Heuristics for Distributed Pseudo-tree Regeneration

Jonas Helfer
jonas.helfer@epfl.ch
Supervisors: Thomas Léauté, Boi Faltings
EPFL Artificial Intelligence Laboratory (LIA)
`http://liawww.epfl.ch/`

January 9, 2010

Contents

1 Project Description

The goal of this project is to develop a new heuristic for pseudo-tree regeneration in S-DPOP [8]. S-DPOP is a version of DPOP [6] which should show performance improvements when resolving problems after small changes. Examples of such problems are: tracking moving targets in sensor networks, truck-task scheduling and also matching patients and donors in kidney exchanges [9, 1] - a new problem for DCOP, which we investigate in this project .

In this project, the DCOP-platform FRODO [4], developed by the artificial intelligence lab (LIA) at EPFL, will be used. Because there is no existing implementation of S-DPOP on FRODO, a significant part of the project consists in implementing S-DPOP on FRODO for the first time. The most important part of the project is the development and validation of a new DFS heuristic, which should maximize the amount of reuse when a problem is resolved after a minor change.

Furthermore, we want to evaluate the performance of S-DPOP and the new heuristic on a real-life problem. For this, we use the kidney exchange problem, which is a problem class that has been studied in the centralized setting, but which is new to the DCOP community.

2 Introduction

2.1 What Is Distributed Constraint Optimization?

Distributed Constraint Optimization Problems (DCOP) are optimization problems in which variables are distributed over several agents. As in centralized constraint optimization problems, the goal is to optimize the overall utility (a function of the variables and their values). Because the variables and constraints are owned by different agents, a DCOP algorithm must be implemented in a distributed way, exchanging messages (information) between agents only where necessary. Aside from DPOP, there are currently several other algorithms which can solve DCOPs, such as Adopt [5] and NCBB [2]. Adopt and NCBB differ from DPOP in that they exchange an exponential number of messages of polynomial size relative to tree depth, whereas DPOP exchanges a linear number of messages of exponential size with respect to induced tree-width. DPOP is usually faster than Adopt, whenever enough memory is available to run the algorithm.

2.2 A Quick Look at DPOP

DPOP reduces the number of messages exchanged by arranging the variables (nodes) and constraints (edges) in a pseudo-tree (also called DFS) where messages are only exchanged between kids and parents. Usually, the largest messages sent between DPOP-variables are the messages exchanged during the UTIL-phase. Because of the arrangement in a pseudo-tree, each variable only ever sends one UTIL-message to its parent, resulting in a total number of N-1 messages sent up in the tree (N being the number of variables). The size of the messages however, increases exponentially with the induced width of the tree. The induced width is given by the size of the maximal set of back-edges which have overlapping tree-paths and distinct roots (parents in the pseudo-tree) [6].

Each additional back-edge with a distinct root adds a dimension to the hypercube (the information contained in a UTIL-message) traveling along the tree-edge in the pseudo tree.

Because of this, it is desirable to generate optimal pseudo-trees with respect to minimal induced width. It is also desirable to increase concurrency by making as many independent branches as possible. However, finding such an optimal pseudo-tree is an NP-hard problem. This is why existing distributed algorithms for pseudo-tree generation make use of heuristics, simple rules of thumb that are known to produce good-quality pseudo-trees.

2.3 Why Use S-DPOP?

In many cases, even solving relatively simple DCOPs can consume a lot of memory and time. There are certain groups of problems solvable by DCOP, in which small changes occur over time. Every time such a change occurs, DPOP must compute the solution from scratch. S-DPOP [8] can on the other hand reuse the information obtained from the previous solution and save time and memory by not recalculating everyting. To be able to reuse as much information as possible, the pseudo-tree obtained from the new problem should resemble the old one as much as possible, which is where the heuristic developed in this project comes in.

2.4 Project Goal

The goal of this project was to invent, implement, and evaluate a new heuristic for distributed pseudo-tree regeneration that can be used when a small change occurs in the optimization problem (e.g. a variable or a constraint appears or disappears). The heuristic should generate a new pseudo-tree that is as similar to the initial one as possible, so that computation carried out to solve the initial problem can be reused to solve the modified problem.

