

Protecting Privacy in Multi-agent Optimization

Éric Zbinden
eric.zbinden@epfl.ch

Supervisors:

Thomas Léauté (thomas.leaute@epfl.ch)
Prof. Boi Faltings (boi.faltings@epfl.ch)

EPFL Artificial Intelligence Laboratory (LIA)
<http://liawww.epfl.ch/>

January 8, 2010

Contents

1	Introduction	3
2	Project Overview	3
2.1	DPOP	3
2.2	P-DPOP	3
2.3	Project Management	4
2.3.1	Spiral 1: Secure Variable Election	5
2.3.2	Spiral 2: Agent, Topology and Decision Privacy	6
3	Secure Variable Election	6
3.1	Implemented Features	7
4	Variable Obfuscation	8
4.1	Implemented Features	8
5	Experimental Results	9
5.1	Effects of Secure DFS Heuristics on Induced Width	10
5.2	Effects of Secured UTIL and VALUE Propagation	13
6	Conclusion	14

1 Introduction

Many real-life optimization problems involve multiple entities, or agents (individuals, companies...), with their own private constraints and preferences, communicating with each other in order to find a solution that maximizes the overall public satisfaction. In Artificial Intelligence, the field of Distributed Constraint Optimization (DCOP) has been addressing such multi-agent optimization problems, through distributed message-passing algorithms such as the DPOP algorithm [5].

However, the research in DCOP has been neglecting the privacy of the information exchanged by the agents during the computing of the solution, which is critical to many real-life problems. While agents are willing to cooperate with each other to produce an optimal solution, they are most often reluctant to reveal their private constraints and preferences to other, which hinders this cooperation.

The goal of this project was to implement, test, and evaluate a secured version of DPOP, P-DPOP [3]. P-DPOP provides strong agent privacy and topology privacy by randomization, constraints privacy and limited decision privacy by obfuscation. The algorithm was implemented in Java, as part of the open-source FRODO platform for DCOP [4].

2 Project Overview

2.1 DPOP

The DPOP algorithm [5] leaks privacy information in its three phases. First, in the DFS construction phase, the identity of the root is revealed to every agent as well as some information about the topology of the constraint graph.

Second, in the UTIL propagation phase, all the costs passed in the messages are in clear text. Some information is semiprivate, for example an agent learns what values are feasible for its own variables under different circumstances. However, agents learn the identity of all their ancestors even with whom they are not connected.

Third, in the VALUE propagation phase, assignments circulate through the graph in clear text. All agents receive the final assignments, even for variables of other agents with whom they do not have constraints.

2.2 P-DPOP

The P-DPOP algorithm [3] is a secured version of DPOP that provides guarantees on what private information can or cannot be leaked to others agents. First in the DFS construction, P-DPOP uses random integers to obfuscate variable IDs to

Algorithm 1: P-DPOP: DPOP with privacy guarantees (taken from [3]).

P-DPOP($\mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{C}$)

Initialization:

- 1 For each binary constraint $c(x, y) \in \mathcal{C}$, agent $a(x)$ generates a vector of random obfuscating keys $\mathbf{O}(x)$ that it sends to agent $a(y)$, which does likewise
- 2 For each variable $x_i \in \mathcal{X}$, agent $a(x_i)$ generates a codename $C(x_i)$, and codenames $C(v_1), \dots, C(v_k)$ for x_i 's domain values, and sends them to all agents owning a variable linked to x_i by a constraint

Anonymous DFS construction:

- 3 Choose root of DFS tree using Algorithm 2
- 4 Construct DFS labeling

UTIL propagation:

- 5 Wait for *UTIL* messages from all children
- 6 Partially deobfuscate received *UTIL* messages using known keys and codenames
- 7 As in *DPOP*, join resulting messages with own unary constraints and binary constraints involving (pseudo-)parents' variables; project x_i out
- 8 Obfuscate result and send to parent

VALUE propagation:

- 9 Wait for *VALUE* message from parent; deobfuscate it
 - 10 Compute optimal value v_i^* for x_i
 - 11 Send *VALUE* messages to all children using the codenames $C(x_i)$ and $C(v_i^*)$
-

determine the root of DFS tree. In the UTIL and VALUE propagation, all variable names and domains are also obfuscated by random codenames. Furthermore, all utilities transmitted are hidden by adding large random numbers. See Algorithm 1.

