FRODO:
a FRamework for Open/Distributed Optimization
Version 2.15

User Manual

PDF Version

Thomas Léauté Brammert Ottens
Radoslaw Szymanek

`https://frodo-ai.tech`

January 4, 2017

Contents

1 Legal Notice **4**

2 Introduction **4**

3 FRODO Architecture **4**
3.1 Communications Layer 5
3.2 Solution Spaces Layer 5
3.3 Algorithms Layer 6

4 How to Use FRODO **8**
4.1 Installation Procedure and Requirements 8
4.2 File Formats 8
4.2.1 Problem File Format 9
4.2.2 Agent Configuration File Format and Performance Metrics . 14
4.2.3 Support for Multi-Agent Systems 17
4.3 Simple Mode 18
4.3.1 With Graphical User Interface 18
4.3.2 Without GUI 19
4.4 Advanced Mode 19
4.4.1 Running in Local Submode 21
4.4.2 Running in Distributed Submode 22
4.5 API Mode, With or Without XCSP 25
4.6 How to Run Experiments 26
4.6.1 How to Start an Experiment 27
4.6.2 How to Produce Graphs 29
4.7 Troubleshooting 34

5 How to Extend FRODO **35**
5.1 Step 1: Writing the Agent Configuration File 35
5.2 Step 2: Implementing the Module(s) 37
5.2.1 The Interface `IncomingMsgPolicyInterface` 38
5.2.2 Sending Messages 38
5.2.3 The Module Constructor 39
5.2.4 Reporting Statistics 40
5.3 Step 3: Implementing a Dedicated *Solver* 41
5.4 Step 4: Testing 42

A Catalogue of Constraints **43**
A.1 Extensional Soft Constraint 44
A.2 Extensional Hard Constraints 45
A.3 Vehicle Routing Constraint 45
A.4 Intensional Hard Constraints 46
A.5 Intensional Soft Constraints 46
A.6 Global Constraints 48
A.6.1 *All Different* Constraint 48
A.6.2 *Cumulative* Constraint 48
A.6.3 *Diff2* Constraint 49
A.6.4 *Element* Constraint 49
A.6.5 *Weighted Sum* Constraint 50

B Catalogue of Benchmarks **51**
B.1 Graph Coloring 51
B.2 Meeting Scheduling 52
B.3 Random Max-DisCSP 53
B.4 Auctions and Resource Allocation Problems 53
B.5 Distributed Kidney Exchange Problems 54
B.6 Equilibria in Party Games 55
B.7 Vehicle Routing Problems (DisMDVRP) 55

1 Legal Notice

2 Introduction

FRODO is a Java open-source framework for distributed combinatorial optimization, initially developed at the Artificial Intelligence Laboratory (LIA) of École Polytechnique Fédérale de Lausanne (EPFL), Zwiterland. 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 this previous version, please refer to [26]. FRODO currently supports SynchBB [7], MGM and MGM-2 [19], ADOPT [21], DSA [37], DPOP [27], S-DPOP [28], MPC-Dis(W)CSP4 [32, 31], O-DPOP [29], AFB [4], MB-DPOP [30], Max-Sum [3], ASO-DPOP [25], P-DPOP [2], P^2-DPOP [13], $\mathbb{E}$[DPOP] [14, 16], Param-DPOP, $P^{3/2}$-DPOP [15], and DUCT [24]. FRODO also comes with a suite of benchmark problem generators, described in Appendix B.

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.

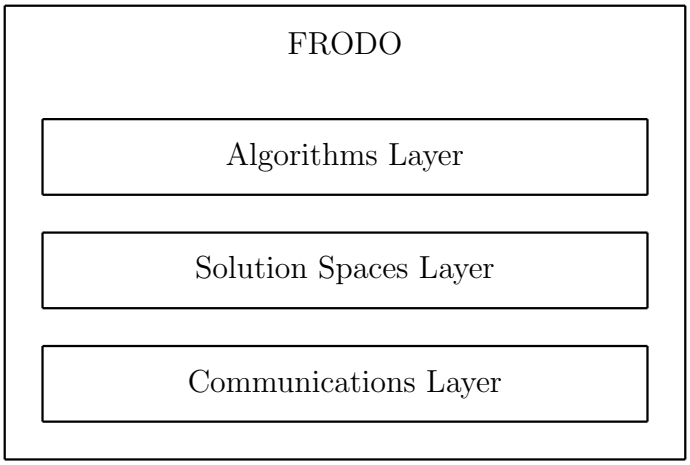

Figure 1: General FRODO software architecture.

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 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 provides several implementations of utility solution spaces, the simplest one being the *hypercube*. A *hypercube* is an extensional representation of a space in which each combination of assignments to variables is associated with a given utility (or cost). Infeasible assignments can be represented using special a special infinite utility/cost. Solution spaces can also be expressed *intensionally*, based on JaCoP constraints [9].

Solution spaces can be a way for agents to exchange information about their subproblems. For instance, in the *UTIL propagation* phase in *DPOP* [27], 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.

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 and maintenance.

FRODO currently supports the following algorithms: SynchBB [7], MGM and MGM-2 [19], ADOPT [21], DSA [37], DPOP [27], S-DPOP [28][1], MPC-Dis(W)CSP4 [32, 31], O-DPOP [29], AFB [4], MB-DPOP [30], Max-Sum [3], ASO-DPOP [25], P-DPOP [2], P^2-DPOP [13], $\mathbb{E}$[DPOP] [14, 16], Param-DPOP, $P^{3/2}$-DPOP [15], and DUCT [24]. 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. FRODO also provides a convenient algorithm to count the number of optimal solutions, which can be found in the package `frodo2.algorithms.dpop.count`.

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 three modules, which correspond to DPOP's three phases.

1. The *DFS Generation* module has the agents elect a root variable and exchange tokens so as to order all variables in the DCOP along a DFS tree rooted at the elected variable, following the algorithm in [12] (Section 4.4.2).

2. The *UTIL Propagation* module implements DPOP's traditional *UTIL propagation* phase [27], 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.

3. The *VALUE Propagation* module corresponds to DPOP's *VALUE propagation* phase [27], 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.

[1]The warm restart functionality in S-DPOP is currently only available through the API; see `frodo2.algorithms.dpop.restart.test.TestSDPOP` for a sample use.

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 elements:

- `frodo2.jar` is a Java 8-compliant executable JAR file;
- `LICENSE.txt` contains the FRODO license;
- `RELEASE_NOTES.txt` summarizes changes made from version to version;
- `FRODO_online_manual.html` automatically redirects to the FRODO online user manual;
- the `agents` folder contains sample agent configuration files for the multiple algorithms implemented in FRODO;
- the `experiments` folder contains examples of Python scripts to compare the performance of these algorithms on various problem domains (Section 4.6). These scripts have been developed and tested with Python 3.5. Optionally, if you want to make full use of FRODO's Python module to also produce graphs of your experimental results, you should install the `matplotlib` [8] Python module;
- a `lib` folder. The following third-party libraries should be downloaded separately from their respective distributors and put in the `lib` folder:
  - `jdom-2.0.6.jar` [10] is used to write and parse XCSP files;
  - `jacop-4.4.0.jar` [9] is required for intensional constraints;
  - `or-objects-3.0.3.jar` [22] is only required to run the DisMDVRP benchmarks.

In order to use FRODO's GUI, it is also necessary to separately install Graphviz [6] 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 [23], with 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. The resulting XCSP format is a superset of the XDisCSP 1.0 format used in the DisCHOCO 2 platform [34]; this makes it possible to use DisCHOCO as a benchmark problem generator for FRODO. DisCHOCO supports multiple classes of benchmarks, including meeting scheduling, sensor networks, graph coloring, random Max-DisCSPs, and n-queens. Depending on the XCSP parser used (`XCSPparser` or `JaCoPxcspParser`), FRODO supports a restricted subset, and an extended subset of XCSP, respectively.

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

  <agents nbAgents="3">
    <agent name="agentX" />
    <agent name="agentY" />
    <agent name="agentZ" />
  </agents>

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

  <variables nbVariables="3">
    <variable name="X" domain="three_colors" agent="agentX" />
    <variable name="Y" domain="three_colors" agent="agentY" />
    <variable name="Z" domain="three_colors" agent="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 (restricted XCSP subset).

Restricted XCSP Subset: Extensional Soft Constraints Only Figure 2 shows an example FRODO XCSP file, using the restricted XCSP subset supported by the `XCSPparser`. The file consists of five main sections:

1. The `<agents>` section defines the agents in the DCOP. This is an extension made to the XCSP 2.1 format [23] in order to adapt it to distributed problems.

2. 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.

3. 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 `agent` field is an extension made to the XCSP 2.1 format [23] in order to adapt it to distributed problems.

4. 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, the `XCSPparser` currently only supports the *soft* relations (`semantics = "soft"`), which list possible utility values (or cost values, depending on whether the attribute `maximize` of the `presentation` tag 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 cost value $+\infty$ to all assignments in which the two variables are equal, and a cost 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 the special infinite costs `infinity`, and resort to the value `1` instead, such that the cost of a particular solution corresponds to its number of constraint violations.

5. 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.

Each constraint may have an optional `agent` attribute; when present, only the referred agent knows the constraint. If set to `"PUBLIC"`, the constraint is known to *all* agents (even those not involved in the constraint).

Restricted XCSP Subset with Support for StochDCOP FRODO also supports a variant of the XCSP format that can be used to model DCOPs under Stochastic Uncertainty (StochDCOPs) [16], which include *random variables* that model sources of uncertainty in the problem. Expressing a StochDCOP involves the following two extensions of the previously described XCSP format, as illustrated in Figure 3.

