FRODO:
a FRamework for Open/Distributed Optimization
Version 2.7.1

User Manual

PDF Version

Thomas Léauté Brammert Ottens
Radoslaw Szymanek
firstname.lastname@epfl.ch

EPFL Artificial Intelligence Laboratory (LIA)

`http://liawww.epfl.ch/frodo/`

November 24, 2010

Contents

1 Legal Notice

2 Introduction

FRODO is a Java open-source framework for distributed combinatorial optimization.

This manual describes FRODO version 2.x, which a complete re-design and re-implementation of the initial FRODO platform developed by Adrian Petcu. For more details on the initial platform, please refer to [16]. FRODO currently supports SynchBB [8], ADOPT [13], DSA [21], DPOP [17], S-DPOP [18], O-DPOP [19], ASO-DPOP [15], P-DPOP [6], P^2-DPOP [10], $\mathbb{E}$[DPOP] [11] and Param-DPOP.

3 FRODO Architecture

This section describes the multi-layer, modular architecture chosen for FRODO. The three layers are illustrated in Figure 1; we describe each layer in some more detail in the following subsections.

3.1 Communications Layer

The communications layer is responsible for passing messages between agents. At the core of this layer is the `Queue` class, which is an elaborate implementation of a message queue. Queues can exchange messages with each other (via shared memory if the queues run in the same JVM, or through TCP otherwise), in the form of

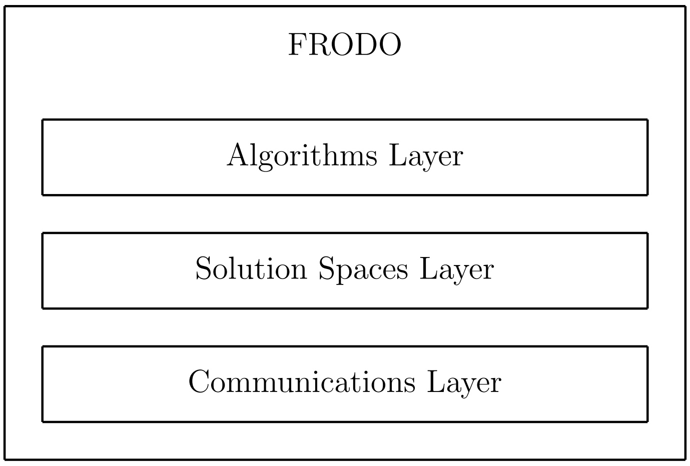


Figure 1: General FRODO software architecture.

Java `Message` objects. Classes implementing `IncomingMsgPolicyInterface` can register to a queue in order to be notified whenever messages of specific types are received. Such classes can be called *policies* because they decide what to do upon reception of certain types of messages.

Typically, in FRODO each agent owns one queue, which it uses to receive and send messages. Each queue has its own thread, which makes FRODO a multi-threaded framework. Special care has been put into avoiding threads busy waiting for messages, in order to limit the performance implications of having one thread per agent, in experiments where a large number of agents run in the same JVM.

3.2 Solution Spaces Layer

FRODO is a platform designed to solve combinatorial optimization problems; the *solution spaces* layer provides classes that can be used to model such problems. Given a space of possible assignments to some variables, a *solution space* is a representation of assignments of special interest, such as assignments that correspond to solutions of a given problem. Intuitively, one can think of a solution space as a constraint or a set of constraints that describes a subspace of solutions to a problem.

In the context of optimization problems, *utility solution spaces* are used in order to describe solutions spaces in which each solution is associated with a *utility*. Alternatively, the utility can be seen as a *cost*, if the objective of the problem is to minimize cost rather than maximize utility.

In order to reason on (utility) solution spaces, FRODO implements operations on these spaces. Examples of operations are the following:

- *join* merges two or more solutions spaces into one, which contains all the solutions in the input spaces;
- *project* operates on a utility solution space, and removes one or more variables from the space by optimizing over their values in order to maximize utility or minimize cost;
- *slice* reduces a solution space by removing values from one or more variable domains;
- *split* reduces a utility solution space by removing all solutions whose utility is above or below a given threshold.

FRODO will provide two implementations of utility solution spaces: a *hypercube* and a *utility diagram* (only hypercubes are currently implemented). A *hypercube* is an extensional representation of a space in which any combination of assignments to variables is allowed, and is associated with a given utility. Infeasible assignments then have to be represented using a negative, very large – effectively infinite – utility. *Utility diagrams* are more efficient than hypercubes when the space of feasible solutions is sparse, because they only represent the feasible solutions, using data structures similar to decision diagrams.

Solution spaces can be a way for agents to exchange information about their subproblems. For instance, in the *UTIL propagation* phase in *DPOP* [17], agents exchange *UTIL messages* that are hypercubes describing the highest achievable utility for a subtree, depending on the assignments to variables in the subtree's separator. *H-DPOP* [9] uses some form of utility diagrams instead of hypercubes.

3.3 Algorithms Layer

The algorithms layer builds upon the solution spaces layer and the communication layer in order to provide distributed algorithms to solve DCOPs. In FRODO, an algorithm is implemented as one or more *modules*, which are simply policies that describe what should be done upon the reception of such or such message by an agent's queue, and what messages should be sent to other agents, or to another of the agent's modules. This modular design makes algorithms highly and easily customizable, and facilitates code reuse, simplicity, and maintenance.

FRODO currently supports the following algorithms: SynchBB [8], ADOPT [13], DSA [21], DPOP [17], S-DPOP [18][1], O-DPOP [19], ASO-DPOP [15], P-DPOP [6],

[1]The warm restart functionality in S-DPOP is currently only available through the API; see `ch.epfl.lia.frodo.algorithms.dpop.restart.test.TestSDPOP.java` for a sample use.

P^2-DPOP [10], $\mathbb{E}$[DPOP] [11] and Param-DPOP. Param-DPOP is an extension of DPOP that supports special variables called *parameters*. Contrary to traditional *decision variables*, the agents do not choose optimal assignments to the parameters; instead, they choose optimal assignments to their decision variables and output a solution to the parametric DCOP that is a function of these parameters.