The pseudo-tree regeneration heuristic will be a part of the new implementation of S-DPOP on the open source platform FRODO [4].

3 Spiral Development

To guarantee constant progress, quality code and a timely delivery of the project, we used the spiral software development method.

Spiral development combines advantages of top-down and bottom-up concepts. It is a software development method (SDM) also known as the spiral lifecycle model. It integrates features of both the prototyping model and the waterfall model.

The following subsections describe spiral development as it was used in this project.

3.1 Iterations

1. Understanding the problem, setting the requirements for the module in question and defining how it should fit into existing code.

2. A preliminary design is created for the new module. This phase is the most important part of the "Spiral Model" as it allows to recognize problems in the prototype or conflicts with existing modules.

3. A first prototype is created according to the preliminary design, tested with appropriate JUnit tests and then committed to the CVS. A report is compiled to document the progress.

4. The subsequent prototypes are created as follows:

   (a) evaluating the previous prototype with experiments

   (b) defining the requirements of the next prototype

   (c) designing the next prototype

   (d) constructing and testing the next prototype

   (e) extending the report to include the new prototype and its results

3.2 Project Stages

Spiral 1 Self-stabilizing UTIL propagation: Implement UTIL-message storage and reuse and evaluate the performance under the assumption that only the utilities change, but not the pseudo-tree. This Spiral was good way to get familiar with the FRODO-platform and also served as a core element of S-DPOP. JUnit tests were written to assure that S-DPOP works correctly (meaning the solution is the same as with DPOP, and UTIL messages are reused when possible)

Spiral 2 Pseudo-tree regeneration: Invent and implement a heuristic, which reproduces the pseudo-tree in such a way, that as much information as possible can be reused from previous runs. In this spiral, both variable election and DFS-generation were modified and additional information about the DFS was stored. JUnit tests assure that the heuristic is indeed working correctly.

Spiral 3 Performance evaluation on a real problem: Understand the kidney exchange problem, implement the realistic kidney-pool-generator described in [9] for FRODO, formulate the corresponding DCOP and finally evaluate the message-reuse of S-DPOP on realistic changes to the problem.

4 Self-stabilizing UTIL Propagation

The goal of Spiral 1 was to implement the UTIL-reuse module of S-DPOP to reduce the number of messages sent between agents, assuming that the pseudo-tree has not changed. The main idea of this part was to get familiar with FRODO and DPOP and to lay the foundation for the implementation of S-DPOP on FRODO.

4.1 Implementation

The implementation of Spiral 1 included the following parts

- Creating an *empty message type* to minimize the size of messages passed between agents if possible. This message replaces the UTIL-message in case it has not changed from the previous run. Because DPOP is a synchronous algorithm, a message needs to be sent even when no change has occurred.
- Implementing the *UTIL reuse* module which intercepts messages and stores them in a tree map for subsequent warm restarts, shown in Figure 1. When the algorithm is run again, it replaces messages which are identical to the ones used in the previous run with the new *empty message type*. Upon reception of such an empty message, the module places the stored UTIL-message in the Queue of the agent.
- Implementing JUnit tests to validate the output of the algorithm by comparing it to the output of DPOP
- Running experiments on randomly modified problems to get an idea of the effect of the module on the sizes of messages passed between agents.