2.3 Project Management

For this project, *spiral management* was used. It is an iterative approach of system development. The term *spiral* is used to describe the process that is illustrated in Figure 1. The mechanisms go back several times to earlier sequences, over and over again, circulating like a spiral.

At the end of every spiral, we obtain a complete product. The next spiral increases the functionalities of the previous spiral with additional work. Accordingly the report is also written for every spiral release, when all specificities and important points are still fresh in mind. Product and report quality are also improved

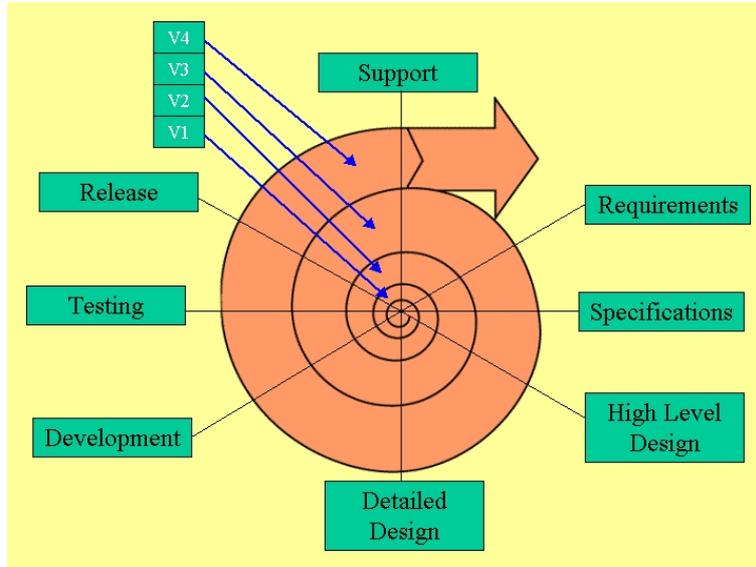


Figure 1: The spiral development process [2]

at every spiral. This is a plus compared to traditional *waterfall management*. Another improvement in spiral management is a better control of time. If the final deadline for the project is shortened for any reason, the product of the previous spiral can be turned in. But as a time inconvenient, spiral management can iterate much longer than waterfall management.

This project was managed in three spirals:

1. The DFS construction;
2. Agent, topology and decision privacy in both UTIL propagation and the VALUE propagation phase;
3. The third spiral should have focused on constrain privacy in UTIL propagation, but could not be completed in time.

For all the implemented code of this project, Junit tests are provided to ensure correctness. They give a reliability of the implemented code and an insurance that the algorithm functions as envisaged. It also helps debug and permits to find unexpected bugs.

2.3.1 Spiral 1: Secure Variable Election

This spiral focused on the implementation of the secured variable election module on top of the existent FRODO variable election module, and the evaluation of the

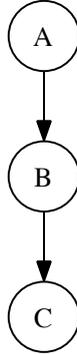


Figure 2: DFS tree

impact of this heuristic compared to the best known DPOP heuristic (the *most connected* heuristic).

2.3.2 Spiral 2: Agent, Topology and Decision Privacy

This spiral put the emphasis on the implementation of a new module for variable obfuscation and the evaluation of the impact of this module with and without “mergeback” (see Section 4.1) compared to DPOP.