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

  <agents nbAgents="2">
    <agent name="agentX" />
    <agent name="agentY" />
  </agents>

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

  <variables nbVariables="3">
    <variable name="X" domain="three_colors" agent="agentX" />
    <variable name="Y" domain="three_colors" agent="agentY" />
    <variable name="Z" domain="three_colors" type="random" />
  </variables>

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

  <probabilities nbProbabilities="1">
    <probability name="PROB" arity="1" nbTuples="2" semantics="soft" defaultProb="0.5">
      0.25: 1 | 2
    </probability>
  </probabilities>

  <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" />
    <constraint name="Z_prob" arity="1" scope="Z" reference="PROB" />
  </constraints>
</instance>
```

Figure 3: An example StochDCOP corresponding to the graph coloring problem in Figure 2, but in which variable Z is random.

1. Random variables are identified by `<variable>` elements in which the `agent` attribute is replaced with a `type` attribute, whose value must be set to `"random"`.

2. For each random variable there must be a `<constraint>` element whose scope only includes that random variable, and whose `reference` attribute is equal to the name of a `<probability>` element. These `<probability>` elements are listed in a new `<probabilities>` section; the format of the `<probabilities>` section and the `<probability>` elements is almost the same as the `<relations>` section and `<relation>` element respectively, except that the attribute `nbRelations` is replaced with `nbProbabilities`, `defaultCost` is replaced with `defaultProb`, and the values of the probabilities for a given variable must sum up to 1.

Extended XCSP Subset: Adding Intensional Constraints The parser `JaCoPxcspParser` makes it possible to express constraints using a much richer syntax, including intensional constraints based on predicates, functions, and global constraints. Figure 4 shows the same FRODO XCSP file as in Figure 2, but this time, using this extended XCSP subset. There are two differences with respect to the extensional representation in Figure 2:

1. The `<predicates>` element is the *intensional, hard* equivalent of the extensional, soft `<relations>` element. Each predicate declares a whitespace-delimited list of `parameters`, each preceded by its type (currently, only `int` is supported). The `functional expression` is a logical expression over the parameters that defines the constraint. The following functions are currently supported, and can be recursively combined to compose complex expressions: `abs()` (absolute value), `neg()` (opposite), `add(,)` (binary addition), `sub(,)` (subtraction), `mod(,)` (modulo), `mul(,)` (binary multiplication), `div(,)` (integer division), `pow(,)` (exponentiation), `min(,)` (binary minimum), `max(,)` (binary maximum), `eq(,)` ($=$), `ne(,)` ($\neq$), `ge(,)` ($\geq$), `gt(,)` ($>$), `le(,)` ($\leq$), `lt(,)` ($<$), `not()` (logical *not*), `and(,)` (binary logical *and*), `or(,)` (binary logical *or*), `xor(,)` (binary logical *xor*), `if(,,)` (*if-then-else*), `iff(,)` (binary equivalence).

2. Each `<constraint>` element declares a list of `parameters`, which must be either variables or constants, in the order corresponding to the order of the parameters of the referred predicate.

Note that the `JaCoPxcspParser` still supports extensional, soft `relations`; it also supports *intensional*, soft `functions`, described in Section A.5 (replacing `nbPredicates` with `nbFunctions`).

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

  <agents nbAgents="3">
    <agent name="agentX" />
    <agent name="agentY" />
    <agent name="agentZ" />
  </agents>

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

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

  <predicates nbPredicates="1">
    <predicate name="NEQ">
      <parameters> int X1 int X2 </parameters>
      <expression>
        <functional> ne(X1, X2) </functional>
      </expression>
    </predicate>
  </predicates>

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

Figure 4: An example FRODO XCSP file (extended XCSP subset).

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 5 presents a sample agent configuration file.

Performance Metrics FRODO supports the following performance metrics:

- **Numbers and sizes of messages 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;
  - the total number of messages sent and received by each agent;
  - the total amount of information sent (in bytes), sorted by message type;
  - the total amount of information sent/received by each agent (in bytes);
  - the size (in bytes) of the largest message, 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.

- **Non-Concurrent Constraint Checks (NCCCs)** [5]: To activate the counting of NCCCs, set the attribute `countNCCCs` to `"true"` in the parser definition. **Note:** the `JaCoPxcspParser` currently does not support NCCCs, because the notion of a constraint check and how to count them is ill-defined in JaCoP.

- **Simulated time** [33]: To activate the simulated time metric, set the attribute `measureTime` to `"true"` in the agent's configuration file. Simulated time is enabled by default, and should only be disabled if the platform is such that each agent gets a dedicated processor/core. When enabled, FRODO proceeds as follows.

  Each agent has an internal clock. When it sends a message, the agent appends to it a timestamp that indicates the time at which the message was sent, according to the agent's internal clock. When it receives a message, if

```
<agentDescription className = "frodo2.algorithms.SingleQueueAgent"
  measureTime = "true" measureMsgs = "false" >

  <parser parserClass = "frodo2.algorithms.XCSPparser"
          displayGraph = "false"
          utilClass = "frodo2.solutionSpaces.AddableInteger"
          countNCCCs = "false" />

  <modules>
    <module className = "frodo2.algorithms.varOrdering.dfs.DFSgenerationParallel"
      reportStats = "true" >
      <rootElectionHeuristic
        className = "frodo2.algorithms.heuristics.ScoringHeuristicWithTiebreaker" >
        <heuristic1
          className = "frodo2.algorithms.heuristics.MostConnectedHeuristic" />
        <heuristic2
          className = "frodo2.algorithms.heuristics.ScoringHeuristicWithTiebreaker" >
          <heuristic1
            className = "frodo2.algorithms.heuristics.SmallestDomainHeuristic" />
          <heuristic2
            className = "frodo2.algorithms.heuristics.VarNameHeuristic" />
        </heuristic2>
      </rootElectionHeuristic>
      <dfsGeneration className = "frodo2.algorithms.varOrdering.dfs.DFSgeneration" >
        <dfsHeuristic className =
          "frodo2.algorithms.varOrdering.dfs.DFSgeneration$ScoreBroadcastingHeuristic">
          <scoringHeuristic
            className = "frodo2.algorithms.heuristics.ScoringHeuristicWithTiebreaker" >
            <heuristic1
              className = "frodo2.algorithms.heuristics.MostConnectedHeuristic" />
            <heuristic2
              className = "frodo2.algorithms.heuristics.SmallestDomainHeuristic" />
          </scoringHeuristic>
        </dfsHeuristic>
      </dfsGeneration>
    </module>

    <module className = "frodo2.algorithms.dpop.UTILpropagation"
      reportStats = "true" />

    <module className = "frodo2.algorithms.dpop.VALUEpropagation"
      reportStats = "true" />
  </modules>

</agentDescription>
```

Figure 5: Example of a FRODO agent configuration file, corresponding to the classical version of DPOP.

the message's timestamp is later than the agent's internal clock, the agent updates its clock to match the timestamp. When the algorithm terminates, its runtime is then defined as the latest time indicated by any agent's clock.

This is the same mechanism as the one used by the NCCC metric, except that instead of counting constraint checks, the agent counts time. However, unlike for the NCCC metric, to simulate the situation in which each agent would be running on a dedicated processor/core, it is necessary to make sure that, at any point in time, only a single agent is active with its clock ticking, while all other agents are "sleeping" with their clocks "frozen."

To achieve this, FRODO uses a central mailer that collects all messages sent by the agents into an outbox, and only delivers them one at a time, to each of its destinations in turn (if the message has multiple destinations). An agent's clock is only ticking when it is busy processing a message received; when the agent is done processing the message, its clock is frozen, and the control is returned to the central mailer, which can then deliver the next message. To enforce causality, the central mailer delivers the messages by increasing order of their timestamps.

Note that this implementation slightly differs from DCOPolis' implementation [33], in which messages are processed in *batches*: the central mailer retrieves all outgoing messages from its outbox, puts them in a temporary, timestamp-ordered queue, and delivers the messages from this temporary queue in sequence. The disadvantage of DCOPolis' approach is that, while the ordering by timestamp is enforced inside each batch, it may be violated from one batch to the next, and therefore it is possible for an agent to receive a message from agent a_1 with timestamp t_1 *after* another message from agent a_2 with timestamp $t_2 > t_1$. This should not happen if message delivery is assumed instantaneous, which is the assumption made by the simulated time metric since it only measures computation time, and excludes message delivery time.

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 5), the DFS Generation module can report the DFS tree that is computed and used by DPOP, using the DOT format [6], 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 `frodo2.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 [6] be preliminarily installed as described in Section 4.1.

4.2.3 Support for Multi-Agent Systems

FRODO provides preliminary, limited support for more general Multi-Agent Systems (MAS), in which there may be multiple types of agents, performing different algorithms. To enable this feature, the agent configuration file should be modified as in Figure 6, declaring one `<modules>` element for each agent type. The `className` of each module should refer to a class that implements the interface `IncomingMsgPolicyInterface<String>`, as documented in Section 5.2.

```
<agentDescription className = "frodo2.algorithms.SingleQueueAgent" >

  <parser parserClass = "frodo2.algorithms.MASparser"
          probDescClass = "frodo2.solutionSpaces.MASProblemInterface" />

  <modules agentType = "agentType1" >
    <module className = "..." />
    ...
  </modules>

  <modules agentType = "agentType2" >
    <module className = "..." />
    ...
  </modules>
  ...
</agentDescription>
```

Figure 6: An agent configuration file declaring multiple agent types.

The user should subclass `MASparser` and `MASProblemInterface` as necessary, depending on the MAS problem class considered. The problem file must include a description of each agent's subproblem, as illustrated in Figure 7. For convenience, FRODO makes it possible to specify each agent's subproblem in a separate file; this can be achieved as in Figure 8, where the root element of `agent1.xml` is the corresponding `<agent>` element from Figure 7.