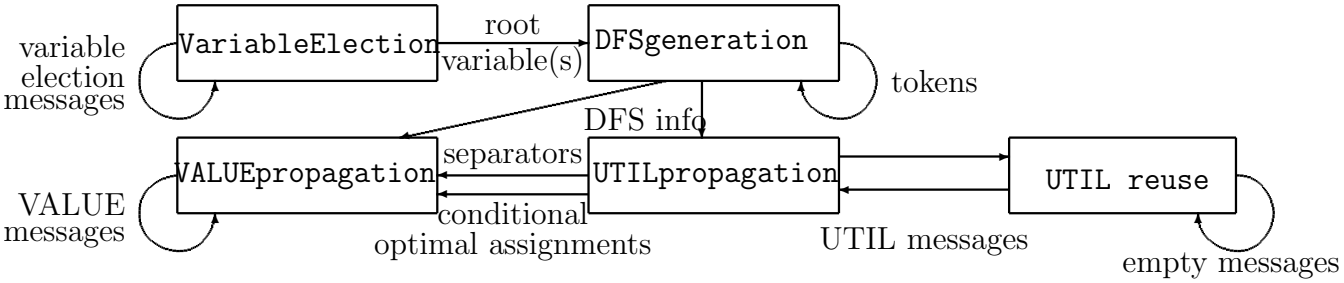

Figure 1: Schema of S-DPOP modules, including the new UTIL-reuse module

4.2 Experimental Results

Initial tests on a series of random problems with 12 agents, 12 variables and 50 constraints show that the total size of messages passed is greatly reduced when changes affect only few constraints. If more than 20% of the constraints are modified, the effect of *UTIL reuse* can be neglected.

Figure 2 shows the relative size of of all messages passed in a warm rerun with S-DPOP compared to DPOP.

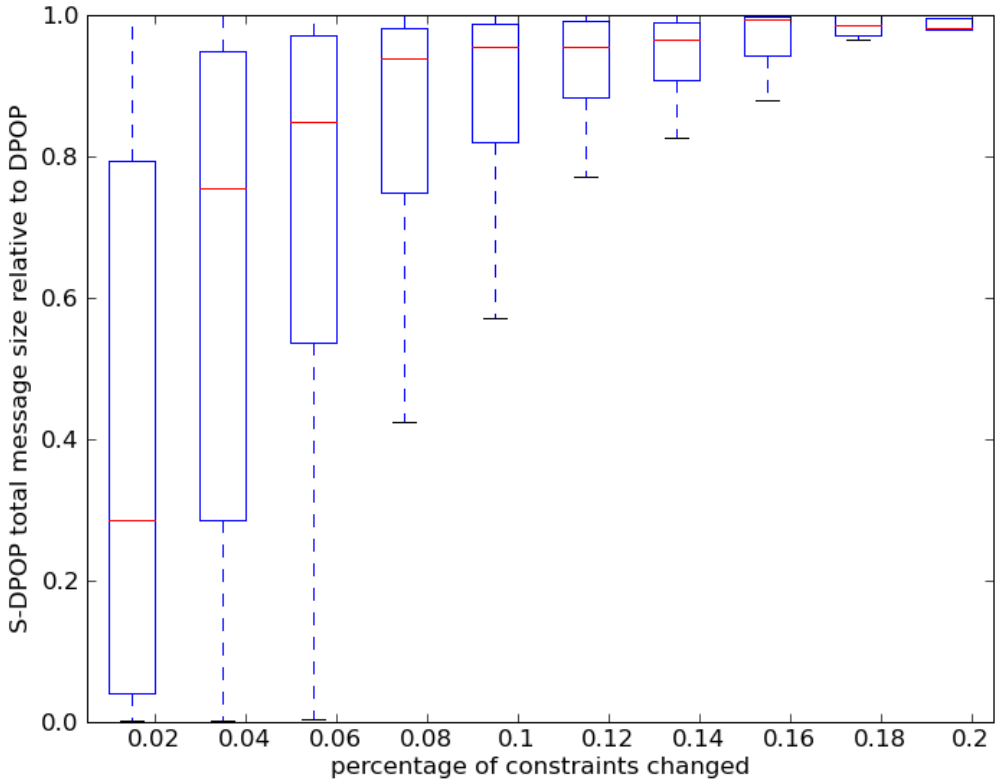

Figure 2: Total message size during rerun with S-DPOP compared to DPOP (tree not changed), 100 runs per column (500 for the 2%)

The scattering of the data suggests that the percentage of constraints changed is not the best indicator for the benefit of UTIL reuse. Whether stored messages can be reused depends very much on the location of the changed relations in the pseudo-tree. If the change happens to be far down in the DFS, changes will propagate upwards in the pseudo-tree and render the stored messages useless.

