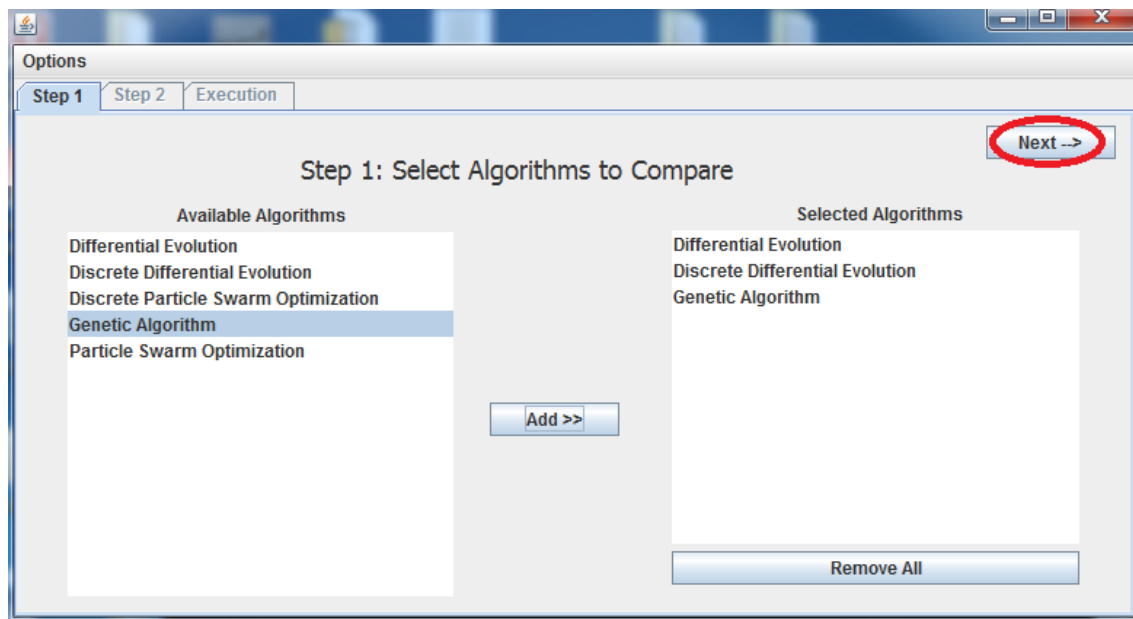
Step 2: Select an algorithm for comparison and click the “Add” button



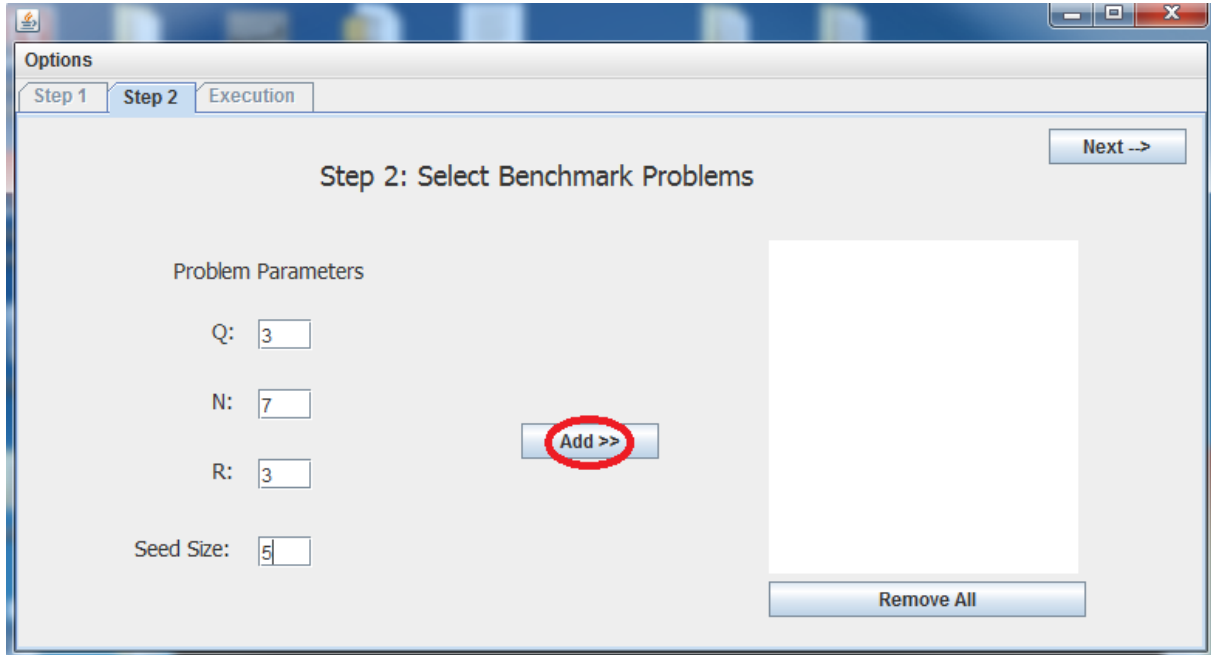
Step 3: Select a set of parameters for the added algorithm and click the “Save” button