```
<MASproblem numberOfAgents = "n" >
  <agent type = "agentType1" name = "nameOfAgent1" >
    <problem>
      ...
    </problem>
  </agent>
  ...
</MASproblem>
```

Figure 7: Structure of a MAS problem file.

```
<MASproblem numberOfAgents = "n"
  xmlns:xi="http://www.w3.org/2001/XInclude"
  xml:base="file:folder/containing/the/included/files">

  <xs:include href="agent1.xml"/>
  ...
</MASproblem>
```

Figure 8: Structure of a MAS problem file using XInclude.

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. 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 `frodo2.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:

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. If the path to the problem file is also omitted, FRODO generates and solves a random problem using DPOP; this requires JUnit to be on the classpath. 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 9. It allows the user to specify (and, optionally) edit a problem file in XCSP format, to render the corresponding constraint graph, 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 10. To render these graphs, FRODO uses Graphviz [6], which must be preliminarily installed as described in Section 4.1.

4.3.2 Without GUI

The simple mode without GUI is launched using the `main` method of the class `AgentFactory` in the package `frodo2.algorithms`, which can be achieved using the following command, called from within the directory containing the FRODO JAR file:

```
java -cp frodo2.jar frodo2.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 TCP. In this mode, each computer runs a *daemon*, which initially waits for a centralized *controller* to tell it to start the solving the problem(s). 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.

Using the advanced mode, it is possible to set up batch experiments. The configuration file (see Figure 11) can contain a list of problems that will be solved

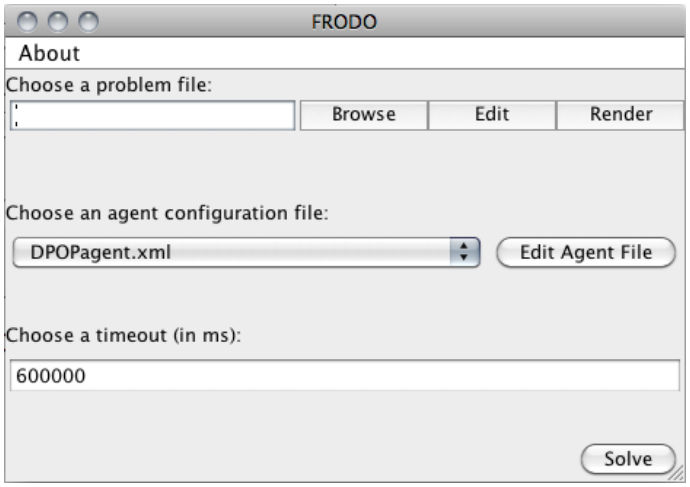

Figure 9: FRODO's main GUI window.

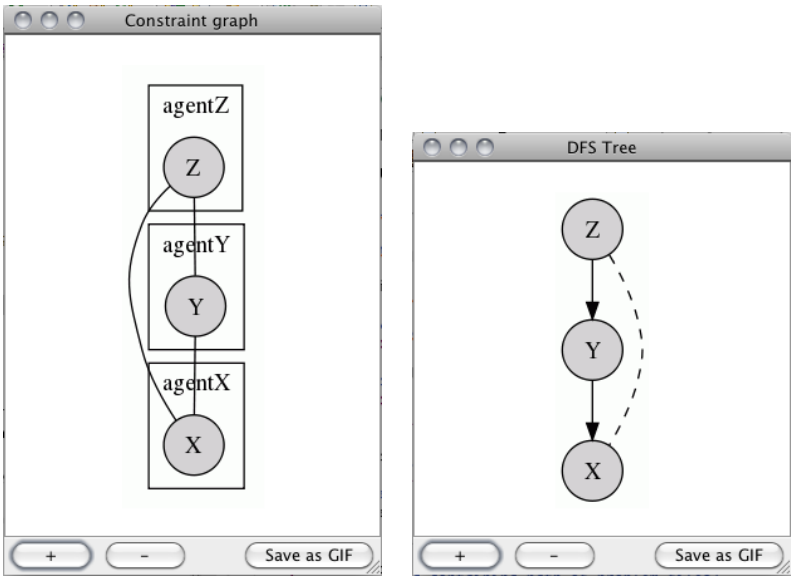

Figure 10: The constraint graph and DFS tree rendered by FRODO's GUI for the problem instance in Figure 2.

sequentially by the agents. The agent configuration to be used is defined by the field `agentName` in the `agentDescription` element, which should refer to a file that is distributed with FRODO inside `frodo2.jar`. It is also possible to replace the `agentName` field with `fileName = "agent.xml"`, where `agent.xml` is the name of a file outside `frodo2.jar` 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).

IMPORTANT NOTE: The advanced mode does not support the simulated time metric (Section 4.2.2). Furthermore, it should only be used on a (distributed or centralized) platform such that each agent gets a dedicated processor/core.

```
<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 11: Example of a configuration file for FRODO's advanced mode.

4.4.1 Running in Local Submode

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

```
java -cp frodo2.jar frodo2.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 passes the `open` command to the controller prompt:

`Controller` > `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 11. To run the experiments, simply give the `start` command:

```
Controller > start
```

When all the experiments are finished, the controller can be exited by giving the `exit` command:

```
Controller > exit
```

4.4.2 Running in Distributed Submode

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

```
java -cp frodo2.jar frodo2.controller.Controller
```

To set the work directory one can 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 `frodo2.daemon`, using the following command from within the directory containing `frodo2.jar`:

```
java -cp frodo2.jar frodo2.daemon.Daemon
```

To set the work directory one can use the `-workdir` argument. The IP address of the controller can either be given with the command-line argument `-controller`

ip_address, or by issuing the command at the daemon console prompt:

`Daemon daemon@hostname:port > controller` *ip_address*

The port number used for the controller is 3000. The default port number used for the daemon is 25000, but this can be changed using the command-line argument `-daemonport` *port_number*. Each agent spawn by the daemon will be assigned an increment of this port number, the first agent getting port *port_number*+10. 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:

`Controller > get daemons`

At this point, there are two possible options to tell FRODO which agent configuration file and problem file(s) to use, depending on whether the controller is omniscient (i.e. it knows the overall problems) or whether each daemon can specify its own subproblem.

Omniscient controller In the case when the controller is omniscient, the configuration file (Figure 11) should be provided to the controller using the `open` command. Each problem in the configuration file must describe the overall problem for all agents. The algorithms can then be launched using the `start` command.

`Controller > open` *configuration_file*
`Controller > start`

The following then happens:

1. The controller extracts each agent's subproblem from the overall problem, and sends these subproblems to the daemons, in a round robin fashion. If there are more agents than daemons, each daemon might receive more than one subproblem;

2. Each daemon opens its subproblem(s) and checks which agents it needs to be able to send messages to. For each such agent that is not also run locally by the daemon, the daemon sends a request to the controller for that agent's IP address and port number;

3. The controller communicates the requested IP addresses and port numbers to the daemons;

4. Once the agents know how to communicate with each other, the controller tells the daemons to start executing the chosen DCOP algorithm. From that point onwards, the agents communicate with each other directly, i.e. the inter-agent communication is fully decentralized (it does not go via the controller);

5. Each agent reports statistics about the execution of the algorithm to the controller, and also reports to the controller when it has finished executing the algorithm.

Non-omniscient controller In this setting, the controller is only used as a "white pages" service that the daemons register to and that the agents can use to look up how to connect to other agents. Contrary to the case of the omniscient controller, the problems themselves are not known to the controller; each agent's local subproblems are loaded by the daemons themselves:

`Daemon daemon@hostname:port` > `open` *configuration_file*

Each agent's local subproblem must indicate the name of the agent corresponding to the subproblem; this must be documented as an attribute `self` of the `<agents>` tag. The XCSP extract below illustrates what this would look like for Agent *agentX* from the problem instance in Figure 2.

```
<agents nbAgents="3" self="agentX">
  <agent name="agentX" />
  <agent name="agentY" />
  <agent name="agentZ" />
</agents>
```

The following Java command can be used to extract each agent's subproblem from an overall problem instance:

```
java -cp frodo2.jar frodo2.algorithms.XCSPparser -split problemFile.xcsp
```

Once each daemon has been given its subproblem instance, the controller is then used to tell the agents to start solving the problem:

`Controller` > `start`

Once the `start` command has been issued to the controller, the same steps happen as in the case of the omniscient controller, except that Step 1 is skipped, and in Step 5, the agents report statistics locally rather than to the controller.

4.5 API Mode, With or Without XCSP

It is also possible to interact with FRODO directly through its Java API. This is particularly recommended for users who would not want to have to write XCSP problem instance files. 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.jdom2.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. FRODO's benchmarking problem generators usually also provide methods that directly produce `Document` objects. Alternatively, if you do not want to have to deal with XCSP, the solvers also provide `solve` methods that take in objects implementing `DCOPProblemInterface`, such as:

```
public Solution solve (DCOPProblemInterface problem,
                       int nbrElectionRounds) { ... }
```

To construct an object that implements `DCOPProblemInterface`, it is possible to use the `Problem` class. Variables can be manually added to a problem using the method `Problem.addVariable(String, String, V[])`, and constraints using the method `Problem.addSolutionSpace(UtilitySolutionSpace)`. A simple example of a `UtilitySolutionSpace` is a `Hypercube`. The spaces supported in FRODO and how to generate them are described in detail in Appendix A.

The second input `nbrElectionRounds` to the `solve` method 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.

If you intend to run experiments that involve measuring and comparing the runtimes of various 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.