5 Pseudo-tree Regeneration

This part is the core part of the project, presenting a heuristic for pseudotree regeneration, which should rebuild the DFS in such a way that much of the information stored from previous runs can be reused, i.e. in such a way that the sizes of UTIL-messages sent are decreased as much as possible.

Changes in the tree can involve the appearance and disappearance of variables and constraints as well as changes in the utility function. The utility function does not affect the pseudo-tree, so any changes to it are not relevant for this part of the project. Addition or removal of variables can be interpreted as connecting or disconnecting previously separated variables with new constraints, so for the sake of simplicity, we will only look at changing constraints and not variables.

Due to the nature of information storage in UTIL messages (hypercubes), they can be reused if and only if the message sent from variable x to variable y is exactly the same in two consecutive runs of S-DPOP. Any change in a branch of the pseudo-tree will likely propagate all the way up the tree and thus render many stored messages obsolete. Changing the root of the tree can topple branches, in which case the messages also become obsolete. It thus seems that the best bet is to preserve the "hierarchy" in the tree and hope that the change will be limited to one branch.

5.1 The Min-depth and No-change Heuristic

One possibility to preserve as much of the structure as possible, is to make sure that the new root is the old root - unless it was removed, of course - and to make sure that variables are added to the DFS in the same order as in the last run. The only information required is the depth of each variable in the previous DFS (the root having depth = 0) and the previous neighbor relationships.

The min-depth heuristic, used for variable election, will simply pick the node with the lowest depth as the root. If two nodes have the same depth or if no previous information is available, a tiebreaker heuristic kicks in.

The no-change heuristic, used for DFS-generation, will pick the kids in the same order as before, based on the neighbourhood-map stored from the previous run. If a new variable appears, it will get picked last. If there is no information available, a tiebreaker heuristic will be used.

5.2 Example

Figure 3 shows a very simple scenario, in which just one variable is added. The min-depth and no-change heuristics make sure that variable a is again elected leader, even though the variables c or f could also be chosen by the most-connected

heuristic (which we used for this project as the heuristic in DPOP and as tie-breaker in S-DPOP). If c or f were chosen as new leaders, the pseudo-tree would be toppled and no longer similar to the initial one. In this small example, the addition of a variable in a low branch of the tree causes the messages further up in the branch to most likely change, which means the information stored in a previous run becomes obsolete. However, any information in other branches and unconnected sub-branches can be completely reused.

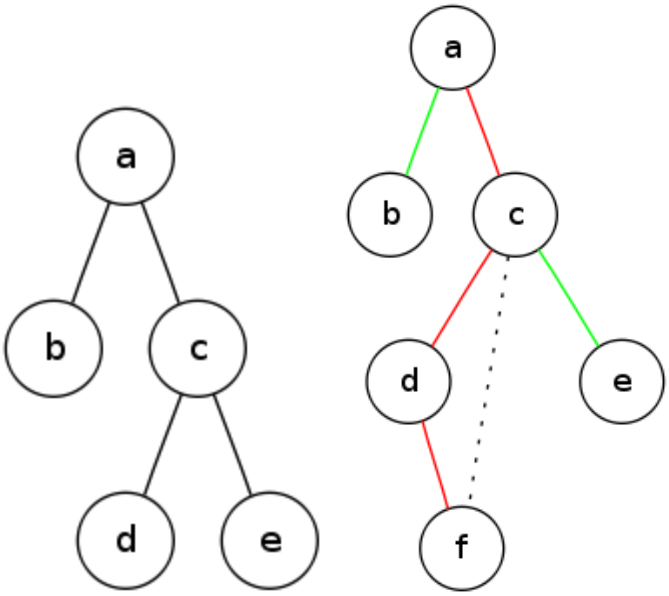

Figure 3: A simple scenario in which one variable is added. Non-reusable messages are indicated in red, reuseable ones in green

6 S-DPOP Performance Evaluation

As the tests in the section on UTIL-reuse have shown, tests on theoretical problems and controllable changes can have some interesting results. But they do not say anything about what happens when S-DPOP is used to solve a more realistic problem, because even small changes in the actual problem can result in major changes in terms of variables and constraints of the DCOP. The kidney exchange problem [9, 1] seems to be a good candidate for these tests, because it naturally changes over time, when new patients and donors appear or patients die.