3 Secure Variable Election

DPOP has two major privacy problems in its variable election module: first it leaks out the maximum variable ID, i.e. all agents learn which is the elected variable. For instance in Figure 2: A is elected root. The DPOP algorithm indicates to C that A is the root and even worse, DPOP reveals A’s ID, which, in the case of the *most connected* heuristic, contains A’s number of neighbors. But C should not learn about A’s existence.

Second, DPOP leaks out topology privacy. Agents know automatically the direction and the distance (in number of edges in the constraint graph) between them and the elected variable. Because the maximum ID is propagated step by step through the graph, this implies that variables that received the maximum ID at step 5 learn that they are at distance 5 of the root. The direction from where the maximum ID came also indicates automatically the direction where the root is located.

Algorithm 2: Anonymous leader election (taken from [3]).

```
1: elect_leader(a)
2: Generate unique obfuscated identifying number ID
3:  $max \leftarrow \mathbf{rand}(0 \dots ID)$ 
4:  $nb\_lies \leftarrow \mathbf{rand}(n \dots 2n)$ 
5: for  $nb\_lies$  times do
6:   Send  $max$  to all neighbors
7:   Get  $max_1 \dots max_k$  from all neighbors
8:    $max\_tmp \leftarrow \mathbf{max}(max, max_1, \dots, max_k)$ 
9:    $max \leftarrow \mathbf{rand}(max\_tmp \dots \mathbf{max}(ID, max\_tmp))$ 
10:  $max \leftarrow \mathbf{max}(max, ID)$ 
11: for  $(3n - nb\_lies)$  times do
12:   Send  $max$  to all neighbors
13:   Get  $max_1 \dots max_k$  from all neighbors
14:    $max \leftarrow \mathbf{max}(max, max_1, \dots, max_k)$ 
15: if  $max = ID$  then
16:   Choose any of the agent's variables  $x_r$  and mark it as the root of the DFS
   tree
```

To avoid these privacy problems, two features are implemented in P-DPOP. The leader election module follows Algorithm 2: every variable obfuscates itself using a large random ID number. In this manner, all variables will learn the maximum ID, but they cannot link this ID number back to the corresponding variable.

The second feature is, in the transmission phase, allowing variables to lie. Lying consists in transmitting a value for the maximum ID that is sub-evaluated. Agents will lie a random number of times, between N and $2N$, where N is an upper bound on the diameter of the constraint graph. Therefore, topology cannot be inferred because the maximum ID will propagate randomly and un-uniformly slower.

3.1 Implemented Features

The secure and insecure variable election modules are very similar. They elect the root with the same communication protocol based on viral propagation. The lack of privacy is not really in this protocol itself, which is why the secure module sub-classes the previous insecure module. They differ in the manner they generate and propagate IDs. The secure module uses generated random numbers and can sub-evaluate the maximum ID it transmits. Because of this sub-evaluation, the secured module uses three times more steps to elect the root.

Following FRODO's modular implementation, the secure variable election mod-

ule can also be used in the DPOP algorithm. This feature is used to demonstrate the isolated impact of secure DFS heuristics in Section 5.1.

Also, Algorithm 2 proposes to generate only positive random numbers. In our implementation, the span of random numbers generated by the secure module is between -2^{31} and $2^{31} - 1$. A larger range allows higher privacy and avoids ID collisions.

4 Variable Obfuscation

During DPOP’s UTIL and VALUE propagation phases, variable names and domains are transmitted in clear text. Variable Obfuscation solves this issue. First all variables generate randomly a codename and random numbers for their domains. All codenames must be unique to avoid errors during the computation. Codenames are generated as the hexadecimal representation of a random 32-bit number, in a way to propose a large span of possible codenames with the shortest string as possible for lowering the cost of transmission.

In a second phase, all variables send their codenames and obfuscated domains to their children and pseudo-children. When a variable then sends a UTIL or a VALUE message, the Variable Obfuscation module catches the message and encodes the variables and values contained in the message. At reception of a UTIL or VALUE message, the module also catches it and decodes all known codenames and values. This way, a variable has access only to information on the variables it has constrains with.