To illustrate FRODO's modular philosophy, let us consider the implementation of the DPOP algorithm. A DPOP agent uses a single queue, and is based on the generic, algorithm-independent `SingleQueueAgent` class. The behavior of this generic agent is specialized by plugging in four modules, which correspond to DPOP's four phases.

1. The *Variable Election* module describes how the agent should exchange messages with its neighboring agents in order to collectively elect one variable of the overall problem. This is implemented following P-DPOP's naive variable election algorithm [6].

2. The *DFS Generation* module has the agents exchange tokens so as to order all variables in the DCOP along a DFS tree, rooted at the variable elected by the Variable Election module, and following P-DPOP's DFS generation algorithm [6].

3. The *UTIL Propagation* module implements DPOP's traditional *UTIL propagation* phase [17], during which hypercubes describing solutions to increasingly large subproblems are aggregated and propagated along the DFS tree in a bottom-up fashion, starting at the leaves of the tree.

4. The *VALUE Propagation* module corresponds to DPOP's *VALUE propagation* phase [17], which is a top-down propagation of messages containing optimal assignments to variables.

This modular algorithm design makes it easy to implement various versions of DPOP, either by parametrizing one or more modules to make them behave slightly differently, or by completely replacing one or more modules by new modules to implement various behaviors of the agents.

4 How to Use FRODO

This section describes how to use FRODO to solve Distributed Constraint Optimization Problems (DCOPs).

4.1 Installation Procedure and Requirements

FRODO is distributed in a compressed a ZIP file, which, when expanded, contains the following five elements:

- `frodo2.jar` is a Java 1.5-compliant executable JAR file;
- `LICENSE.txt` contains the FRODO license;
- `RELEASE_NOTES.txt` summarizes changes made from version to version;
- `FRODO_User_Manual.pdf` is this user manual, also available in HTML format on the FRODO website;
- A `lib` folder. The following third-party libraries should be downloaded separately from their respective distributors and put in the `lib` folder:
  - JDOM (version 1.1.1) [4];
  - OR-Objects (version 2.0.1) [5], only required to run the DisMDVRP benchmarks.

In order to use FRODO's GUI, it is also necessary to separately install Graphviz [2] and to make sure that Graphviz' `dot` executable is on the search path.

4.2 File Formats

FRODO takes in two types of files: files defining optimization problems to be solved, and configuration files defining the nature and the settings of the agents (i.e. the algorithm) to be used to solve them.

4.2.1 Problem File Format

The file format used to describe DCOPs is based on the XCSP 2.1 format [14], with two small extensions necessary to describe which agent owns which variable, and whether the problem is a maximization or a minimization problem, as the XCSP format was designed for centralized CSPs and WCSPs, not distributed optimization problems. Figure 2 shows an example FRODO XCSP file.

A FRODO XCSP file consists of four main sections:

1. The `<domains>` section defines domains of values for the variables in the DCOP. There need not be one domain per variable; several variable definitions can refer to the same domain.

```
<instance>
  <presentation name="sampleProblem" maxConstraintArity="2"
              maximize="false" format="XCSP 2.1_FRODO" />

  <domains nbDomains="1">
    <domain name="three_colors" nbValues="3">1..3</domain>
  </domains>

  <variables nbVariables="3">
    <variable name="X" domain="three_colors" owner="agentX" />
    <variable name="Y" domain="three_colors" owner="agentY" />
    <variable name="Z" domain="three_colors" owner="agentZ" />
  </variables>

  <relations nbRelations="1">
    <relation name="NEQ" arity="2" nbTuples="3" semantics="soft" defaultCost="0">
      infinity: 1 1|2 2|3 3
    </relation>
  </relations>

  <constraints nbConstraints="3">
    <constraint name="X_and_Y_have_different_colors" arity="2" scope="X Y" reference="NEQ" />
    <constraint name="X_and_Z_have_different_colors" arity="2" scope="X Z" reference="NEQ" />
    <constraint name="Y_and_Z_have_different_colors" arity="2" scope="Y Z" reference="NEQ" />
  </constraints>
</instance>
```

Figure 2: An example FRODO XCSP file.

2. The `<variables>` section lists the variables in the DCOP, with their corresponding domains of allowed values, and the names of the agents that own them. One agent may own more than one variable. This `owner` field is an extension made to the XCSP 2.1 format [14] in order to adapt it to distributed problems.

3. The `<relations>` section defines generic *relations* over variables. A relation is to a *constraint* what a domain is to a variable: it describes a generic notion over a certain number of variables, without specifying the names of the variables. This notion can then be implemented as constraints on specific variables.

   Among all possible types of relations that are defined in the XCSP format, FRODO currently only supports the *soft* relations (`semantics = "soft"`), which list possible utility values (or cost values, depending on whether the root attribute `maximize` is `"true"` or `"false"`), and for each utility, the assignments to the variables that are associated with this utility. In the example in Figure 2, the binary relation assigns the utility value $+\infty$ to all assignments in which the two variables are equal, and a utility value of 0 to all other assignments (as specified by the `defaultCost` field). This example relation is essentially a soft relation representation of the hard inequality relation; the use of the special utilities/costs `-infinity` and `infinity` makes it possible to express hard constraints as soft constraints. Notice however that for MaxDisCSP problems in which the goal is to minimize the number of conflicts, it is necessary to avoid the use of these special infinite costs/utilities, and resort to normal, large values instead.

4. The `<constraints>` section lists the constraints in the DCOP, by referring to previously defined relations, and applying them to specific variable tuples.

Appendix A describes the format for relations and constraints in more detail.

4.2.2 Agent Configuration File Format and Performance Metrics

FRODO takes in an agent configuration file that defines the algorithm to be used, and the various settings of the algorithm's parameters when applicable. Figure 3 presents a sample agent configuration file.

Performance Metrics FRODO supports the following performance metrics:

- **Number of messages and amount of information sent**: To activate this metric, set the attribute `measureMsgs` to `"true"` in the agent configuration file. FRODO then reports the total number of messages sent, sorted by message type, as well as the total sum of the sizes of all messages sent, also sorted by message type. Note that this can be computationally expensive, as measuring message sizes involves serialization.

