FRODO 2.0: An Open-Source Framework for Distributed Constraint Optimization

Thomas Léauté, Brammert Ottens, and Radoslaw Szymanek

École Polytechnique Fédérale de Lausanne (EPFL)
Artificial Intelligence Laboratory (LIA)
firstname.lastname@epfl.ch

Abstract. Distributed Constraint Optimization (DCOP) is a field that has recently been getting more and more attention from academia and industry. However, very few open-source, off-the-shelf tools are currently available to solve DCOPs; examples are FRODO, DisChoco and DCOPolis. A DCOP platform should possess the following key qualities: the framework should be *reliable* and extensively tested, *deployable* in a truly distributed setting, and *modular* so that it is easy to customize and extend. This paper introduces the Java-based *FRODO 2.0* framework, which possesses all three qualities. It is a complete re-design of the FRODO framework, released under the GNU Affero GPL license.

1 Introduction

As the field of Distributed Constraint Optimization (DCOP) is gaining popularity and momentum in the academic world, and more industrial partners might be tempted to apply various DCOP algorithms to real-life problems, the scarcity of available DCOP frameworks is becoming dire. While a few frameworks already exist, such as FRODO [1], DisChoco [2] or DCOPolis [3], we have been looking for a platform that would possess all of the three following qualities, which we deem necessary: *reliability*, *deployability* and *modularity*.

The DCOP community is concerned with developing new algorithms, and comparing their performances against existing ones. For this purpose, an open-source DCOP framework must provide some guarantees of *reliability* of the algorithm implementations it offers as benchmarks, in terms of correctness and completeness. *Modularity* also makes it easier to modify an algorithm's behavior, and combine algorithms to produce hybrids, while facilitating code reuse. Another important quality required for a framework to be used in industry is the capability to readily *deploy* algorithms in a truly decentralized setting.

This paper announces the release of the *FRODO 2.0* framework for DCOP [4]. While it still bears the same *FRODO* name as a tribute to the early framework by Petcu [1], version 2.0 is a completely new Java framework, re-designed and re-implemented from scratch with the three previously mentioned qualities in mind. In particular, it comes with a very extensive suite of randomized unit tests to provide guarantees of correctness and completeness of the DCOP algorithms.

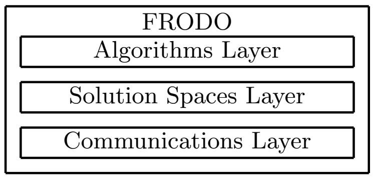

Fig. 1. General FRODO software architecture.

Its code is systematically documented using Doxygen to facilitate maintenance and encourage external contributions. Its expressive XML problem file format is shared with the open-source CP solver JaCoP [5], which we plan to integrate into the framework to support complex local subproblems. The FRODO framework is distributed under the GNU Affero GPL.

2 FRODO Architecture

This section describes the multi-layer, modular architecture chosen for FRODO (Figure 1). We describe each layer in more detail in the following subsections.

2.1 Communications Layer

The *communications layer* is responsible for passing messages between agents. At its core is the `Queue` class, which is an elaborate implementation of a message queue. Each `Queue` object runs its own thread, retrieving messages as they are delivered into its inbox. Messages are Java objects extending the general `Message` class, and have specific *types*. DCOP algorithms are implemented in the form of one or more *listeners* that register to `Queue`s for certain types of messages (Section 2.3). `Queue` objects are more than just a means to buffer received messages; they can also be used to send messages to other `Queue`s, through *pipes* that connect pairs of `Queue`s to form a network. Pipes can be of two types: *shared-memory pipes* are used to connect `Queue`s running in the same JVM, while *TCP pipes* handle the communication between `Queue`s in separate JVMs, possibly also on separate computers; a mix of both types of pipes can be used.