4.1 Implemented Features

This module can be used in two manners: with or without “mergeback”, which corresponds to whether or not variables are capable of decoding and merging back edges. All codenames contained in a UTIL message represent a constraint edge in the graph. When two constraints (two back edges, or one tree edge and one back edge) refer to the same variable, these edges can be merged into only one, reducing the size of the UTIL message.

With “mergeback”, all variables generate one unique codename that they transmit to every (pseudo-)child. This allows children that received a message that contains the codename for one of these (pseudo-)parents to decode it and to send only one codename for this (pseudo-)parent instead of multiple.

Without “mergeback”, all variables generate as many unique codenames as they have (pseudo-)children. Then they send to each a different codename. A child will not be able to recognize his (pseudo-)parent’s codename in a received

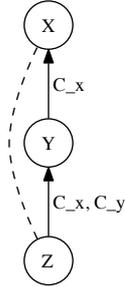


Figure 3: CodeNames sent with “mergeBack”

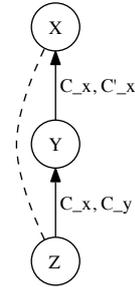


Figure 4: CodeNames sent without “mergeBack”

message. And when this child sends in turn a message, it will potentially contain multiple different codenames for the same variable.

Let us consider the graph in Figures 3 and 4, where codenames C_x , C'_x refer to X and C_y to Y . In the UTIL propagation phase, Z will send a message to Y that contains C_y and C_x . If “mergeback” is activated, Y will recognize C_x and therefore will send in its UTIL message to X only C_x . But if “mergeback” is not activated, Y will not be able to decode C_x because it received from X a different codename C'_x . Therefore Y will send to X a UTIL message that contains both C_x and C'_x .

In Algorithm 1, codenames and domain values of a variable are sent to all agents owning a neighboring variable. In our implementation however, only the agents who own a child or a pseudo-child of this variable receive this information, which reduces the number of messages exchanged.

5 Experimental Results

All experiments have been run on a Dell Precision M4300, Intel Core 2 Duo CPU 2.4GHz, on Windows XP, Eclipse 3.4.1 with 1024 Mbytes of JVM memory. For all experimental results, graphics represent the median with 95% confidence interval over 300 experiments.

The confidence intervals for the median are computed as follows: after the run of all 300 experiments, all results are sorted to find the median. Then confidence intervals are inferred as shown in Figure 5, with median in red and confidence interval in blue. For the formulae to be correct, the number of considered problems n must be at least 71. In experimental results, if a confidence interval is not present, this means that the interval is of length 0.

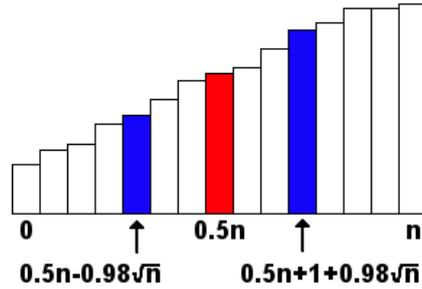


Figure 5: Represented median and confidence interval

5.1 Effects of Secure DFS Heuristics on Induced Width

The secured *random integer* heuristic does not allow the utilization of the best known heuristic, which is the *most connected* heuristic. Therefore, the DFS trees generated will have higher induced widths. This implies a bigger maximum number of variables in UTIL messages, which is an important factor of complexity.

The results from Figures 6, 8 and 9 were obtained for random graphs with 15 variables, 10 agents, and from 25 to 55 constraints. The time measurement of the leader election module alone is not constrained by the allocated memory, unlike for the full algorithm, so we were able to perform tests on graphs with more variation in connectivity, from low connectivity to complete graphs. The results in Figure 7 are obtained for random graphs with 25 variables, 15 agents, and from 10 to the maximum 300 constraints, using the *simulated time metric* [7].