  NOTE: The Variable Election module exchanges a number of messages that is linear in the parameter `nbrSteps`. For optimal results, this parameter should be set to a value just above the diameter of the largest connected component in the constraint graph. A good rule of thumb is to set it to a value just above the total number of variables in the DCOP.

- **Simulated time** [20]: When the attribute `measureTime` is set to `"true"` (as it is by default), FRODO uses a centralized mailman to ensure that messages are delivered sequentially one at a time, in increasing order of their time stamps.

- **Non-Concurrent Constraint Checks (NCCCs)** [7]: To activate the counting of NCCCs, set the attributes `countNCCCs` to `"true"` in the relevant module definitions.

```
<agentDescription className = "ch.epfl.lia.frodo.algorithms.SingleQueueAgent"
  measureTime = "true" measureMsgs = "false" >

  <parser parserClass = "ch.epfl.lia.frodo.algorithms.XCSPparser"
          displayGraph = "false"
          utilClass = "ch.epfl.lia.frodo.solutionSpaces.AddableInteger" />

  <modules>
    <module className = "ch.epfl.lia.frodo.algorithms.dpop.VariableElection"
      nbrSteps = "150" >
      <varElectionHeuristic
        className = "ch.epfl.lia.frodo.algorithms.heuristics.ScoringHeuristicWithTiebreaker" >
        <heuristic1
          className = "ch.epfl.lia.frodo.algorithms.heuristics.MostConnectedHeuristic" />
        <heuristic2
          className = "ch.epfl.lia.frodo.algorithms.heuristics.ScoringHeuristicWithTiebreaker" >
          <heuristic1
            className = "ch.epfl.lia.frodo.algorithms.heuristics.SmallestDomainHeuristic" />
          <heuristic2 className = "ch.epfl.lia.frodo.algorithms.heuristics.VarNameHeuristic" />
        </heuristic2>
      </varElectionHeuristic>
    </module>

    <module className = "ch.epfl.lia.frodo.algorithms.dpop.DFSgeneration"
      reportStats = "true" >
      <dfsHeuristic
        className = "ch.epfl.lia.frodo.algorithms.dpop.DFSgeneration$ScoreBroadcastingHeuristic" >
        <scoringHeuristic
          className = "ch.epfl.lia.frodo.algorithms.heuristics.ScoringHeuristicWithTiebreaker" >
          <heuristic1
            className = "ch.epfl.lia.frodo.algorithms.heuristics.MostConnectedHeuristic" />
          <heuristic2
            className = "ch.epfl.lia.frodo.algorithms.heuristics.SmallestDomainHeuristic" />
        </scoringHeuristic>
      </dfsHeuristic>
    </module>

    <module className = "ch.epfl.lia.frodo.algorithms.dpop.UTILpropagation"
      reportStats = "true"
      countNCCCs = "false" />

    <module className = "ch.epfl.lia.frodo.algorithms.dpop.VALUEpropagation"
      reportStats = "true" />
  </modules>

</agentDescription>
```

Figure 3: Example of a FRODO agent configuration file, corresponding to the classical version of DPOP.

Other Statistics Several algorithmic modules can also report other statistical information about the problem. Whenever applicable, you can set the attribute `reportStats` to `"true"` to get access to these statistics. For instance, in the case of DPOP (Figure 3), the DFS Generation module can report the DFS tree that is computed and used by DPOP, using the DOT format [2], while the VALUE Propagation module can report the optimal assignments to the DCOP variables and the corresponding total utility. Setting the parser's attribute `displayGraph` to `true` also results in displaying the constraint graph in DOT format. Wherever applicable, setting the attribute `DOTrenderer` to `ch.epfl.lia.frodo.gui.DOTrenderer` (instead of the empty string) will render graphs in a GUI window instead of printing them in DOT format. This functionality requires that Graphviz [2] be preliminarily installed as described in Section 4.1.

4.3 Simple Mode

FRODO can be run in two modes: in simple mode, and in advanced mode (Section 4.4). In simple mode, all agents run in the same Java Virtual Machine. The distributed nature of the algorithm is simulated by assigning (at least) one thread per agent. Agents exchange messages by simply sharing pointers to objects in memory.

4.3.1 With Graphical User Interface

The simple mode with Graphical User Interface (GUI) is launched using the `main` method of the class `SimpleGUI` in the package `ch.epfl.lia.frodo.gui`. This is defined as the default entry point of `frodo2.jar`, therefore the following command should be used from within the directory containing the FRODO JAR file:

```
java -jar frodo2.jar
```

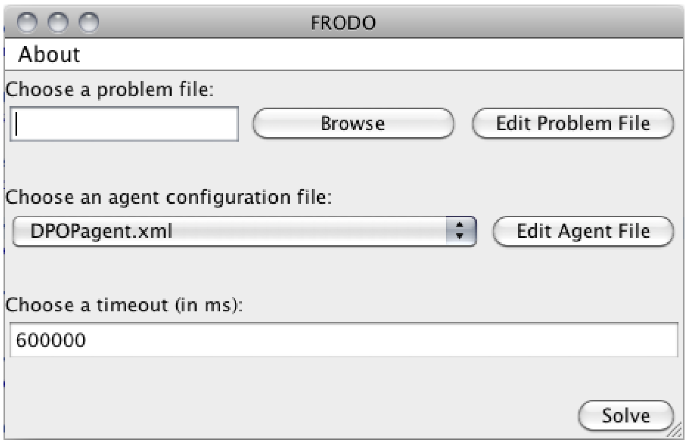
The method takes in two optional arguments, in the following order (the first argument is only optional if the second is not specified):

1. the path to the problem file;
2. the path to the agent file.

If the path to the agent file is omitted, FRODO uses the DPOP agent file `DPOPagent.xml` by default. The simple mode supports one option:

- `-timeout` *msec*: sets a timeout, where *msec* is a number of milliseconds. The default timeout is 10 minutes. If set to `0`, the timeout is disabled.

A screenshot of the GUI is presented in Figure 4. It allows the user to specify (and, optionally) edit a problem file in XCSP format, to select (and, optionally) edit an agent configuration file, and to impose a timeout. During the execution of the chosen DCOP algorithm, FRODO also displays in separate windows the constraint graph and the variable ordering used, as illustrated in Figure 5. To render these graphs, FRODO uses Graphviz [2], which must be preliminarily installed as described in Section 4.1.