Finally, it is also possible to run FRODO in distributed mode through the API, via the `DistributedSolver`. Have a look at the `DistributedSolverDemo` (which should be run instead of the `Controller` from Section 4.4.2) for an example of how to do this.

4.6 How to Run Experiments

The `experiments` folder that comes with FRODO provides examples of how to run experiments to compare the performance of various algorithms on various problem domains. These examples consist in Python scripts that make use of the `frodo2` Python module that is included in `frodo2.jar`.

There are several reasons why we strongly recommend running experiments using scripts outside of Java (in this case, in Python). First, initializing the JVM takes time, and if one were to call the algorithms one after another in a loop inside Java, only the first algorithm(s) would have to pay the price of the JVM initialization, and the experiments would not be fair; restarting a new JVM for each algorithm on each problem instance addresses this undesirable experimental bias by having each algorithm equally pay the price of JVM initialization. Second, if only one JVM were used, this JVM would tend to age as the experiment progresses, and algorithms could become slower and slower. Finally, if the experiment pushes some of the algorithms to their limits (which we recommend they should), then on some problem instances some algorithms could end up timing out without properly releasing all their resources, or the JVM could even run out of memory and abruptly terminate. To summarize, restarting a fresh JVM for each algorithm on each problem instance guarantees that what has happened during one run will not influence the performance of the JVM during the following run.

In order to make use of the `frodo2` Python module, you must first import it using the following code, which assumes that your Python script lives and is started inside the `experiments` folder.

```
import sys
sys.path.append("../frodo2.jar/frodo2/benchmarks")
import frodo2
```

Section 4.6.1 first describes how to format the inputs to the `run` function that runs the experiments. Section 4.6.2 then describes how to report the experimental results in graphs.

4.6.1 How to Start an Experiment

To start an experiment, you should call the `run` method by passing it 8 arguments as follows. The nature and contents of each argument is documented below.

```
frodo2.run(java, javaParams, generator, genParams,
           nbrProblems, algos, timeout, output)
```

`java` is a string that contains the name (possibly prefixed by a path) of the Java executable; for instance `"java"` or `"java.exe"`.

`javaParams` is a list of strings, each string being one argument to be passed to the Java executable. This includes for instance the classpath, and how much memory you want to allocate to the JVM, as illustrated below.

```
javaParams = ["-Xmx2G", # maximum 2GB of Java heap space
              "-classpath", "../frodo2.jar"]
```

`generator` is the package-prefixed name of the Java class containing the `main` method that creates a random problem instance. For example, to run an experiment on graph coloring problems, `generator` should be set to `"frodo2.benchmarks.graphcoloring.GraphColoring"`.

`genParams` is a list of arguments to be passed to the `main` method of the problem generator. Each entry in the list can be of three different types:

- a string will be passed directly as an argument to the `main` method;
- a number will be first converted to a string, and then passed to the `main` method;
- a list of numbers will be iterated over by the `run` function, calling the problem generator by passing it each number (automatically converted to a string).

Continuing on the example of graph coloring experiments, the following setting will create random problems of varying numbers of nodes (from 3 included to 11 excluded), with a constant density of 0.4, a constant tightness of 0.0 (initially, all colors are allowed for all nodes), and 3 colors.

```
genParams = [list(range(3, 11)), # from 3 to 10 nodes
             .4,                 # the density
             0.0,                # the tightness
             3]                  # the number of colors
```

`nbrProblems` is the number of problem instances that will be created for each combination of parameters passed to the problem generator. To be able to compute confidence intervals for the median, this number should be a least 6, but to get disjoint confidence intervals that allow you to draw statistically significant conclusions, you might have to set it to 101 or more, depending on the variance and differences in the performance of the algorithms.

`algos` is a list of algorithms, where each algorithm is defined as a list of 4 strings, and an optional additional 5th parameter:

1. the (unique) name that will be used to refer to the algorithm in the experimental results;
2. the package-prefixed Java class name of the solver for that algorithm;
3. the path-prefixed name of the agent configuration file;
4. the name of the XCSP file created by the problem generator;
5. (optional) the Java parameters, using the same format as `javaParams`.

For instance, to compare the performance of DPOP [27] and SynchBB [7] on graph coloring problems, you should set `algos` to the following.

```
algos = [["DPOP",
          "frodo2.algorithms.dpop.DPOPsolver",
          "../agents/DPOP/DPOPagent.xml",
          "graphColoring.xml"],
         ["SynchBB",
          "frodo2.algorithms.synchbb.SynchBBsolver",
          "../agents/SynchBB/SynchBBagent.xml",
          "graphColoring.xml"]]
```

`timeout` is the number of seconds after which each algorithm will be interrupted if it has not terminated yet.

`output` is a string containing the (possibly path-prefixed) name of the output semicolon-separated CSV file to which the experimental results will be written.

Once an experiment has been started, it can be interrupted by passing `CTRL+C` to the Python interpreter. However, when you do so, the experiment will not be abruptly interrupted; instead the `frodo2` module will wait until all algorithms have finished running on the current problem instance before stopping the experiment. This delayed-interruption mechanism is used to avoid introducing an experimental

bias [11]: if you stopped an experiment at any random time, you would be more likely to stop it during a long-lasting run than a short-lasting run (the probability of interrupting a run that would have lasted 2 min is twice that of interrupting a run that would have lasted 1 min). This would introduce a bias in the results, making the interrupted algorithm appear to perform better than it really does.

If you still want to abruptly interrupt a running experiment, you can pass `CTRL+C` twice to the Python interpreter; the experimental results already gathered for some algorithms on the current problem instance will be discarded. Notice also that this delayed-interruption functionality may not be available if you run your Python script from within an IDE rather than from the command line. For instance, in the Eclipse IDE, pressing the red stop button will abruptly kill the Python interpreter rather than pass it an interruption signal.

4.6.2 How to Produce Graphs

The `run` function of the `frodo2` Python module will record experimental results in a CSV file whose format is documented in below. You can read the raw data in that file yourself to report the results of your experiment, or you can use the `plot` or `plotScatter` functions in `frodo2` to consolidate this raw data.

Format of the Output File The output file is a semicolon-separated CSV file that can be easily imported into your favorite spreadsheet program. The first line in the file contains the headers, and each subsequent line contains the results of running one algorithm on one problem instance. The columns are the following.

- The first column contains the name of the algorithm, as defined in the `algos` argument passed to the `run` function.
- The second column indicates whether the algorithm timed out, where `0` indicates that it terminated without timing out, and `1` means it was interrupted after timing out.
- The third column contains the unique name of the problem instance. This can be useful if you want to compare pairwise the performance of two algorithms on the same problem instances.
- The following columns contain statistics about the problem instance, as documented by the problem generator inside the XCSP file it produced. In the example of the graph coloring problem generator, these statistics include the number of colors, the maximum degree of the graph, its number of disconnected components, its number of nodes, its density...

- The final columns contain statistics about how the algorithm performed on that problem instance, such as its number of NCCCs [5], its simulated time [33], the number, maximum size and total size of the messages exchanged, the treewidth of the pseudo-tree used (when applicable), and the cost of the solution found (set to `"NaN"` — meaning "Not a Number" — if the algorithm timed out).

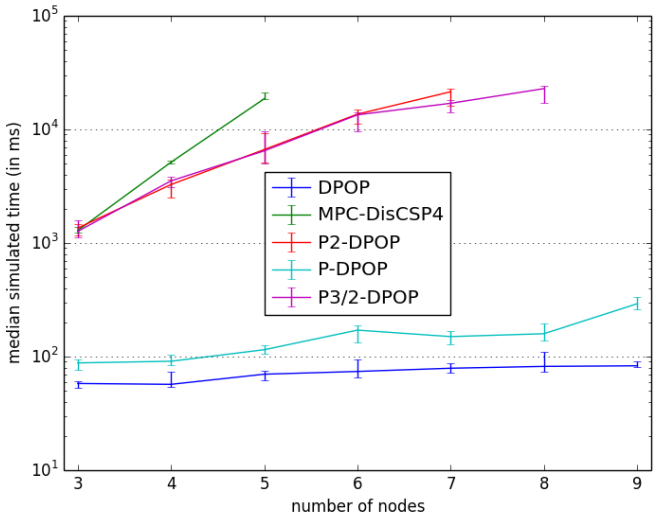

Figure 12: Sample graph produced by the `plot` function using the `matplotlib` module, for graph coloring problems with 3 colors, a density of 0.4, a timeout of 30 seconds and 2GB of Java heap space

The `plot` Function In addition to the `run` function, the `frodo2` Python module also provides a `plot` function that consolidates the raw data written by the `run` function. This function can be called as followed:

```
frodo2.plot(resultsFile, xCol, yCol, block, ylog)
```

where:

`resultsFile` is a string containing the (possibly path-prefixed) name of the CSV file containing the raw experimental results written by the `run` function;

`xCol` is the index of the column from that file containing the data that should be used for the x axis (by convention, the first column has index 0);

`yCol` is the index of the column containing the data for the y axis;

`block` is an optional parameter (default value is `True`) that controls whether the `plot` function should block until the figure window has been closed;

`ylog` is an optional parameter (default value is `True`) that controls whether the `plot` function should use a log scale for the y axis.

The `plot` function consolidates the raw data in the `yCol`-th column by computing its median value and the corresponding 95% confidence interval. When the `matplotlib` [8] Python module is available on the Python path, the `plot` function will directly draw a graph such as in Figure 12. Missing data points for a given algorithm corresponds to problem sizes for which the algorithm timed out on more than 50% of the problem instances (i.e. the median is infinite). Please refer to the `matplotlib` documentation [8] if you want to customize the graph.

If the `matplotlib` module is not available, the `plot` function will instead write the consolidated results to another semicolon-separated CSV file, whose format is documented in Table 1. The file can then be imported into your favorite spreadsheet program to produce a graph manually.