2.2 Solution Spaces Layer

This layer provides classes that can be used to model and reason about constraint optimization problems. A *solution space* can be seen as a constraint or a combination of constraints that describes a subspace of solutions to a problem. In the context of DPOP [6], solution spaces are not only used to represent each agent's local subproblem; they are also a way for agents to exchange information during the *UTIL propagation* phase. In the classical version of DPOP, the solution spaces are *hypercubes*. In contrast, the H-DPOP algorithm [7] uses *utility*

```
<instance>
  <presentation name="sampleProblem" maxConstraintArity="2" format="XCSP 2.1_FRODO" />
  <domains nbDomains="1">
    <domain name="three_colors" nbValues="3">1..3</domain>  </domains>
  <variables nbVariables="3">
    <variable name="X" domain="three_colors" owner="agentX" />
    <variable name="Y" domain="three_colors" owner="agentY" />
    <variable name="Z" domain="three_colors" owner="agentZ" />  </variables>
  <relations nbRelations="1">
    <relation name="NEQ" arity="2" nbTuples="3" semantics="soft" defaultCost="0">
      infinity: 1 1|2 2|3 3  </relation>  </relations>
 <constraints nbConstraints="2">
    <constraint name="X_not_equal_Y" arity="2" scope="X Y" reference="NEQ" />
    <constraint name="X_not_equal_Z" arity="2" scope="X Z" reference="NEQ" />  </constraints>
</instance>
```

Fig. 2. An example FRODO XCSP file.

diagrams, which are more efficient than hypercubes when the space of feasible solutions is sparse, because they only represent the feasible solutions.

FRODO provides efficient implementations of solution spaces, as well as useful operations on them. Examples of operations used by DPOP include the traditional *join* and *projection* operations. The $\mathbb{E}$[DPOP] algorithm [8] also uses more advanced operations such as *expectation* and *consensus*.

While it is possible to interact with FRODO directly by inputing objects of class `SolutionSpace`, the framework has also been designed to accept DCOP problem files in XCSP 2.1 format [9]. It is an expressive, XML-based format for CSPs and WCSPs that supports the n-ary table constraints used in FRODO, as well as other more complex constraints. We slightly extended it to represent DCOPs by specifying which agent owns which variable(s). Fig. 2 gives a possible representation of a graph coloring DCOP in this format. Our choice of the XCSP format was motivated by our plans to integrate FRODO with the open-source CP solver JaCoP [5], which also uses an extension of XCSP and provides support for many types of global constraints. Such an integration would enable FRODO to be used for DCOPs in which each agent must solve a complex local subproblem.

2.3 Algorithms Layer

Each algorithm is implemented as one or more *modules*, which listen for incoming messages in the agent's `Queue`, and exchange messages with other local and remote modules. Typically, a module is associated with a phase of the algorithm; Fig. 3 illustrates DPOP's modules and their messages. The `VariableElection` and `DFSgeneration` modules are implemented following the algorithms in [10].

The advantages of such a modular design are manyfold. For instance, as several algorithms operate on *DFS pseudo-trees*, the `VariableElection` and `DFSgeneration` modules can be reused *as is* across algorithms. The main advantage of modularity in the context of DPOP is that it makes it possible to easily implement numerous hybrids of the existing versions of DPOP. Combin-

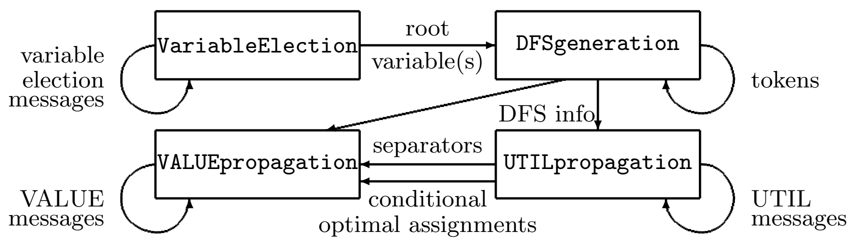

Fig. 3. Modular implementation of DPOP.

ing various modules to produce algorithms is performed by simply declaring the modules and their parameter values in an XML agent definition file.

The algorithms currently implemented in FRODO are DPOP [6], *Param-DPOP*, and ADOPT [11]. 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.

3 Experimental Setups

FRODO has been designed not only to be easily extendable and modular, but also to facilitate experimentation. A command-line based *controller* tool allows the user to run batch experiments, defined in an XML document specifying what algorithm to run and what XCSP problems to solve. The controller then takes care of setting up the agents and distributing the subproblems to be solved. However, it is also possible to set up the different agents and their subproblems locally, after which they register to the controller and connect to their neighbors.