Choose a problem file:

Choose an agent configuration file: DPOPagent.xml

Choose a timeout (in ms): 600000

Figure 4: FRODO's main GUI window.

4.3.2 Without GUI

The simple mode without GUI is launched using the `main` method of the class `AgentFactory` in the package `ch.epfl.lia.frodo.algorithms`, which can be achieved using the following command, called from within the directory containing the FRODO JAR file:

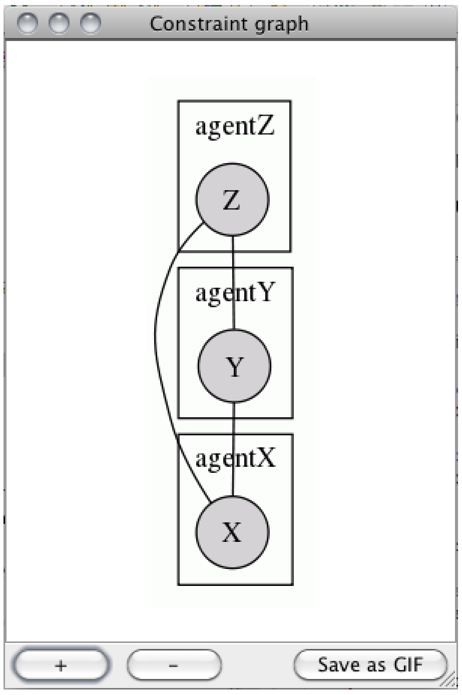
```
java -cp frodo2.jar ch.epfl.lia.frodo.algorithms.AgentFactory
```

The arguments are almost the same as for the simple mode with GUI, except that the path to the problem file is required, and the following option is also supported:

- `-license`: FRODO prints out the license and quits.

4.4 Advanced Mode

FRODO's advanced mode can be used to run algorithms in truly distributed settings, with agents running on separate computers and communicating through



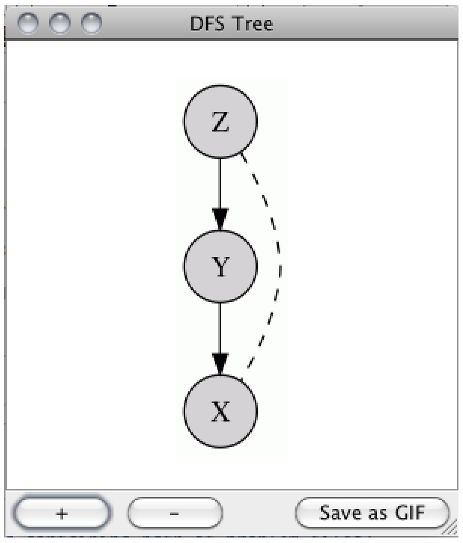


Figure 5: The constraint graph and DFS tree rendered by FRODO's GUI for the problem instance in Figure 2.

```
<experiment>

 <configuration>
   <resultFile fileName = "resultFile.log"/>
   <agentDescription agentName = "algorithms/dpop/DPOPagent.xml"/>
 </configuration>

 <problemList nbProblem = "2">
    <file fileName = "problem1.xml"/>
    <file fileName = "problem2.xml"/>
 </problemList>

</experiment>
```

Figure 6: Example of a configuration file for FRODO's advanced mode.

TCP. In this mode, each computer runs a *daemon*, which initially waits for a centralized *controller* to send the descriptions of the algorithm to use and the problem(s) to solve. The controller is only used during the initial setup phase; once the algorithm is started, the agents communicate with each other directly, and the controller could even be taken offline. In the context of experiments, for the purpose of monitoring the solution process on a single computer, agents can also be set up to report statistics and the solution to the problem(s) to the controller.

Another advantage of using the advanced mode is that it is very easy to set up batch experiments. The configuration file (see Figure 6) can contain a list of problems that will be solved sequentially by the controller. The agent to be used is defined by the field `agentName` in the `agentDescription` element. It is also possible to replace the `agentName` field with `fileName = "agent.xml"`, where `agent.xml` is the name of a file describing the agent to be used.

FRODO's advanced mode has two submodes:

- The *local* submode uses only one JVM and a single computer; there is only one daemon, spawned by the controller itself, and all agents run in the controller's JVM.
- In *distributed* submode, daemons are started by the user in separate JVMs (possibly on separate computers), and wait for the problem and algorithm descriptions from the centralized controller.

IMPORTANT NOTE: The advanced mode does not support the simulated time metric (Section 4.2.2).

4.4.1 Running in Local Submode

To run the controller in local submode, the `Controller` class in the package `ch.epfl.lia.frodo.controller` must be launched with the argument `-local`, using the following command from within the directory containing `frodo2.jar`:

```
java -cp frodo2.jar ch.epfl.lia.frodo.controller.Controller -local
```

As an optional argument, one can set the work directory by giving the argument `-workdir` *path*. The default work directory is the one from where the controller is launched.

When the controller is launched, a simple console-based UI is started. To load a particular configuration file, one uses the `open` command:

>`open` *configuration_file*

This command tells the controller to load the configuration file that contains all the information necessary to run the experiments. A sample configuration file can be found in Figure 6. To run the experiments, simply give the command `start`. When all the experiments are finished, the controller can be exited by giving the `exit` command.

4.4.2 Running in Distributed Submode

To run the controller in distributed submode, the `Controller` class in the package `ch.epfl.lia.frodo.controller` must be launched, without the `-local` option, using the following command from within the directory containing `frodo2.jar`:

```
java -cp frodo2.jar ch.epfl.lia.frodo.controller.Controller
```

To set the work directory one can again use the `-workdir` argument. When running in distributed mode, the controller assumes that the agents must be run on a set of daemons. These daemons can run on the same machine or on different machines. To start a daemon, open a new console, and launch the `Daemon` class in the package `ch.epfl.lia.frodo.daemon`, using the following command from within the directory containing `frodo2.jar`:

```
java -cp frodo2.jar ch.epfl.lia.frodo.daemon.Daemon
```

The IP address of the controller can either be given with the command-line argument `-controller` *ip_address*, or by issuing the command in the daemon console:

```
>controller ip_address
```

The port number used for the controller is 3000. When all the daemons are running, one can check whether they are correctly registered to the controller by using the following command in the controller console:

```
>get daemons
```

The configuration file can be loaded by the `open` command and the experiments started by using the `start` command. When the experiments are started, the agents are assigned to the different daemons in a round robin fashion. In the future we intend to allow for more flexibility in assigning agents to particular daemons.