Table 1: Format of the CSV file output by the `plot` function

`y axis label:`	$yLabel$				
$xLabel$	$algo_1$	$algo_1^-$	$algo_1^+$	$algo_2$	...
`1.0`	`10.3`	`0.5`	`0.4`	`14.2`	...
...	...	...	...	...	...

$yLabel$ is the title for the y axis;

$xLabel$ is the title for the x axis, and the header for the column containing the x values (in Table 1, the first value is $x = 1.0$);

$algo_i$ is the name of the ith algorithm, and the header for the column containing the y values for this algorithm (in Table 1, the value corresponding to $x = 1.0$ for $algo_1$ is $y = 10.3$);

$algo_i^-, algo_i^+$ are the headers for the columns that contain the bounds for the confidence interval, such that, if y_i is the value in the column $algo_i$, its confidence interval is $\left[y_i - algo_i^-, y_i + algo_i^+\right]$ (in Table 1, the confidence interval for the data point $y = 10.3$ is $[10.3 - 0.5, 10.3 + 0.4] = [9.8, 10.7]$).

The `plotScatter` Function The `plotScatter` function can be used to compare the performance of two algorithms against one another on the same problem instances. This function can be called as follows:

```
frodo2.plotScatter(resultsFile, xAlgo, yAlgo, metricsCol,
                   timeouts, block, loglog)
```

where:

`resultsFile` is a string containing the (possibly path-prefixed) name of the CSV file containing the raw experimental results written by the `run` function;

`xAlgo` is the name of the algorithm whose performance should be on the x axis;

`yAlgo` is the name of the algorithm whose performance should be on the y axis;

`metricsCol` is the index of the column in the results file corresponding to the chosen performance metric (the first column has index 0);

`timeouts` if `True` (default), timeouts will be plotted. Should be set to `False` to zoom in on the data points corresponding to problem instances on which both algorithms terminated without timeout, making the graph more readable;

`block` is an optional parameter (default value is `True`) that controls whether the `plotScatter` function should block until the figure window has been closed;

`loglog` optionally specifies whether the graph should use log-log scales (corresponding to the default value `True`) or natural scales.

When the `matplotlib` [8] Python module is available on the Python path, the `plotScatter` function will directly draw a graph such as in Figure 13. Please refer to the `matplotlib` documentation if you want to customize the graph. If the `matplotlib` module is not available, the `plotScatter` function will instead write the results to another semicolon-separated CSV file, whose format is documented in Table 2. The file can then be imported into your favorite spreadsheet program to produce a graph manually.

Table 2: Format of the CSV file output by the `plotScatter` function

name of the performance metrics	
name of the x algorithm	*name of the y algorithm*
`1.2`	`1.4`
`2.4`	`3.6`
...	...

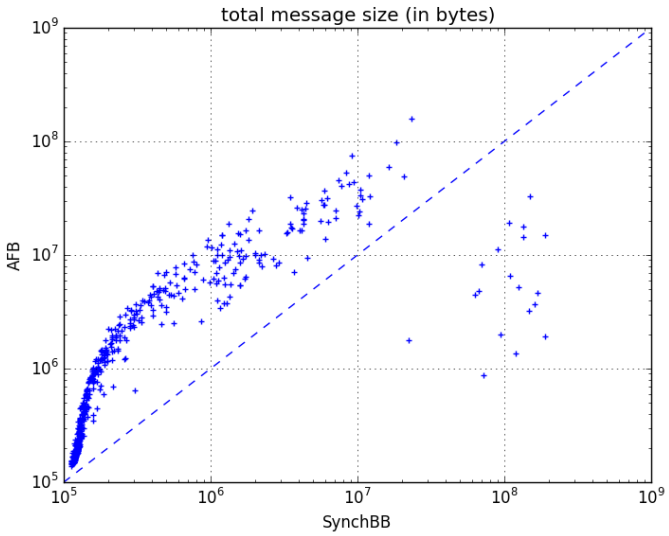

Figure 13: Sample graph produced by the `plotScatter` function using the `matplotlib` module, comparing the information exchanged by AFB vs. SynchBB (max-width, min-domain heuristic) on random Max-DisCSP problem instances of 10 variables, domain size 10, density 0.4, and tightness varying from 0.4 to 0.99.

Important Note on Reporting Performance Notice that the `plot` function reports the *median* performance, not the *average* or *expected* performance. There is a very good rationale behind this [11]. First, good experimental results should ***always*** include confidence intervals, which are a guarantee that the results are statistically significant. A 95% confidence interval means that there is a 95% probability that the median performance of the given algorithm is contained in the interval. In particular, when comparing two algorithms, if their confidence intervals are disjoint, you can claim with 95% confidence that the median performance of one algorithm is higher than the median performance of the other.

Without confidence intervals, the conclusions drawn from your results might be flawed, because it could be that you have run the algorithms on an insufficient number of problem instances, and that running them on more problem instances would have produced different results and different conclusions. By computing and reporting confidence intervals, and letting your experiments run until the intervals are disjoint, you can draw conclusions that are more likely to be correct.

Furthermore, while you could just report the *average* performance and its standard deviation, these results would be both less *robust* and less *significant* than reporting the *median* performance and its confidence interval. First, the results would be less robust, because the value of the *average runtime* performance de-

pends on the arbitrary value you have chosen for the timeout. If the algorithm needs a virtually infinite amount of time to solve some of the problem instances, then increasing the arbitrary timeout threshold will not help, and will result in an artificially increased value for the *average runtime*. In contrast, as long as the algorithm times out in less than 50% of the cases, the value of the *median runtime* will *not* depend on the arbitrary value you have chosen for the timeout.

Reporting the *median* performance rather than the *average* performance is also more significant, because of a limitation of the *law of large numbers*. The whole philosophy behind running algorithms on sample problem instances is that the law of large numbers guarantees that, as you increase the number of problem instances, your performance results will asymptotically converge to the true performance of the algorithm on the given problem class. The issue is that the law of large numbers does not apply to *heavy-tailed distributions*, which is the case of your raw data distribution as soon as the algorithm times out on some instances. In that case, reporting the *average* performance and its standard deviation only informs about the performance of the algorithm *on the problem instances you have used*, and is not guaranteed to reflect the true average performance of the algorithm. In contrast, even in the case of heavy-tailed distributions, the *median* performance and its confidence interval *does* converge to the true median performance of the algorithm, because the median value is robust to the heavy tail (i.e. to the timeouts).

4.7 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.

You can also display the messages exchanged by all agents by adding the `MessageDebugger` module to the agent configuration file, as illustrated below. This can degrade the performance of the algorithms, and should only be used for debugging purposes.

```
<module className = "frodo2.algorithms.test.MessageDebugger"
        hideSystemMessages = "true" />
```

Various helpful tools (such as a bug tracker and a support request tracker) are also available on FRODO's SourceForge website. 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 5; we now illustrate step-by-step how to write the configuration file for SynchBB. The general structure of an agent configuration file is given below (in XML format).

```
<agentDescription className = "frodo2.algorithms.SingleQueueAgent"
  measureTime = "true"
  measureMsgs = "false" >

  <parser parserClass = "frodo2.algorithms.XCSPparser"
    displayGraph = "false" />

  <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 = "frodo2.algorithms.varOrdering.election.VariableElection"
  nbrSteps = "150" >

  <varElectionHeuristic
    className = "frodo2.algorithms.heuristics.ScoringHeuristicWithTiebreaker" >
    <heuristic1
      className = "frodo2.algorithms.heuristics.SmallestDomainHeuristic" />
    <heuristic2
      className = "frodo2.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 *max width* heuristic [35] 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 = "frodo2.algorithms.varOrdering.linear.LinearOrdering"
  reportStats = "true" />
```

Other DCOP algorithms based on a pseudo-tree ordering of the variables instead of a total ordering should reuse the `DFSgenerationParallel` module implemented for DPOP (Figure 5).

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 5). For SynchBB, which is a simpler, single-phase algorithm, a single module is sufficient (Figure 14).

```
<module className = "frodo2.algorithms.synchbb.SynchBB"
  reportStats = "true"
  convergence = "false" />
```

Figure 14: XML fragment describing the parameters of the `SynchBB` module.

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 one additional parameter: `convergence` is a boolean attribute that specifies 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).

Overriding the Message Types of Existing Modules In some circumstances, in order to reuse existing modules, it can be necessary to modify the types of the messages they listen to and exchange. An example of such a situation is that of P^2-DPOP [13], which uses two different modules to elect a root variable: `SecureVarElection` elects an initial, temporary root, and `SecureRerooting` elects the true root used at each iteration of the algorithm. P^2-DPOP also uses the module `DFSgeneration` to construct pseudo-trees, which normally listens to the output of `SecureVarElection`, but must be made to listen to the output of `SecureRerooting` instead. Another example situation would be one in which a new, custom module has to be placed between two existing modules, such that the new module intercepts the outputs of the first module and modifies them before passing them to the second. FRODO provides a simple way to achieve this, via the agent configuration file. For instance, P^2-DPOP declares its module `DFSgeneration` as follows (only showing the relevant XML elements).

```
<module className="frodo2.algorithms.varOrdering.dfs.DFSgeneration">
  <messages>
    <message name="ROOT_VAR_MSG_TYPE"
      value="OUTPUT"
      ownerClass="frodo2.algorithms.dpop.privacy.SecureRerooting"/>
  </messages>
</module>
```

This enforces that, before the module `DFSgeneration` is instantiated, its public static field `DFSgeneration.ROOT_VAR_MSG_TYPE` that is used for the type of the messages containing the elected root should be reset to the value of the public static field `SecureRerooting.OUTPUT`, which is the type used by `SecureRerooting` for its output messages. The attribute `ownerClass` is optional; if it is not specified, then the new message type is simply the value of the attribute `value`.