The controller can be run in two different modes, depending on whether one or more machines are available to perform the experiments. In the *simple mode*, all agents are created in the same JVM as the controller. The *advanced mode*, however, allows one to run experiments in a distributed fashion. It requires a *daemon* to run on each machine participating in the experiment. These daemons are registered to the controller, which distributes the agents among them. FRODO currently distributes randomly and evenly the agents across the daemons, but future versions will support assigning specific agents to specific daemons.

As for performance evaluation, FRODO measures the amount of information conveyed via messages, the number of messages sent by each module, and the number of Non-Concurrent Constraint Checks (NCCCs) [12]. All these measures are reported to the central controller, regardless of whether the experiments are run on a single, or on multiple machines. Aside from these traditional performance measures, agents can also report other statistics, such as the chosen DFS or the optimal solution found, which is a useful metric for local search algorithms.

4 Conclusion

This paper announces the release of the new FRODO 2.0 framework for DCOP, which possesses several key qualities: *reliability* through extensive unit testing, *deployability* in truly distributed settings, and *modularity* in the DCOP algorithm implementations. Ongoing work includes performance improvements to the solutions spaces layer, and the implementation of ASODPOP [13] and $\mathbb{E}$[DPOP] [8]. As our first external user and contributor, Xavier Olive is also working on providing symmetry-breaking preprocessing methods [14] and MPI-based pipes. In the near future, we plan to implement H-DPOP [7] and S-DPOP [15], to support the *simulated time* metric from [3] and ENCCCs [16], message delay [17] and message loss, and to couple FRODO with the JaCoP solver [5] to support more complex, local subproblems.

References

1. Petcu, A.: FRODO: A FRamework for Open/Distributed constraint Optimization. Technical Report 2006/001, EPFL, Lausanne (Switzerland) (2006)
2. Ezzahir, R., Bessiere, C., Belaissaoui, M., Bouyakhf, E.H.: DisChoco: A platform for distributed constraint programming. In: IJCAI-DCR'07. (2007) 16–27
3. Sultanik, E.A., Lass, R.N., Regli, W.C.: DCOPolis: A framework for simulating and deploying distributed constraint optimization algorithms. In: CP-DCR. (2007)
4. FRODO: An open-source framework for DCOP. `http://liawww.epfl.ch/frodo/`
5. JaCoP: Java constraint programming solver. `http://jacop.osolpro.com/`
6. Petcu, A., Faltings, B.: DPOP: A Scalable Method for Multiagent Constraint Optimization. In: Proceedings of IJCAI'05. (2005) 266–271
7. Kumar, A., Petcu, A., Faltings, B.: H-DPOP: Using hard constraints for search space pruning in DCOP. In: Proceedings of AAAI'08. (July 13—17 2008) 325–330
8. Léauté, T., Faltings, B.: E[DPOP]: Distributed constraint optimization under stochastic uncertainty using collaborative sampling. In: DCR'09. (2009)
9. Organising Committee of the Third International Competition of CSP Solvers: XML Representation of Constraint Networks – Format XCSP 2.1. (2008)
10. Faltings, B., Léauté, T., Petcu, A.: Privacy guarantees through distributed constraint satisfaction. In: Proceedings of IAT'08. (December 9–12 2008)
11. Modi, P.J., Shen, W.M., Tambe, M., Yokoo, M.: ADOPT: Asynchronous distributed constraint optimization with quality guarantees. AI **161** (2005) 149–180
12. Meisels, A., Kaplansky, E., Razgon, I., Zivan, R.: Comparing performance of distributed constraints processing algorithms. In: AAMAS-DCR'02. (2002) 86–93
13. Ottens, B., Faltings, B.: ASODPOP: Making open DPOP asynchronous. In: Proceedings of the Doctoral Program of CP'08. (2008)
14. Olive, X., Nakashima, H.: Breaking symmetries in distributed constraint programming problems. In: Proceedings of the IJCAI09 DCR Workshop. (2009)
15. Petcu, A., Faltings, B.: S-DPOP: Superstabilizing, fault-containing multiagent combinatorial optimization. In: Proceedings of AAAI'05. (2005) 449–454
16. Chechetka, A., Sycara, K.: No-commitment branch and bound search for distributed constraint optimization. In: AAMAS'06. (2006) 1427—1429
17. Zivan, R., Meisels, A.: Message delay and DisCSP search algorithms. Annals of Mathematics and Artificial Intelligence **46**(4) (April 2006) 415–439