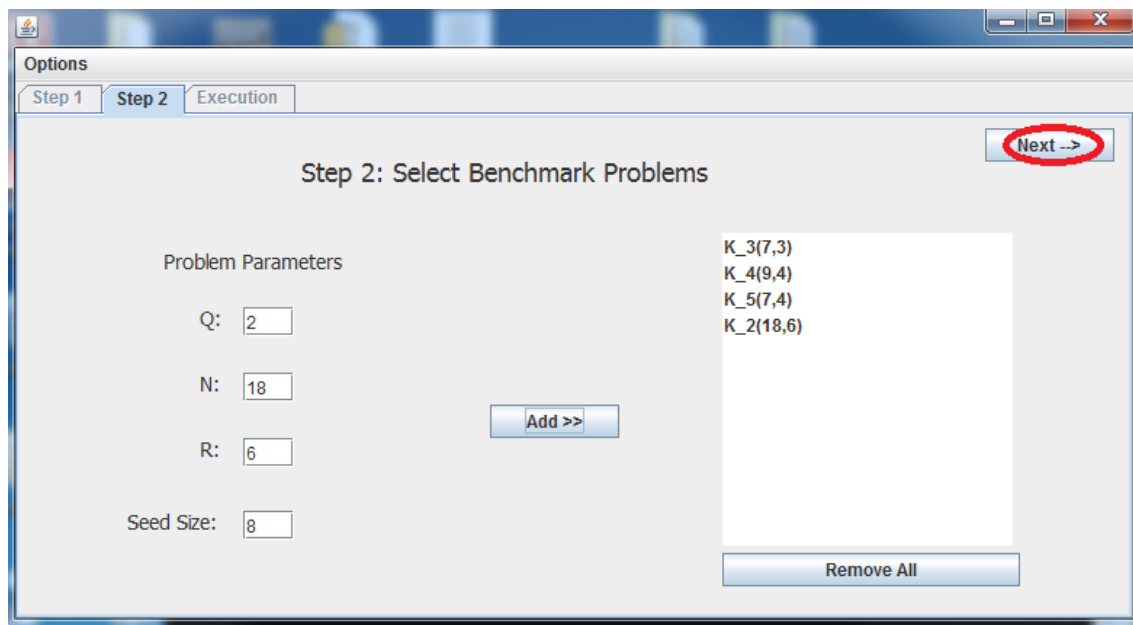
Step 4: When all desired algorithms have been added, click the “Next” button to continue