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`, etc.

Optionally, to improve the performance of your algorithm in terms of message sizes, you can implement your own message classes by subclassing `Message`. 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. Because `Message` implements `Externalizable`, you must not forget to do the following two things when you subclass `Message`:

1. Provide a public default constructor;

2. Properly override `writeExternal()` and `readExternal()`.

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`. This does not kill the agent; it only sends a notification of termination to FRODO. If another message is later received from a neighboring agent, the method `notifyIn()` will be called on this message, as before. If the algorithm does not have a built-in termination detection mechanism, but should terminate when all agents are idle (i.e. all agents are waiting for messages, but there are no more messages to be delivered), then the algorithm should terminate when it receives a messages of type `AgentInterface.ALL_AGENTS_IDLE`.[2]

5.2.3 The Module Constructor

All modules declared in the agent configuration file must have a constructor with the signature in Figure 15.

```
public MyModule (DCOPProblemInterface, org.jdom2.Element) { ... }
```

Figure 15: 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 14, and the constructor can be implemented as in Figure 16.

[2]Idleness detection is currently only supported when simulated time is enabled.

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

Figure 16: The constructor for the `SynchBB` module.

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 14 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.jdom2.Element, DCOPProblemInterface) { ... }
```

Notice that the order of the inputs is reversed compared to the constructor of classes implementing `IncomingMsgPolicyInterface`, given in Figure 15. 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 15. The second input describes the **overall** DCOP problem (while in Figure 15 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.

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 (Section 4.5) or from a Python script (Section 4.6). The abstract class `AbstractDCOPsolver` can be extended to produce such a solver; it declares the following two abstract methods:

```
public abstract ArrayList<StatsReporter> getSolGatherers ();
public 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:

```
public 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 3 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 (if your algorithm is complete and guaranteed to find the optimal solution).

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 `frodo2.algorithms.test` provides various methods to create random DCOP instances to be used as inputs for the tests.

A Catalogue of Constraints

The catalogue of constraints supported by FRODO depends on the XCSP parser used, as defined in the agent configuration file (Section 4.2.2). The simplest parser, `XCSPparser`, only supports soft, extensional constraints based on `relations`. The special parser `XCSPparserVRP` additionally supports *vehicle routing* global constraints. Finally, the most advanced parser based on JaCoP [9], `JaCoPxcspParser`, supports extensional (soft or hard) constraints (called `relations`), intensional, hard constraints (`predicates`), intensional, soft constraints (`functions`), and some global constraints: *all different*, *cumulative*, *diff2*, *element* and *weighted sum*.

This appendix describes in some level of detail the XCSP format for the constraints currently supported, as well as how to create such constraints without using XCSP, when applicable. Note that we also provide XML Schema files to help users write and validate XCSP problem files (see the files `XCSPschema*.xsd` in the `frodo2.algorithms` package). To check an XCSP problem file agains the XML schema, its `<instance>` element should include two attributes as follows:

```
<instance xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
          xsi:noNamespaceSchemaLocation = "path/XCSPschema.xsd" >
```

where the relative path to the appropriate XSD file should be adjusted accordingly. If you use JDOM to generate XCSP problem files, you should use the following command to set these attributes:

```
Element elmt = new Element ("instance");
elmt.setAttribute("noNamespaceSchemaLocation","path/XCSPschema.xsd",
   Namespace.getNamespace("xsi",
      "http://www.w3.org/2001/XMLSchema-instance"));
```

Note that the XML Schema standard version 1.0 does not support XCSP subsets used by `XCSPparserVRP` and `JaCoPxcspParser` (in which the type of the constraint depends on the value of the attribute `reference`); therefore we provide XML Schema 1.1 files instead (`XCSPschemaVRP.xsd` and `XCSPschemaJaCoP.xsd`). Using these XML Schema files to verify XCSP files requires an XSD parser that supports XML Schema 1.1; we suggest the use of the Xerces2 Java Parser [36], version 2.11.0-xml-schema-1.1-beta or later. To check an XCSP file `problem.xml` against the schema file `schema.xsd`, add `xercesSamples.jar`, `xercesImpl.jar` and `org.eclipse.wst.xml.xpath2.processor_1.1.0.jar` to your classpath, and run the following command:

```
java jaxp.SourceValidator -xsd11 -a schema.xsd -i problem.xml
```

A.1 Extensional Soft Constraint

XCSP Format FRODO's format for extensional soft constraints is based on the official XCSP 2.1 format for weighted tuples [23], 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`.

Java Class The class used to implement extensional soft constraints is the generic class `Hypercube<V, U>`, where `V` is the type of variable values, and `U` is the type of utility values (which, for most applications, can both be set to `AddableInteger`). A hypercube can be instantiated using one of its constructors, such as the following:

```
public Hypercube (String[] variables, V[][] domains,
                  U[] utilities, U infeasibleUtil) { ... }
```

where `infeasibleUtil` must be set to `ProblemInterface.getPlusInfUtility()` (resp. `getMinInfUtility`) if the problem is a minimization (resp. maximization) problem, and `utilities` must be an array of size equal to the product of all variable domain sizes. The utility for each assignment to the variables can then be specified using the method `setUtility(V[] assignment, U utility)`.

Important note: if you want FRODO to count constraint checks (NCCCs), you should use the following constructor instead:

```
public Hypercube (String[] variables, V[][] domains,
                  U[] utilities, U infeasibleUtil
                  DCOPProblemInterface<V, U> problem) { ... }
```

A.2 Extensional Hard Constraints

Extensional hard constraints are only supported by `JaCoPxcspParser`. They are specified using `relations` just like extensional soft constraints, except that they no longer mention costs/utilities, and the `semantics` are now either `"supports"` (all specified tuples are allowed, all others are disallowed) or `"conflicts"` (all specified tuples are disallowed, all others are allowed). For instance:

```
<relation name="relationName" arity="3" nbTuples="4"
          semantics="supports">
  0 0 0 | 0 0 1 | 0 1 0 | 0 1 1
</relation>
```

A.3 Vehicle Routing Constraint

When the special parser `XCSPparserVRP` is used, FRODO also supports intensional, *vehicle routing* constraints, as described in [17]. A vehicle routing constraint is specified as in Figure 17, for an example involving 3 customers and 4 vehicles. 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 Java class used to represent such constraints is VehicleRoutingSpace, in the package solutionSpaces.vehiclerouting.

Some of the customers may have uncertain locations, and the problem is then a StochDCOP [16]. In this case, the xCoordinate and yCoordinate attributes actually define the center of an "uncertainty circle," whose radius is defined by the value of the additional attribute uncertaintyRadius. A second additional attribute uncertaintyAngleVar gives the name of the (integer-valued) random variable corresponding to the uncertain position of the customer on this circle. For instance, in Figure 17, the last customer's position is uncertain.

```
<constraint name="uniqueConstraintName" arity="4"
      reference="global:vehicle_routing" scope="cust1 cust2 cust3 r3">
    <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"
                      uncertaintyRadius="0.5" uncertaintyAngleVar="r3" />
        </customers>
    </parameters>
</constraint>
```

Figure 17: A vehicle routing constraint.

A.4 Intensional Hard Constraints

Intensional hard constraints (only supported by the JaCoPxcspParser) can be expressed using constraint elements, whose reference is the unique name of a predicate [23], which is defined in Figure 18. When referring to a predicate, a constraint must specify the values (constants or variables names) that should be assigned to the parameters of the predicate, as in Figure 19.

A.5 Intensional Soft Constraints

Intensional soft constraints (only supported by the JaCoPxcspParser) can be expressed using constraint elements, whose reference is the unique name of a

```
<predicate name="uniquePredicateName">
  <parameters> int p1 int p2 ... int pn </parameters>
  <expression>
    <functional>
      boolean expression over (p1, p2, ..., pn)
    </functional>
  </expression>
</predicate>
```

where a *boolean expression* is formally defined as follows:

```
<booleanExpression> ::= "not(" <booleanExpression> ")"
  | "and(" <booleanExpression> "," <booleanExpression> ")"
  | "or(" <booleanExpression> "," <booleanExpression> ")"
  | "xor(" <booleanExpression> "," <booleanExpression> ")"
  | "iff(" <booleanExpression> "," <booleanExpression> ")"
  | "eq(" <integerExpression> "," <integerExpression> ")"
  | "ne(" <integerExpression> "," <integerExpression> ")"
  | "ge(" <integerExpression> "," <integerExpression> ")"
  | "gt(" <integerExpression> "," <integerExpression> ")"
  | "le(" <integerExpression> "," <integerExpression> ")"
  | "lt(" <integerExpression> "," <integerExpression> ")"

<integerExpression> ::= <integer> | <identifier>
  | "neg(" <integerExpression> ")" | "abs(" <integerExpression> ")"
  | "add(" <integerExpression> "," <integerExpression> ")"
  | "sub(" <integerExpression> "," <integerExpression> ")"
  | "mul(" <integerExpression> "," <integerExpression> ")"
  | "div(" <integerExpression> "," <integerExpression> ")"
  | "mod(" <integerExpression> "," <integerExpression> ")"
  | "pow(" <integerExpression> "," <integerExpression> ")"
  | "min(" <integerExpression> "," <integerExpression> ")"
  | "max(" <integerExpression> "," <integerExpression> ")"
  | "if(" <booleanExpression> "," <integerExpression> ","
    <integerExpression> ")"
```

Figure 18: Syntax for a `predicate`.