4.5 Running Experiments

To run experiments and gather statistics about various algorithms on DCOP benchmark instances, it is necessary to interact with FRODO directly through the Java API, rather than through the command-line like in FRODO's *simple mode* and *advanced mode* (Sections 4.3 and 4.4). To this purpose, FRODO provides a special class called a *solver* for each DCOP algorithm, which is a sub-class of the abstract class `AbstractDCOPsolver`. Solvers provide several `solve` methods, the most useful of which is the following:

```
public Solution solve (org.jdom.Document problem,
                       int nbrElectionRounds) { ... }
```

The first input must be a JDOM `Document` object that represents the DCOP problem to solve, in XCSP format (Section 4.2.1). You can generate such a `Document` object from an XCSP file using one of the static `parse` methods of the `XCSPparser` class. Alternatively, FRODO's benchmarking problem generators usually provide methods that directly produce `Document` objects.

The second input is the number of rounds for the `VariableElection` module used to choose the first variable in the variable ordering (for the DCOP algorithms that need one). It is important to set this parameter as low as possible to reduce the complexity of the `VariableElection` module, while keeping it higher than the diameter of the constraint graph to ensure correctness. For random, unstructured problems, this parameter can be set to the number of variables in the problem. For more structured problems, it might be possible to set it to a lower value; for instance, if the problem domain has the property that each agent's local subproblem is a clique, then this parameter can be set to 2 times the number of agents, which is smaller than the number of variables as soon as each agent owns at least 2 variables.

Finally, if you intend to measure the runtime of the algorithms (be it wall clock time or *simulated time*), it is recommended to destroy and create a new JVM after each run. Otherwise, the algorithm that is run first might be disadvantaged by the time it takes to initialize the JVM and load all required Java classes.

4.6 Troubleshooting

If you encounter errors or exceptions when using FRODO, you might want to pass the option `-ea` to the JVM in order to enable `asserts`. With this option on, FRODO will perform some optional (potentially expensive) tests on its inputs, which can sometimes help resolve problems. Various helpful tools (such as a bug tracker) are available on FRODO's SourceForge website. The FRODO development team can also be contacted by email (refer to the contact details on the title page of this document). We warmly welcome constructive feedback about FRODO in order to constantly improve the platform and make it better fit users' needs.

5 How to Extend FRODO

This section briefly describes the recommended steps one should go through in order to implement a new DCOP algorithm inside FRODO. This procedure is illustrated using the SynchBB algorithm.

5.1 Step 1: Writing the Agent Configuration File

Modularity is and must remain one of the strong points of FRODO. When considering implementing a new algorithm, first think carefully about possible phases of the algorithm, which should be implemented in separate modules if possible. A DCOP algorithm is then defined by its *agent configuration file*, which lists all the modules that constitute the algorithm. The configuration file for DPOP was already given in Figure 3; we now illustrate step-by-step how to write the configuration file for SynchBB. The general structure of an agent configuration file is the following (in XML format).

```
<agentDescription className = "ch.epfl.lia.frodo.algorithms.SingleQueueAgent"
  measureTime = "true"
  measureMsgs = "false" >

  <parser parserClass = "ch.epfl.lia.frodo.algorithms.XCSPparser"
    displayGraph = "false"
    utilClass = "ch.epfl.lia.frodo.solutionSpaces.AddableInteger"/>
  <modules>
    <!-- List of module elements -->
  </modules>
</agentDescription>
```

Several modules are already available for you to reuse, in particular when it comes to generating an ordering on the variables before the core of the DCOP algorithm is started.

The `VariableElection` Module This module can be reused to implement any algorithm that needs to elect a variable, for instance as the first variable in the variable ordering. It works by assigning a score to each variable, and then uses a viral propagation mechanism to find the variable with the highest score. It must be parameterized by a number of steps for the viral propagation, which must be greater than the diameter of the constraint graph to ensure correctness. It can also be parameterized by a set of scoring heuristics and recursive, tie-breaking heuristics. For instance, SynchBB elects the first variable in its ordering using the `VariableElection` module, with the *smallest domain* heuristic, breaking ties by lexicographical ordering of the variable names.

```
<module className = "ch.epfl.lia.frodo.algorithms.dpop.VariableElection"
  nbrSteps = "150" >

  <varElectionHeuristic
    className = "ch.epfl.lia.frodo.algorithms.heuristics.ScoringHeuristicWithTiebreaker" >
    <heuristic1
      className = "ch.epfl.lia.frodo.algorithms.heuristics.SmallestDomainHeuristic" />
    <heuristic2
      className = "ch.epfl.lia.frodo.algorithms.heuristics.VarNameHeuristic" />
  </varElectionHeuristic>
</module>
```

The `LinearOrdering` Module This module constructs a total ordering of the variables, starting with the variable chosen by the `VariableElection` module. Currently, it uses the *min width* heuristic in oder to produce low-width variable orders; a future version of FRODO might make this heuristic customizable. The module takes in an boolean parameter `reportStats` whose purpose is explained in Section 5.2.4.

```
<module className = "ch.epfl.lia.frodo.algorithms.synchbb.LinearOrdering"
  reportStats = "true" />
```

Other DCOP algorithms based on a pseudo-tree ordering of the variables instead of a total ordering should reuse the `DFSgeneration` module implemented for DPOP (Figure 3).

The Main Module – `SynchBB` After the two modules for generating the variable ordering have been declared, it remains to declare the module(s) that constitute the core of the DCOP algorithm. Typically, if the algorithm is easily decomposable into several phases, there should be one module per phase, like in the case of DPOP (Figure 3). For SynchBB, which is a simpler, single-phase algorithm, a single module is sufficient (Figure 7).