Figure 6 shows the median induced width over 300 runs, with 95% confidence intervals (whose widths vary between 0 and 1). It also shows a linear increase in induced width. In general, the secure heuristic generates two more variables in its biggest UTIL message than the *most connected* heuristic. For problems with domain size 3, these two additional variables multiply by 9 (3^2) the memory requirements.

The simulated time metric has a granularity that depends on the underlying operating system and may be of 10 milliseconds [1]. This explains the zero results for low connectivity problems and the plateaus in Figure 7. The computational time of the secured heuristic grows clearly faster than the insecure leader election.

Figure 8 demonstrates that information exchanged by the secure leader election phase is about 2.5 times bigger than the insecure. This is a small difference compared to the exponential factor induced in the UTIL propagation, as shown in Figure 9.

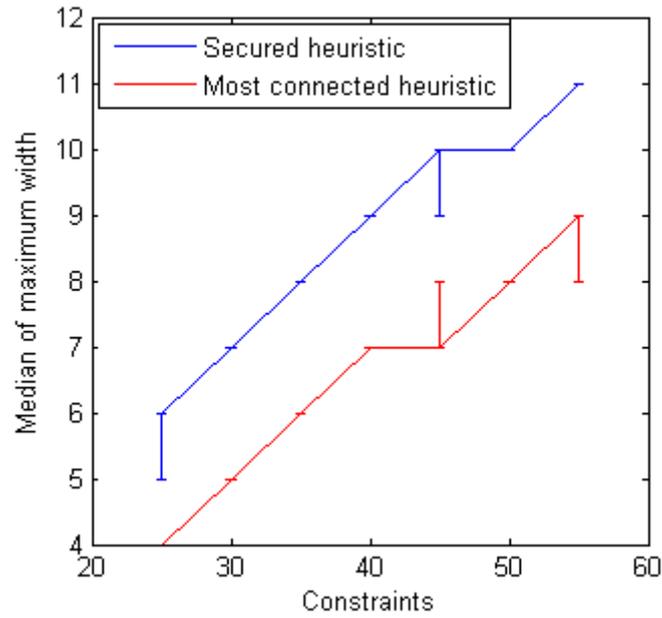


Figure 6: Effect of secured leader election on DFS width

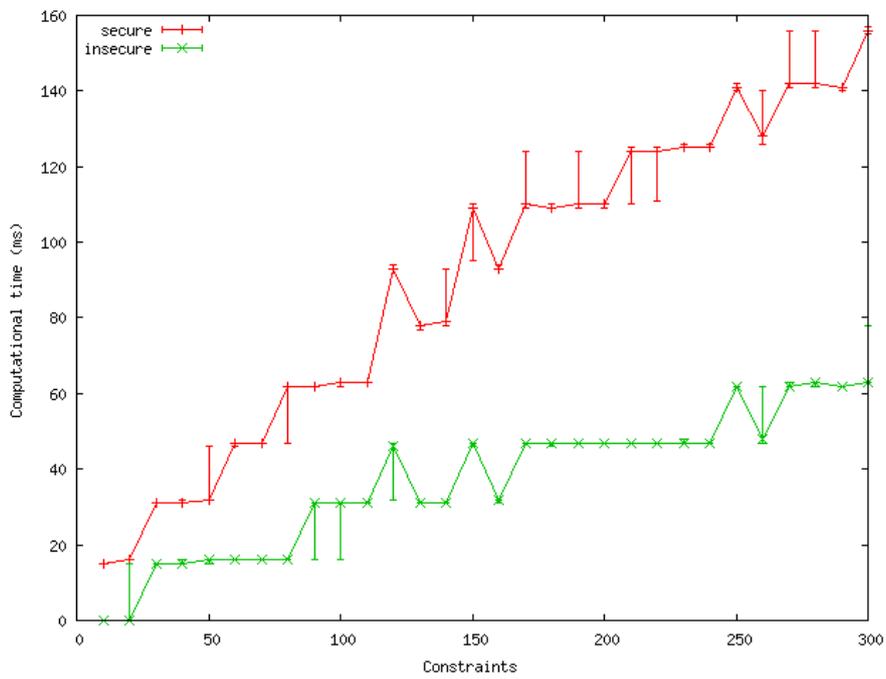


Figure 7: Computational time of leader election

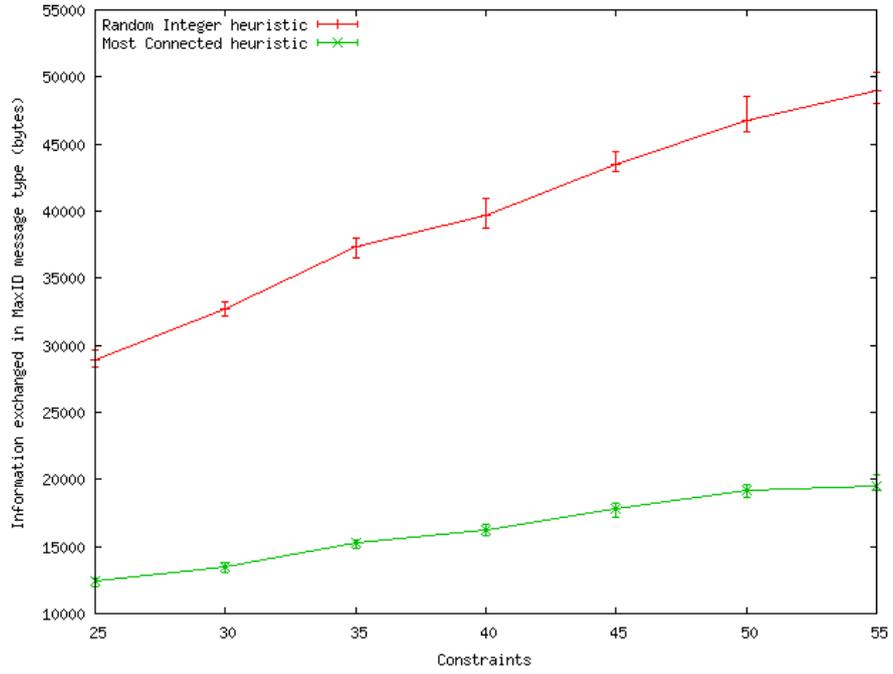


Figure 8: Information exchanged by Leader election phase

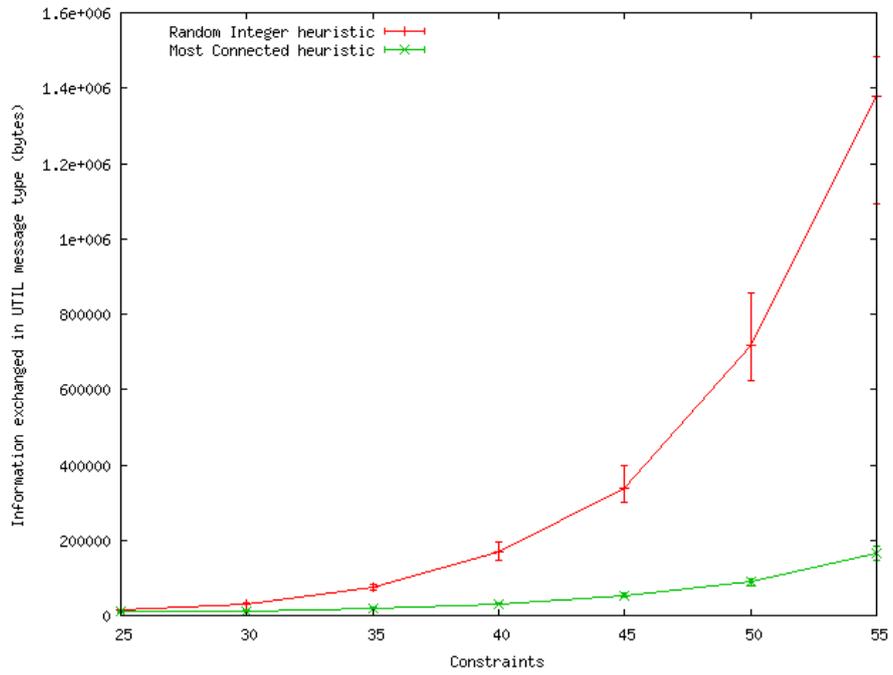


Figure 9: Information exchanged by UTIL propagation phase

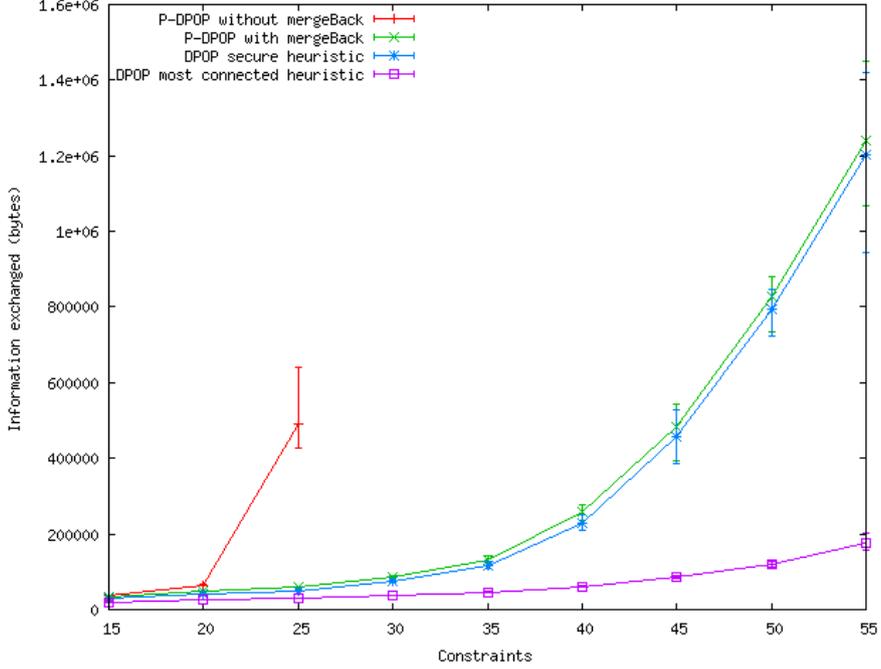


Figure 10: Total information exchanged by complete algorithms

5.2 Effects of Secured UTIL and VALUE Propagation

Effects appear in two factors: securing these phases produces more messages than DPOP’s insecure approach because it requires initially transmitting messages containing the codenames to be used. Without “mergeback”, the number of variables contained in a UTIL message will also increase by a factor that depends on the connectivity of the graph.

The following results were obtained for random graphs with 15 variables, 10 agents, and from 15 to 55 constraints. Results for P-DPOP without “mergeBack” are computed only up to 25 constraints because it exceeded the maximum memory allocated to compute a solution for higher numbers of constraints.

Figures 10 and 11 show that for low numbers of constraints, DPOP and P-DPOP with “mergeback” perform sensibly the same. But it shows the rapid exponential growth of the P-DPOP algorithm without “mergeback” compared to the other algorithms. We also see that the cost of lying in secure variable election and of exchanging codenames is very small compared to the cost of the overall algorithms. The real cost of P-DPOP is in a wider DFS tree, which is the bottleneck of the algorithms in the DPOP family. The computational time difference is linear during the leader election as seen in section 5.1, but the poorness of the DFS tree induces an exponential time difference in the overall algorithm.