```
<constraint name="uniqueConstraintName" arity="n"
  scope="X1 X2 ... Xn" reference="predicateName">
  <parameters> X1 X2 ... Xn </parameters>
</constraint>
```

Figure 19: A predicate-based constraint.

`function` [23], which is defined below. The syntax for an *integer expression* is the same as in Figure 18. The integer value returned by the integer expression corresponds to the cost to be minimized (or the utility to be maximized).

```
<function name="uniqueFunctionName" return="int">
  <parameters> int p1 int p2 ... int pn </parameters>
  <expression>
    <functional>
      integer expression over (p1, p2, ..., pn)
    </functional>
  </expression>
</function>
```

A.6 Global Constraints

From the list of JaCoP global constraints, the `JaCoPxcspParser` currently only supports the *all different*, *cumulative*, *diff2* and *weighted sum* constraints.

A.6.1 *All Different* Constraint

The XCSP syntax for the global *all different* constraint [23] is illustrated below on a ternary constraint.

```
<constraint name="C" arity="3" scope="X0 X1 X2"
  reference="global:allDifferent">
  <parameters> [ X0 X1 X2 ] </parameters>
</constraint>
```

where the list of parameters may also contain integer constants.

A.6.2 *Cumulative* Constraint

The format for the *Cumulative* global constraint is a slight variation over the XCSP format in [23] (the end variables are omitted, and the operator has been

introduced). Below is an example of two tasks to be scheduled on a resource of capacity `limit`, where each task i starts at time step `start_i`, has a duration of `duration_i`, and requires `height_i` units of resource. All parameters `limit`, `start_i`, `duration_i` and `height_i` can be either variables or integers. The only two supported operators are `<eq/>` and `<le/>`.

```
<constraint name="C" arity="7" reference="global:cumulative"
  scope="start_0 duration_0 height_0
         start_1 duration_1 height_1 limit">
  <parameters>
    [
      { start_0 duration_0 height_0 }
      { start_1 duration_1 height_1 }
    ]
    <le/>
    limit
  <parameters/>
<constraint/>
```

A.6.3 *Diff2* Constraint

The XCSP syntax used in FRODO for the global constraint *diff2* is illustrated below, for three rectangles, where `orig_x_i` and `orig_y_i` are the variables for the x and y coordinates of the origin of the ith rectangle, and `size_x_i` and `size_y_i` are the variables for its sizes in the x and y dimensions.

```
<constraint name="C" arity="12" reference="global:diff2"
  scope="orig_x_1 orig_y_1 size_x_1 size_y_1
         orig_x_2 orig_y_2 size_x_2 size_y_2
         orig_x_3 orig_y_3 size_x_3 size_y_3">
  <parameters>
    [
     [ {orig_x_1 orig_y_1} {size_x_1 size_y_1} ]
     [ {orig_x_2 orig_y_2} {size_x_2 size_y_2} ]
     [ {orig_x_3 orig_y_3} {size_x_3 size_y_3} ]
    ]
  </parameters>
</constraint>
```

A.6.4 *Element* Constraint

The *element* global constraint enforces that a variable (or a constant) V be equal to the ith element (constant or variable) in a list, where i is the value of an index

variable I. FRODO slightly extends this definition by also allowing intervals in the list; V being "equal" to an interval then corresponds to V's value being contained in the interval. For example, the following constraint:

$$V \begin{cases} = 1 & if \quad I = 0 \\ = X & if \quad I = 1 \\ \in [0,3] & if \quad I = 2 \end{cases}$$

can be represented by the following XCSP fragment:

```
<constraint name="C" arity="3" reference="global:element"
  scope="I X V">
  <parameters>
   I [1 X 0..3] V
  </parameters>
</constraint>
```

A.6.5 *Weighted Sum* Constraint

The XCSP syntax for the global *weighted sum* constraint is as follows, for the example constraint $X_0 + 2X_1 - 3X_2 > 12$ [23]:

```
<constraint name="C" arity="3" scope="X0 X1 X2"
  reference="global:weightedSum">
  <parameters>[ { 1 X0 } { 2 X1 } { -3 X2 } ] <gt/> 12</parameters>
</constraint>
```

The format supports the following comparison operators: `<eq/>`, `<ne/>`, `<ge/>`, `<gt/>`, `<le/>`, and `<lt/>`.

B Catalogue of Benchmarks

Besides being compatible with the benchmark problem generators in DisCHOCO 2 [34], FRODO also comes with its own rich suite of benchmark problem generators that can be used to evaluate the performances of various algorithms.

B.1 Graph Coloring

In a distributed graph coloring problem, each agent controls a single variable whose value corresponds to a color, which must be different from the respective colors of the agent's neighbors in an underlying graph ([12], Section 2.2.1).

FRODO's random graph coloring problem generator can be invoked using the following command (the optional input parameters are put in brackets):

```
java -cp frodo2.jar frodo2.benchmarks.graphcoloring.GraphColoring \\
     [-i] [-soft] [-mpc] nbrNodes density tightness nbrColors [stochNodeRatio]
```

`-i` outputs a problem in intensional form;

`-soft` outputs a Max-DisCSP instead of a DisCSP;

`-mpc` also outputs an alternative problem formulation in which all constraints are public, for use with the MPC-Dis(W)CSP4 algorithms [31, 32];

`nbrNodes` the number of nodes;

`density` the fraction of pairs of nodes that are neighbors of each other;

`tightness` if > 0, the output problem contains unary constraints of expected tightness `tightness`;

`nbrColors` the number of colors;

`stochNodeRatio` the fraction of nodes whose color is uncontrollable; the output is then a StochDCOP ([12], Section 4.2.1).

Using the API, it is also possible to generate graph coloring problems in which the underlying graphs have a particular structure. This can be done by calling the method `GraphColoring.generateProblem()` whose first input is a `Graph` object, which can be generated using the `RandGraphFactory`. This factory supports acyclic, chordal, ring, and grid graphs.

B.2 Meeting Scheduling

In a meeting scheduling problem, each agent must take part in one or more meetings, and must agree on the time for these meetings with the respective other attendees.

FRODO's random meeting scheduling problem generator can be invoked using the following command (the optional input parameters are put in brackets):

```
java -cp frodo2.jar frodo2.benchmarks.meetings.MeetingScheduling \\
     [-i] [-EAV] [-PEAV] [-EASV] [-infinity value] [-tightness value] \\
     [-maxCost value] nbrAgents nbrMeetings nbrAgentsPerMeeting nbrSlots
```

`-i` outputs a problem in intensional form;

`-EAV` use the *Events As Variables* approach [20];

`-PEAV` use the *Private Events As Variables* approach [20];

`-EASV` use the *Events As Shared Variables* approach, which is the same as the EAV approach except that the variables have no explicit owners;

`-infinity value` specifies the cost incurred by violating one constraint (set to `infinity` by default);

`-tightness value` for each agent and each time slot, the probability in $[0, 1]$ that the agent is not available at that time (default is 0.0);

`-maxCost value` each attendee assigns a random cost in $[0, \texttt{value}]$ to having any meeting at each time slot; the output is then a DCOP instead of a DisCSP;

`nbrAgents` the size of the pool of agents from which meeting participants are drawn randomly (i.e. upper bound on the actual number of agents involved in at least one meeting);

`nbrMeetings` the number of meetings;

`nbrAgentsPerMeeting` the number of agents per meeting;

`nbrSlots` the number of possible time slots for each meeting.

B.3 Random Max-DisCSP

FRODO can also be used to produce completely random, binary-constrained, single-variable-per-agent, Max-DisCSP instances, using the following command:

```
java -cp frodo2.jar \\
     frodo2.benchmarks.maxdiscsp.MaxDisCSPProblemGenerator \\
     nbrVars domainSize  p1 p2
```

`nbrVars` the number of variables;

`domainSize` the size of the variable domains;

`p1` the density of the graph (i.e. the fraction of pairs of variables that are neighbors of each other);

`p2` the tightness of the constraints (i.e. the fraction of the variable assignments that are infeasible).

Like for graph coloring (Section B.1), the method `generateProblem()` can also be called to produce Max-DisCSP instances based on graphs with a specific structure.

B.4 Auctions and Resource Allocation Problems

FRODO can take in random auction problem instances generated by the CATS generator [18] and formalize the winner determination problem as a DCOP (for auctions) or a DisCSP (for pure-satisfaction, resource allocation problems). This can be achieved using the following command (the optional input parameters are put in brackets):

```
java -cp frodo2.jar frodo2.benchmarks.auctions.main.CATSToXCSP \\
     -src file [-out dir] -method id [-min] [-discsp] [-i]
```

`-src file` the problem file output by CATS;

`-out dir` the directory where the XCSP file should be saved;

`-method id` the id of the DCOP formulation $\in [1, 5]$:

1. one binary variable per bid, owned by the corresponding bidder;
2. same as method 1, except that copy variables owned by the auctioneers are introduced to hide from the bidders the list of bidders for each good;

3. each bidder owns one binary variable per good involved in one of its bids, and auctioneers own copy variables ([12], Section 3.7.3);
4. each bidder owns one binary variable for each and every good (no copy variables);
5. each auctioneer owns one public binary variable for each and every bidder, and bidders express private constraints over these variables (to be used with MPC-Dis(W)CSP4 [31, 32]);
6. for each bidder and each good involved in one of the bidder's bids, there is one binary variable with no specific owner; bidders and auctioneers express private constraints over these common variables.

`-min` outputs a cost minimization problem instead of a utility maximization problem;

`-discsp` ignores bid prices and outputs a DisCSP in which each bidder should win exactly one of its bids;

`-i` outputs a problem in intensional form.

B.5 Distributed Kidney Exchange Problems

In a *Distributed Kidney Exchange Problem (DKEP)* ([12], Section 4.2.4), each agent represents a patient/donor pair, where the patient is awaiting a kidney transplant, and the donor is a friend or relative who is willing to donate one kidney but is incompatible with the patient. The problem consists in finding directed cycles such as "donor A gives to a compatible patient B, whose paired donor B gives to a compatible patient C, whose paired donor C gives to donor A's compatible paired patient in return." Such problem instances can be generated using the following command (the optional input parameters are put in brackets):