There were other candidates for changing problems, such as target tracking in sensor networks or truck-task scheduling, but we opted for the kidney-exchange

problem because it has never been solved using distributed constraint optimization, so there was the additional challenge of translating it into a DCOP problem.

6.1 Kidney Exchange

In recent years, it has become possible to transplant a kidney from an unrelated donor with only a minimal risk of rejection by the recipient. Humans have two kidneys, but people can usually do fine with just one. This is why live donations (i.e. the donor being alive before and after the transplantation) are not unusual and more and more people in need of a kidney receive one in time. Nevertheless, some patients cannot find a suitable donor among their relatives or friends for one of two reasons: either their blood types (A, B and 0) are incompatible, or there is a risk of a positive crossmatch between the patient's antigens and the donor's blood. These patient-donor pairs are designated as incompatible, because there is a high risk of rejection if the patient recieves the organ from his donor. This is however not the end of all hopes, because some hospitals have started matching such incompatible patient-donor pairs ad-hoc to realize a pair-wise kidney exchange. In this case the donor of patient A can give to patient B while patient A recieves from donor B, if they happen to be compatible this way. Of course it is also possible to arrange the exchange in larger cycles, as long as every patient, whose donor donates to someone else, ends up receiving a kidney. Figure 4 shows an example compatibility graph with 4 patient-donor pairs. In this scenario, 3 can receive from both 2 and 1, but can give to no no other pair. There are three possible two-way exchanges (1,2),(1,4),(2,4) and one possible three-way exchange in two possible directions (1,2,4) or (2,1,4).

Because the transplants have to take place simultaneously, only two-way or three-way exchanges are practically feasible. For a small number of pairs, the maximum number of exchanges can be easily figured out, but with larger numbers, a more systematic approach is required. It is possible to formulate this problem as a DCOP and solve it with DPOP. This problem seems to be well suited for S-DPOP, because small changes occur naturally, when transplantations are performed, people die or new patient-donor pairs are added to the waiting list.

6.2 Problem Generator

To have a realistic problem, we needed to generate a realistic population of patient donor pairs. Saidman et al. describe a way to do this in their paper [9]. Their method is based on the American population and aims to mimic as closely as possible the real kidney-transplant waiting list. We managed to write a kidney-exchange problem generator for FRODO, which uses the method described by

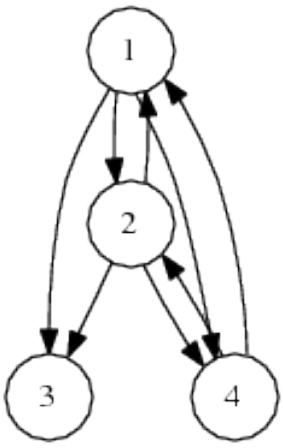

Figure 4: A compatibility graph and the corresponding DFS for a kidney exchange scenario of 6 pairs. The arrows indicate the direction in which the organs can be exchanged

Saidman et al. to generate a realistic population of patient-donor pairs and translated the situation into a DCOP. Figure 4 represents a sample pool of 4 pairs generated by this method.

6.3 DCOP Formulation for the Kidney Exchange Problem

Each variable in the DCOP corresponds to one patient-donor pair, its domain the potential donors for this patient and "0" for the case in which the patient does not recieve a transplant. There are three different kinds of constraints.