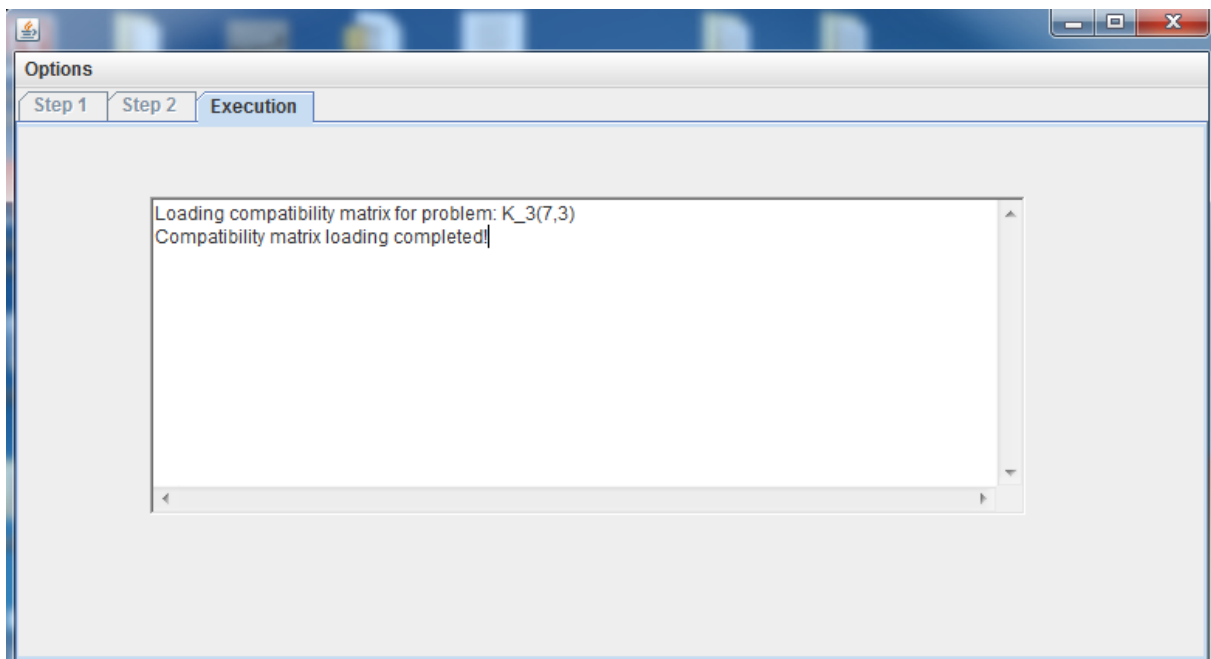
Step 5: Select a set of parameters for a desired problem and click the “Add” button








Step 6: When all desired problems have been added, click the “Next” button to continue



Step 7: Allow the algorithms to execute on each problem in the execution pane



Step 8: When all experiments have completed, results formatted as LaTeX tables can be viewed via *Latex/output.tex*

Name	Date modified	Type	Size
 output.aux	06/01/2016 10:31 ...	AUX File	
 output.log	06/01/2016 10:31 ...	Text Document	
 output.pdf	06/01/2016 10:31 ...	Adobe Acrobat D...	
 output.synctex.gz	06/01/2016 10:31 ...	WinZip File	
 output.tex	09/01/2016 8:38 PM	TeX Document	

Type: TeX Document
Size: 1.28 KB
Date modified: 09/01/2016 8:38 PM

4.2 ExecutionHandler

Experiments can also be run manually in new .java files using the `ExecutionHandler` class. The following sequence of steps are required for this purpose:

1. Create a new `ExecutionHandler` object
2. Load a hashmap of all available algorithm schemas using `XMLSchemaLoader.loadAlgorithmSchemas()`
3. Retrieve the desired algorithm schemas from the schema map by ID
4. Create `ParameterSet` objects for each algorithm and add the desired parameters to each
5. Add algorithm instances to the `ExecutionHandler` using the `attachAlgorithmInstance` method, passing in an algorithm schema and the corresponding parameter set
6. Create problems as `CoveringProblem` objects with the desired parameters
7. Add problem instances to the `ExecutionHandler` using the `attachProblemInstance` method, passing in each problem
8. Customize the `ExecutionHandler` as desired, setting items such as the output path (`setOutputPath`) and number of runs (`setNumRuns`)
9. Execute the `ExecutionHandler` by calling the `run()` method