```
java -cp frodo2.jar frodo2.benchmarks.kidneys.KidneyExchange \\
     [-i] [-s] [-a] nbrPairs
```

`-i` outputs a problem in intensional form;

`-s` outputs a StochDCOP with one random variable modeling each patient's probability to die before the transplant;

`-a` applies arc consistency before outputing the (Stoch)DCOP;

`nbrPairs` the desired number of incompatible patient/donor pairs.

B.6 Equilibria in Party Games

The *party game* is a graphical, one-shot, strategic game in which each player must decide whether to attend a party, not knowing whether his liked or disliked acquaintances will also decide to attend. The problem of computing a Nash equilibrium to such a game can be formulated as a DisCSP ([12], Sections 2.2.6 and 3.7.4). FRODO can generate such random party games using the following command (the optional input parameters are put in brackets):

```
java -cp frodo2.jar frodo2.benchmarks.party.PartyGame \\
     [-i] epsilon mixed topology size [param]
```

`-i` outputs problems in intensional form;

`epsilon` the error for approximate equilibria (0.0 for exact equilibria);

`mixed` a Boolean indicating whether to compute mixed or pure Nash equilibria;

`topology` the type of the game graph ('acyclic', 'chordal', 'grid' or 'ring');

`size` the number of players, except for grid graphs, in which it is the length of the square grid side;

`param` for acyclic graphs, the branching factor; for chordal graphs, the rate of chords; else, unused.

B.7 Vehicle Routing Problems (DisMDVRP)

FRODO can also produce Distributed, Multiple-Depot Vehicle Routing Problems (DisMDVRPs) instances [15], based on the *Cordeau* repository of MDVRP instances in [1]. This can be achieved using the following command (the optional input parameters are put in brackets):

```
java -cp frodo2.jar frodo2.benchmarks.vehiclerouting.CordeauToXCSP \\
    [-e] [-Q maxLoad] [-s minSplit] [-u size] input_Cordeau_file [horizon]
```

`-e` outputs an extensional DCOP instead of using intentional VRP constraints;

`-Q maxLoad` overrides the maximum load for each vehicle specified in the input Cordeau file;

`-s minSplit` uses split deliveries, in which case each customer's order can be split among multiple depots, with a minimum split size of `minSplit`;

`-u size` all customers that can be served by more than one depot have uncertain positions [17] given by random variables of domain size `size`; the output is then a StochDCOP;

`input_Cordeau_file` the path to the input MDVRP file in Cordeau's format;

`horizon` a depot cannot serve a customer that is farther away than `horizon`.

Acknowledgements

We would like to thank the following people who have contributed code to the FRODO platform (in chronological order):

- Xavier Olive worked on adding MPI support;
- Nacereddine Ouaret and Stéphane Rabie contributed to the *solution spaces*;
- Jonas Helfer worked on S-DPOP and on *kidney exchange* benchmarks;
- Éric Zbinden implemented P-DPOP and P^2-DPOP;
- Achraf Tangui implemented a preliminary version of the meeting scheduling problem generator;
- Andreas Schaedeli worked on auction benchmarks and *sum* constraints;
- Arnaud Jutzeler worked on the coupling with JaCoP;
- Alexandra Olteanu implemented AFB and the Max-DisCSP problem generator;
- Sokratis Vavilis and Prof. George Vouros, from the AI-Lab of the University of the Aegean, contributed a preliminary implementation of Max-Sum.

References

[1] Bernabé Dorronsoro. The VRP Web. `http://www.bernabe.dorronsoro.es/vrp/`, March 2007.

[2] 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.

[3] Alessandro Farinelli, Alex Rogers, Adrian Petcu, and Nicholas R. Jennings. Decentralised coordination of low-power embedded devices using the max-sum algorithm. In Lin Padgham, David C. Parkes, Jörg P. Müller, and Simon Parsons, editors, *Proceedings of the Seventh International Conference on*

Autonomous Agents and Multiagent Systems (AAMAS'08), pages 639–646, Estoril, Portugal, May 12–16 2008.

[4] Amir Gershman, Amnon Meisels, and Roie Zivan. Asynchronous forward-bounding for distributed constraints optimization. In Gerhard Brewka, Silvia Coradeschi, Anna Perini, and Paolo Traverso, editors, *Proceedings of the Seventeenth European Conference on Artificial Intelligence (ECAI'06)*, pages 103–107, Riva del Garda, Italy, August 29–September 1 2006. IOS Press.

[5] 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.

[6] Graphviz – Graph Visualization Software. `http://www.graphviz.org/`, 2011.

[7] 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.

[8] John Hunter. The matplotlib python module. `http://matplotlib.org`.

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

[10] The JDOM XML toolbox for Java. `http://www.jdom.org/`, 2009.

[11] Jean-Yves Le Boudec. *Performance Evaluation of Computer and Communication Systems*. EPFL Press, Lausanne, Switzerland, 2010. `http://perfeval.epfl.ch`.

[12] Thomas Léauté. *Distributed Constraint Optimization: Privacy Guarantees and Stochastic Uncertainty*. PhD thesis, Ecole Polytechnique Fédérale de Lausanne (EPFL), Lausanne, Switzerland, November 11 2011.

[13] 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.

[14] 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.

[15] Thomas Léauté and Boi Faltings. Coordinating Logistics Operations with Privacy Guarantees. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence (IJCAI'11)*, pages 2482–2487, Barcelona, Spain, July 16–22 2011. AAAI Press.

[16] Thomas Léauté and Boi Faltings. Distributed Constraint Optimization under Stochastic Uncertainty. In *Proceedings of the Twenty-Fifth Conference on Artificial Intelligence (AAAI'11)*, pages 68–73, San Francisco, USA, August 7–11 2011.

[17] 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.

[18] Kevin Leyton-Brown, Mark Pearson, and Yoav Shoham. Towards a universal test suite for combinatorial auction algorithms. In Anant Jhingran, Jeff MacKie Mason, and Doug Tygar, editors, *Proceedings of the Second ACM Conference on Electronic commerce (EC'00)*, pages 66–76, Minneapolis, Minnesota, USA, October 17–20 2000. ACM Special Interest Group on Electronic Commerce (SIGEcom), ACM. `https://www.cs.ubc.ca/~kevinlb/CATS`.

[19] Rajiv T. Maheswaran, Jonathan P. Pearce, and Milind Tambe. Distributed algorithms for DCOP: A graphical-game-based approach. In David A. Bader and Ashfaq A. Khokhar, editors, *Proceedings of the ISCA Seventeenth International Conference on Parallel and Distributed Computing Systems (ISCA PDCS'04)*, pages 432–439, Francisco, California, USA, September 15–17 2004. ISCA.

[20] Rajiv T. Maheswaran, Milind Tambe, Emma Bowring, Jonathan P. Pearce, and Pradeep Varakantham. Taking DCOP to the real world: Efficient complete solutions for distributed multi-event scheduling. In Nicholas R. Jennings, Carles Sierra, Liz Sonenberg, and Milind Tambe, editors, *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'04)*, volume 1, pages 310–317, Columbia University, New York City, U.S.A., July 19–23 2004. ACM Special Interest Group on Artificial Intelligence (SIGART), IEEE Computer Society.

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

[22] Operations Research – Java Objects. JAR file: `https://sourceforge.net/projects/frodo2/files/3rd_party/`; Git repository: `https://github.com/chemicalweb/or-objects`.

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

[24] Brammert Ottens, Christos Dimitrakakis, and Boi Faltings. DUCT: An upper confidence bound approach to distributed constraint optimization problems. In *Proceedings of the Twenty-Sixth Conference on Artificial Intelligence (AAAI'12)*, volume 1, pages 528–534, Toronto, Ontario, Canada, July 22–26 2012.

[25] 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.

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

[27] 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.

[28] 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.

[29] 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.

[30] Adrian Petcu and Boi Faltings. MB-DPOP: A new memory-bounded algorithm for distributed optimization. In Manuela M. Veloso, editor, *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07)*, pages 1452–1457, Hyderabad, India, January 6–12 2007.

[31] Marius-Călin Silaghi. Hiding absence of solution for a distributed constraint satisfaction problem (poster). In *Proceedings of the Eighteenth International Florida Artificial Intelligence Research Society Conference (FLAIRS'05)*, pages 854–855, Clearwater Beach, FL, USA, May 15–17 2005. AAAI Press.

[32] Marius-Călin Silaghi and Debasis Mitra. Distributed constraint satisfaction and optimization with privacy enforcement. In *Proceedings of the 2004 IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT'04)*, pages 531–535, Beijing, China, September 20–24 2004. IEEE Computer Society Press.

[33] 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.

[34] Mohamed Wahbi, Redouane Ezzahir, Christian Bessiere, and El Houssine Bouyakhf. DisChoco 2: A platform for distributed constraint reasoning. In *Proceedings of the Thirteenth International Workshop on Distributed Constraint Reasoning (DCR'11)*, pages 112–121, Barcelona, Spain, July 17 2011. `http://dischoco.sourceforge.net`.

[35] Richard J. Wallace and Eugene C. Freuder. Conjunctive width heuristics for maximal constraint satisfaction. In *Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI'93)*, pages 762–768, Washington, DC, USA, July 11–15 1993. AAAI Press / The MIT Press.

[36] The Apache Xerces project. `https://xerces.apache.org`.

[37] 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.