The module may be parameterized by various attributes. The `reportStats` parameter has a special usage discussed in Section 5.2.4. The `SynchBB` module has been implemented to take in two additional parameters: `countNCCCs` is a boolean attribute that specifies whether the module should count Non-Concurrent Constraint Checks (Section 4.2.2), and `convergence` whether the module should keep track of the history of its variable assignments so that the experimenter can later analyze the convergence properties of the algorithm (Section 5.2.4).

```
<module className = "ch.epfl.lia.frodo.algorithms.synchbb.SynchBB"
  reportStats = "true"
  countNCCCs = "false"
  convergence = "false" />
```

Figure 7: XML fragment describing the parameters of the `SynchBB` module.

5.2 Step 2: Implementing the Module(s)

In FRODO, the modules defined in the agent configuration file behave like message listeners (one instance per agent in the DCOP), implementing the interface `IncomingMsgPolicyInterface<String>`.

5.2.1 The Interface `IncomingMsgPolicyInterface`

This interface declares the following method, which is called by FRODO whenever the agent receives a message of interest:

```
public void notifyIn (Message msg);
```

`IncomingMsgPolicyInterface<String>` is itself a sub-interface of the interface `MessageListener<String>`, which declares the following two methods:

```
public Collection<String> getMsgTypes ();
public void setQueue(Queue queue);
```

The method `getMsgTypes` must return the types of messages that the module wants to be notified of. The type of a message is defined as the output of

`Message.getType()`. The method `setQueue` is called by FRODO when the agents are initialized, and passes to the module the agent's `Queue` object that the module should use to send messages to other agents.

5.2.2 Sending Messages

Sending messages can be achieved by calling one of the following methods of the module's `Queue` object:

```
Queue.sendMessage (Object to, Message msg)
Queue.sendMessageToMulti (Collection recipients, Message msg)
Queue.sendMessageToSelf (Message msg)
```

The method `sendMessageToSelf` is used by the module to send messages to another module of the same agent. This is how modules communicate with each other within the same agent; for instance, the `SynchBB` module listens for the output messages of the agent's `LinearOrdering` module, which are of the class `OrderMsg`. All messages exchanged by all algorithms must be of the class `Message`, or a subclass thereof. Subclasses corresponding to messages with various numbers of payloads are provided for convenience: `MessageWithPayload`, `MessageWith2Payloads`, and `MessageWith3Payloads`.

Optionally, to improve the performance of your algorithm in terms of message sizes, the message classes should also implement the interface `Externalizable`. This allows for instance to not count the `type` field of the message when measuring its size. This improvement is not necessary for *virtual messages* that are only sent by an agent to itself. Note that the `Externalizable` interface is an advanced feature of Java, and should be handled with care. In particular, it is very important to remember that **all externalizable classes must have a public empty constructor**. If an externalizable class does not have a public empty constructor, Java will not display any compilation error or warning, and the JVM will freeze without giving any justification when it attempts to serialize that class.

Also, notice that the destinations passed to the queue's methods `sendMessage` and `sendMessageToMulti` are the **names of the agents**, not the names of the variables. Finally, when the algorithm has terminated, the module should send a message of type `AgentInterface.AGENT_FINISHED` to itself, which will be caught by `SingleQueueAgent`.

5.2.3 The Module Constructor

All modules declared in the agent configuration file must have a constructor with the following signature:

```
public MyModule (DCOPProblemInterface, org.jdom.Element) { ... }
```

Figure 8: The signature of the required constructor for all FRODO modules.

The first input is used by the module to access the description of the agent's subproblem. The interface `DCOPProblemInterface` declares are large number of methods that the module can call to retrieve information about neighboring agents, variables, domains, and constraints. As explained in Section 3.2, in FRODO, constraints are called *solution spaces*, and should be accessed using one of the `DCOPProblemInterface.getSolutionSpaces` methods.

Important note: for runtime measurements to be correct, none of the methods of `DCOPProblemInterface` should be called within the module's constructor, because all reasoning about the problem should be delayed until the algorithm is actually started. This happens when the agent receives a message of type `AgentInterface.START_AGENT`.

The second input of the module's constructor is a JDOM `Element` object that represents the module's XML fragment from the agent configuration file. For instance, for the `SynchBB` module, the `Element` object contains the XML fragment in Figure 7, and the constructor can be implemented as follows:

```
public SynchBB (DCOPProblemInterface problem, Element parameters) {
  this.problem = problem;
  this.countNcccs = Boolean.parseBoolean(
    parameters.getAttributeValue("countNCCCs"));
  this.convergence = Boolean.parseBoolean(
    parameters.getAttributeValue("convergence"));
}
```

5.2.4 Reporting Statistics

As previously mentioned in Section 4.2.2, it can be useful for a module to report statistics about the problem, the solution process, and the solution found. In FRODO, this is done as follows: a special *statistics gatherer* agent is created that listens to statistics messages sent by all DCOP agents, combines them in order to get a global view of the overall solution process, and makes it available to the user. The code that takes care of aggregating statistics must be implemented inside the module that produces these statistics. To clarify how this works, let us consider the case of the `SynchBB` module.

The `StatsReporter` Interface The XML description of the `SynchBB` module in Figure 7 defines a parameter `reportStats`, set to `true`. FRODO automatically interprets this as the fact that the module implements the interface `StatsReporter`, which declares the following method (among others):

```
public void getStatsFromQueue (Queue queue);
```

Inside this method, the module must notify the statistics gatherer's queue of the types of the statistics messages it wants to aggregate. This can be done by calling the method `Queue.addIncomingMessagePolicy`. The module is then notified of statistics messages received by the statistics gatherer agent by a call to its `notifyIn` method, just like for normal messages.

All modules implementing `StatsReporter` are expected to have a constructor with the following signature:

```
public MyStatsReporter (org.jdom.Element, DCOPProblemInterface) { ... }
```

Notice that the order of the inputs is reversed compared to the constructor of classes implementing `IncomingMsgPolicyInterface`, given in Figure 8. Since `StatsReporter` is a sub-interface of the latter, a module that reports statistics must have both constructors. The first input is the XML description of the module, as in Figure 8. The second input describes the **overall** DCOP problem (while in Figure 8 it described the agent's local subproblem).