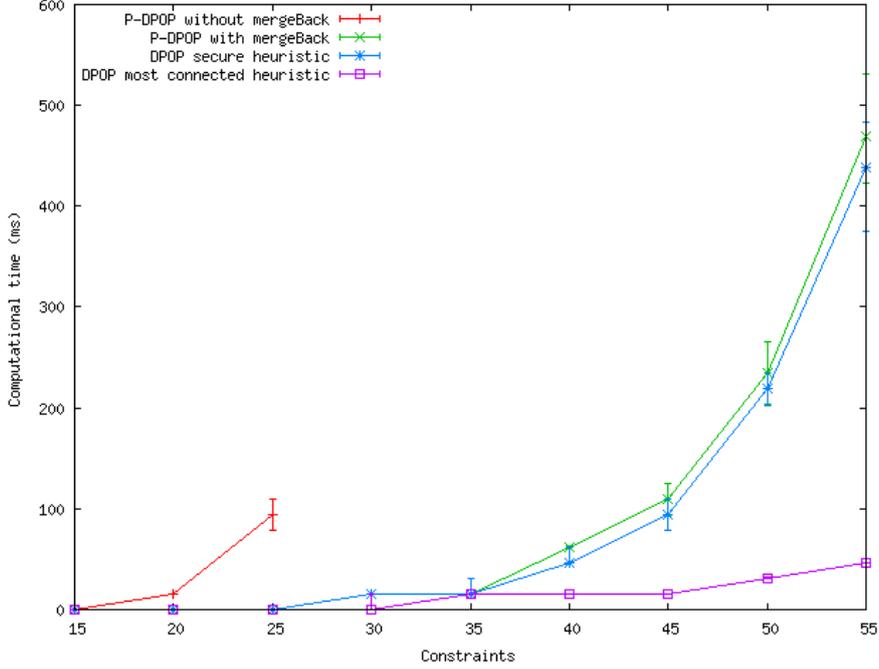


Figure 11: Computational time of complete algorithms

6 Conclusion

When a real-life distributed constraint satisfaction problem must be solved while preserving privacy, DPOP might not fit the privacy requirement. By using secure modules and non-deterministic features, the P-DPOP algorithm is able to grant this privacy.

Secure heuristics clearly generate poorer DFS trees than the insecure, *most connected* heuristic in terms of induced width. The difference might be thought minimal, but DPOP has a complexity that is exponential in this induced width; therefore secure heuristics incur a non negligible cost in memory and message sizes. However, the P-DPOP algorithm seems to be able to scale reasonably well compared to DPOP, on the random problems used in our experiments. On the other hand, additionally requiring that variables be incapable of merging back-edges (without “mergeback”) dramatically increases complexity, resulting in an algorithm that very quickly runs out of memory. Privacy has for sure a cost!

Future work includes experiments using a more realistic problem class, as well as implementing utility obfuscation (missing spiral three), which should have only a limited impact on performance.

References

- [1] Java 2 Platform SE 5.0, `currentTimeMillis`. [`http://java.sun.com/j2se/1.5.0/docs/api/java/lang/System.html#currentTimeMillis\(\)`](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/System.html#currentTimeMillis()).
- [2] Spiral scheme. [`http://www.proj-mgt.com/PMC_Spiral_Details.htm`](http://www.proj-mgt.com/PMC_Spiral_Details.htm).
- [3] 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.
- [4] Thomas Léauté, Brammert Ottens, and Radoslaw Szymanek. FRODO 2.0: An open-source framework for distributed constraint optimization. In Katsutoshi Hirayama, William Yeoh, and Roie Zivan, editors, *Proceedings of the IJCAI'09 Distributed Constraint Reasoning Workshop (DCR'09)*, pages 160–164, Pasadena, California, USA, July 13 2009.
- [5] 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.
- [6] Peter Sestoft. Java performance – Reducing time and space consumption, April 13 2005. [`http://www.dina.kvl.dk/~sestoft/papers/performance.pdf`](http://www.dina.kvl.dk/~sestoft/papers/performance.pdf).
- [7] 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.