A comprehensive example of how to use the `ExecutionHandler` can be found on the following page.

```

public class ExecutionExample {
    public static void main(String[] args) {
        ExecutionHandler handler = new ExecutionHandler();

        // retrieve algorithm schemas
        HashMap<String,AlgorithmSchema> algorithmSchemas = XMLSchemaLoader.loadAlgorithmSchemas();

        AlgorithmSchema gaSchema = algorithmSchemas.get("GA");
        AlgorithmSchema psoSchema = algorithmSchemas.get("PSO");

        // add PSO algorithm parameters
        ParameterSet psoParams = new ParameterSet();
        psoParams.put("SWARM_SIZE",500);
        psoParams.put("MAX_ITERATIONS",200);
        psoParams.put("INERTIAL_WEIGHT",0.7);
        psoParams.put("COGNITIVE_WEIGHT",1.4);
        psoParams.put("SOCIAL_WEIGHT",1.4);

        // add genetic algorithm parameters
        ParameterSet gaParams = new ParameterSet();
        gaParams.put("POP_SIZE",500);
        gaParams.put("MAX_ITERATIONS",200);
        gaParams.put("CROSSOVER_RATE",0.85);
        gaParams.put("MUTATION_RATE",0.10);
        gaParams.put("ELITISM",true);
        gaParams.put("TOURNAMENT_SIZE",3);

        // attach algorithms to the execution handler
        handler.attachAlgorithmInstance(gaSchema, gaParams);
        handler.attachAlgorithmInstance(psoSchema, psoParams);

        // create desired covering problems
        CoveringProblem problemOne = new CoveringProblem(3,9,3,5);
        CoveringProblem problemTwo = new CoveringProblem(3,9,6,5);
        CoveringProblem problemThree = new CoveringProblem(3,10,4,5);
        CoveringProblem problemFour = new CoveringProblem(3,10,5,5);

        // attach problems to the execution handler
        handler.attachProblemInstance(problemOne);
        handler.attachProblemInstance(problemTwo);
        handler.attachProblemInstance(problemThree);
        handler.attachProblemInstance(problemFour);

        // modify settings of execution handler as needed
        handler.setNumRuns(20);
        handler.writeAlgorithmOutput(true);

        // execute all runs
        handler.run();
    }
}

```

5 FAQ

1. **Q: I am running out of memory when attempting to load/generate large compatibility matrices!**

A: Allocate more heap space by running java with arguments `-Xmx <desired heap size>`

2. **Q: A runtime error is occurring stating that the compatibility matrix file is missing for the covering problem I'm trying to run?**

A: This error occurs when the compatibility matrix file has not been generated (or has been misplaced). The compatibility matrix can be generated using the matrix generator interface.

3. **Q: When loading algorithm schemas, the program throws an exception?**

A: This error occurs when there is a formatting problem within the XML schema files. Ensure that the file is in the correct format, detailed in Section 3.2.

4. **Q: When loading my desired algorithm schema, the loaded schema is incorrect!**

A: Ensure that the algorithm schema has an id attribute which is unique.

5. **Q: When writing results out in LaTeX format, the program is unable to write the file.**

A: Ensure that the chosen output path has full write permissions.

6. **Q: I want to use this framework to find optimal codes as well!**

A: Currently, the framework only supports covering codes. However, it is possible that in a future release optimal codes will be supported as well. Especially since the modifications required to accommodate optimal code problems are trivial.

7. **Q: Is there any way to turn off the automatic .code file saving for newly found bests?**

A: Currently, there is no way to turn off the automatic saving. It is a bad idea to do so, since .code files can be checked for errors to ensure that the program is not erroneously finding invalid minimal covering code bounds. This ensures the integrity of research which stems from experiments performed using the CICC framework.