Studying Convergence Many DCOP algorithms such as SynchBB have an any-time behavior, and it can be interesting to study their convergence properties. A sub-interface of `StatsReporter`, called `StatsReporterWithConvergence`, is provided for this purpose. It declares the two following methods:

```
public HashMap< String, ArrayList< CurrentAssignment<Val> > >
       getAssignmentHistories();
public Map<String, Val> getCurrentSolution();
```

Consult the implementation of the `SynchBB` module for an example of how to use this functionality.

Counting NCCCs FRODO makes it possible to count Non-Concurrent Constraint Checks (Section 4.2.2). Adding support for NCCCs to an algorithm such as SynchBB is done by performing the two following operations:

1. Initializing the NCCC count to zero in the module's `setQueue` method, as follows:

```
public void setQueue(Queue queue) {
  this.queue = queue;
  if (this.countNcccs && queue.getNCCCS() < 0)
    queue.setNCCCS(0);
}
```

2. Calling the method `Queue.incrementNCCCs` on the module's queue whenever a constraint check is performed.

5.3 Step 3: Implementing a Dedicated *Solver*

This third step is optional, as the two previous implementation steps already make it possible to use your algorithm in FRODO's *simple mode* and *advanced mode* (Sections 4.3 and 4.4). However, it can be convenient to have a *solver* class to call your algorithm through the API for the purpose of running experiments (Section 4.5) or testing correctness (Section 5.4). The abstract class `AbstractDCOPsolver` can be extended to produce such a solver; it declares the following two abstract methods:

```
protected abstract ArrayList<StatsReporter> getSolGatherers ();
protected abstract S buildSolution ();
```

The method `getSolGatherers` must return instances of the modules that report statistics, which will be automatically added to the queue of the statistics gatherer agent. For SynchBB, only the `SynchBB` module reports relevant statistics about the solution found, and therefore the `SynchBBsolver` class implements this method as follows:

```
protected ArrayList<StatsReporter> getSolGatherers() {
  ArrayList<StatsReporter> solGatherers =
    new ArrayList<StatsReporter> (1);
  this.module = new SynchBB ((Element) null, super.parser);
  this.module.setSilent(true);
  solGatherers.add(module);
  return solGatherers;
}
```

After the algorithm has terminated, the method `buildSolution` is called, which must extract statistics from the modules created in `getSolGatherers` and return an object of type `Solution`. Because SynchBB reports convergence statistics, its solver actually returns an object of class `SolutionWithConvergence`, which extends `Solution`.

5.4 Step 4: Testing

An important strength of FRODO is that it is systematically, thoroughly tested using JUnit tests. As soon as you have completed a first implementation of a module (or, ideally, even before you start implementing it), write JUnit tests to make sure it behaves as expected on its own. FRODO being an intrinsically multi-threaded framework, you should use repetitive, randomized tests whenever it makes sense to do so. Once all modules are assembled together and the algorithm is completed, write unit tests against other algorithms that have already been implemented, to check that the outputs of the algorithms are consistent.

An example of a JUnit test is the class `SynchBBagentTest`, which extends DPOP's test class `DPOPagentTest` to favor code reuse. The use of *solvers* (Section 5.3) make it straightforward to implement unit tests for an algorithm, as demonstrated in the class `P_DPOPagentTest`. The class `AllTests` in the package `ch.epfl.lia.frodo.algorithms.test` provides various methods to create random DCOP instances to be used as inputs for the tests.

A Appendix – Catalogue of Constraints

FRODO currently only supports *extensional soft constraints*, and intensional, *vehicle routing* constraints. This appendix describes in some level of detail the XCSP format for these constraints. Note that we also provide an XML Schema file to help users write and validate XCSP problem files (see the file `XCSPschema.xsd` in the `algorithms` package). There are plans to couple FRODO with the Constraint Programming solver *JaCoP* [3], which would enrich FRODO's constraint catalogue with the numerous global constraints supported in JaCoP.

A.1 Extensional Soft Constraint

FRODO's format for extensional soft constraints is based on the official XCSP 2.1 format for weighted tuples [14], in abridged notation, with the following two modifications:

1. Infinite values are represented by the string `infinity` rather than by the element `<infinity/>`;
2. Utilities/costs and variables are allowed to take on decimal values (but you may then have to specify in the agent configuration file that you want FRODO to use `AddableReal` instead of the default `AddableInteger`).

Figure 2 already provided a small example of an extensional soft constraint. More generally, such a constraint is specified as follows (for a ternary constraint):

```
<constraint name="uniqueConstraintName" arity="3"
            scope="x1 x2 x3" reference="relationName" />
```

where `relationName` must be the unique name of a relation, specified as follows:

```
<relation name="relationName" arity="3" nbTuples="4"
          semantics="soft" defaultCost="0">
  1 : 0 0 0 | 10 : 0 0 1 | infinity : 0 1 0 | infinity : 0 1 1
</relation>
```

where tuples are separated by a pipe character `|`, and each tuple has the format `utilityOrCost : valueForVar1 valueForVar2 ... valueForVarN`. The order of tuples does not matter. The first part of the tuple specifying the utility/cost can be omitted if it is the same as for the previous tuple, such that the following is a valid, shorter representation of the same relation:

```
    <relation name="relationName" arity="3" nbTuples="4"
              semantics="soft" defaultCost="0">
      1 : 0 0 0 | 10 : 0 0 1 | infinity : 0 1 0 | 0 1 1
    </relation>
```

The attribute `defaultCost` specifies the utility/cost assigned to tuples that are not explicitly represented; for instance, in the previous relation, all tuples in which the first variable equals `1` have utility/cost `0`.

A.2 Vehicle Routing Constraint

FRODO also supports intensional, *vehicle routing* constraints, as described in [12]. A vehicle routing constraints is specified as follows (for an instance involving 3 customers and 4 vehicles):

```
<constraint name="uniqueConstraintName" arity="3"
      reference="global:vehicle_routing" scope="cust1 cust2 cust3">
    <parameters>
        <depot nbVehicles="4" maxDist="0.0" maxLoad="80"
               xCoordinate="20.0" yCoordinate="20.0" />
        <customers>
            <customer varName="cust1" id="1" demand="9"
                      xCoordinate="20.0" yCoordinate="26.0" />
            <customer varName="cust2" id="2" demand="3"
                      xCoordinate="27.0" yCoordinate="23.0" />
            <customer varName="cust3" id="3" demand="6"
                      xCoordinate="13.0" yCoordinate="13.0" />
        </customers>
    </parameters>
</constraint>
```

Notice that, contrary to extensional soft constraints (Section A.1) that are defined through the intermediary of `<relation>` elements to which they refer via the attribute `reference`, vehicle routing constraints are defined directly inside the `<constraint>` element, and the attribute `reference` must be set to `"global:vehicle_routing"`.

The XML Schema standard version 1.0 does not support this format (in which the type of the constraint depends on the value of the attribute `reference`); we provide an XML Schema 1.1 file for this extension of the FRODO XCSP format (`XCSPschema_v1.1xsd` in the `algorithms` package). Using this XML Schema to verify XCSP files requires an XML parser that supports XML Schema 1.1; we suggest the use of the Xerces2 Java Parser [1], version 2.10.0-xml-schema-1.1-beta.

References

[1] The Apache Xerces project. http://xerces.apache.org.

[2] Graphviz – Graph Visualization Software. http://www.graphviz.org/.

[3] JaCoP java constraint programming solver. http://jacop.osolpro.com/.

[4] The JDOM XML toolbox for Java. http://www.jdom.org/.

[5] Operations Research – Java Objects. http://opsresearch.com.

[6] Boi Faltings, Thomas Léauté, and Adrian Petcu. Privacy Guarantees through Distributed Constraint Satisfaction. In *Proceedings of the 2008 IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT'08)*, pages 350–358, Sydney, Australia, December 9–12 2008.

[7] Amir Gershman, Roie Zivan, Tal Grinshpoun, Alon Grubshtein, and Amnon Meisels. Measuring distributed constraint optimization algorithms. In *Proceedings of the AAMAS'08 Distributed Constraint Reasoning Workshop (DCR'08)*, pages 17–24, Estoril, Portugal, May 13 2008.

[8] Katsutoshi Hirayama and Makoto Yokoo. Distributed partial constraint satisfaction problem. In Gert Smolka, editor, *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming (CP'97)*, volume 1330, pages 222–236, Linz, Austria, Oct. 29–Nov. 1 1997.

[9] Akshat Kumar, Adrian Petcu, and Boi Faltings. H-DPOP: Using hard constraints for search space pruning in DCOP. In *Proceedings of the Twenty-Third Conference on Artificial Intelligence (AAAI'08)*, pages 325–330, Chicago, Illinois, USA, July 13—17 2008. AAAI Press.

[10] Thomas Léauté and Boi Faltings. Privacy-Preserving Multi-agent Constraint Satisfaction. In *Proceedings of the 2009 IEEE International Conference on PrivAcy, Security, riSk And Trust (PASSAT'09)*, pages 17–25, Vancouver, British Columbia, August 29–31 2009. IEEE Computer Society Press.

[11] Thomas Léauté and Boi Faltings. E[DPOP]: Distributed Constraint Optimization under Stochastic Uncertainty using Collaborative Sampling. In *Proceedings of the IJCAI'09 Distributed Constraint Reasoning Workshop (DCR'09)*, pages 87–101, Pasadena, California, USA, July 13 2009.

[12] Thomas Léauté, Brammert Ottens, and Boi Faltings. Ensuring Privacy through Distributed Computation in Multiple-Depot Vehicle Routing Problems. In *Proceedings of the ECAI'10 Workshop on Artificial Intelligence and Logistics (AILog'10)*, Lisbon, Portugal, August 17 2010.

[13] Pragnesh J. Modi, W Shen, Milind Tambe, and Makoto Yokoo. ADOPT: Asynchronous distributed constraint optimization with quality guarantees. *Artificial Intelligence*, 161:149–180, 2005.

[14] Organising Committee of the Third International Competition of CSP Solvers. *XML Representation of Constraint Networks – Format XCSP 2.1*, January 15 2008. `http://www.cril.univ-artois.fr/~lecoutre/research/benchmarks/benchmarks.html`.

[15] Brammert Ottens and Boi Faltings. Coordinating Agent Plans Through Distributed Constraint Optimization. In *Proceedings of the ICAPS'08 Multiagent Planning Workshop (MASPLAN'08)*, Sydney, Australia, September 14 2008.

[16] Adrian Petcu. FRODO: A FRamework for Open/Distributed constraint Optimization. Technical Report 2006/001, Swiss Federal Institute of Technology (EPFL), Lausanne (Switzerland), 2006.

[17] Adrian Petcu and Boi Faltings. DPOP: A Scalable Method for Multiagent Constraint Optimization. In Leslie Pack Kaelbling and Alessandro Saffiotti, editors, *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI'05)*, pages 266–271, Edinburgh, Scotland, July 31 – August 5 2005. Professional Book Center, Denver, USA.

[18] Adrian Petcu and Boi Faltings. S-DPOP: Superstabilizing, fault-containing multiagent combinatorial optimization. In Manuela M. Veloso and Subbarao Kambhampati, editors, *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI'05)*, pages 449–454, Pittsburgh, Pennsylvania, U.S.A, July 9–13 2005. AAAI Press / The MIT Press.

[19] Adrian Petcu and Boi Faltings. O-DPOP: An algorithm for open/distributed constraint optimization. In *Proceedings of the Twenty-First National Conference on Artificial Intelligence (AAAI'06)*, pages 703–708, Boston, Massachusetts, U.S.A., July 16–20 2006. AAAI Press.

[20] Evan A. Sultanik, Robert N. Lass, and William C. Regli. DCOPolis: A framework for simulating and deploying distributed constraint optimization algorithms. In Jonathan P. Pearce, editor, *Proceedings of the Ninth International Workshop on Distributed Constraint Reasoning (CP-DCR'07)*, Providence, RI, USA, September 23 2007.

[21] Weixiong Zhang, Guandong Wang, Zhao Xing, and Lars Wittenburg. Distributed stochastic search and distributed breakout: properties, comparison and applications to constraint optimization problems in sensor networks. *Journal of Artificial Intelligence Research (JAIR)*, 161(1–2):55–87, Jan. 2005.