- A correctness constraint to make sure each organ is donated to at most one patient and one patient does not recieve more than one organ. This type of constraint links pairs of variables with overlapping domains and assigns utility "-infinity" to the case where the two variables involved have the same non-zero value.
- A two-way constraint between mutually compatible pairs, assigning utility 20 to any exchange where 'a' gives to 'b' and 'b' gives to 'a'
- A three-way constraint between 3 variables assigning utility 29 to the case where the three form a cycle

We have chosen the per-transplant utility of the 3-way exchange to be slightly smaller than the one of the 2-way, to favour the 2-way exchange over the 3-way

where possible, because 2-way exchanges are logistically easier to perform for hospitals (and thus cost less).

Figure 5 shows the DFS generated from the previous compatibility graph. Variable 4 is disconnected from the rest of the graph, because it cannot participate in any possible exchange. The rest of the variables are all connected and thus form a pseudo-tree with a single branch.

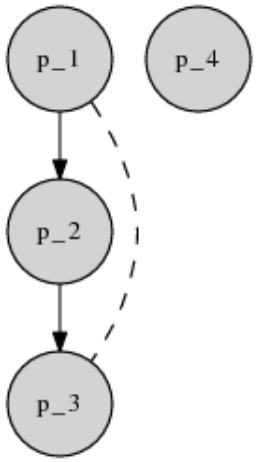

Figure 5: The corresponding DFS for the scenario in Figure 4.

6.4 Experimental Setup

We initially simulated two different scenarios to compare the sizes of the messages exchanged during the UTIL phase between DPOP, S-DPOP with the most-connected heuristic and S-DPOP with the new min-depth and no-change heuristic:

- Scenario 1: Adding pairs to the pool (new patient-donor pairs arrive)
- Scenario 2: Removing random pairs from the pool (patients or donors die)

6.5 Results

Because of the high connectivity of the problem and limited memory (1GB) available on the testing machine, we were only able to test these cases up to a size of at most 11 pairs. But since the memory use grows exponentially with the problem size, even a computer with a much larger memory would not have been able to tackle problems with more than 13 - 15 pairs.

The graphs in the following sections compare the size of UTIL messages exchanged after modifying the problem.

6.5.1 Scenario 1: Adding Random Pairs

We performed experiments on problems ranging from a size of 6 pairs up to 9 pairs and added between 0 and 4 pairs to see what happens. Figure 7 shows the results for pool sizes of 7 and 8 plotted over a logarithmic scale.

When the problem does not change, S-DPOP must obvioulsy outperform DPOP. We added this case to be able to compare the baseline performance.

As Figure 7 shows, S-DPOP offers no statistically significant improvement over DPOP for these problems. Apart from the case with zero change, the confidence intervals always overlap greatly. This outcome was not completely unexpected, as the problem is small and highly connected. For this size, even adding one pair usually constitutes a major change, resulting in very little reuse. Figure 6 shows, how the constraint graph changes after the addition of just one variable. When looking at the pseudo-trees generated (see Fig 5), we observe that generally the tree just consists of one major branch and sometimes several disconnected variables (corresponding to patients who are not compatible with any donor). This observation is in accordance with the results we have obtained in the experiments. It would have been nice to see that S-DPOP offers a significant advantage over DPOP for these kinds of problems. But after observing the DFS changes, this outcome would have been very surprising.

It can be considered as a positive result, that S-DPOP performs no worse than DPOP. This is not as self-evident as it may seem, because as changes accumulate over time, S-DPOP could start performing worse compared to DPOP, because the DFS-tree obtained by the heuristic will be suboptimal in many cases.

6.5.2 Scenario 2: Removing Random Pairs

In this scenario, we observe similar results to the previous one, as all three variants of DPOP have similar performance as soon as one or more variables are removed.

6.5.3 Summary of Results

Overall, the results show, that there is no significant difference in the performance of DPOP and S-DPOP, even for small changes to the problem. This is likely due to the fact that the kidney exchange problem is a highly connected problem, whose pseudo-trees usually have a high induced width, making them difficult to solve with DPOP. A look at the changes in the constraint graph caused by the addition of a single variable seems to intuitively support this argument. Any change to this connected problem is likely to affect the whole tree, making reuse very unlikely, if not impossible. The cases in which reuse occurs and S-DPOP is actually more efficient are counterbalanced by the cases in which no reuse is possible but larger

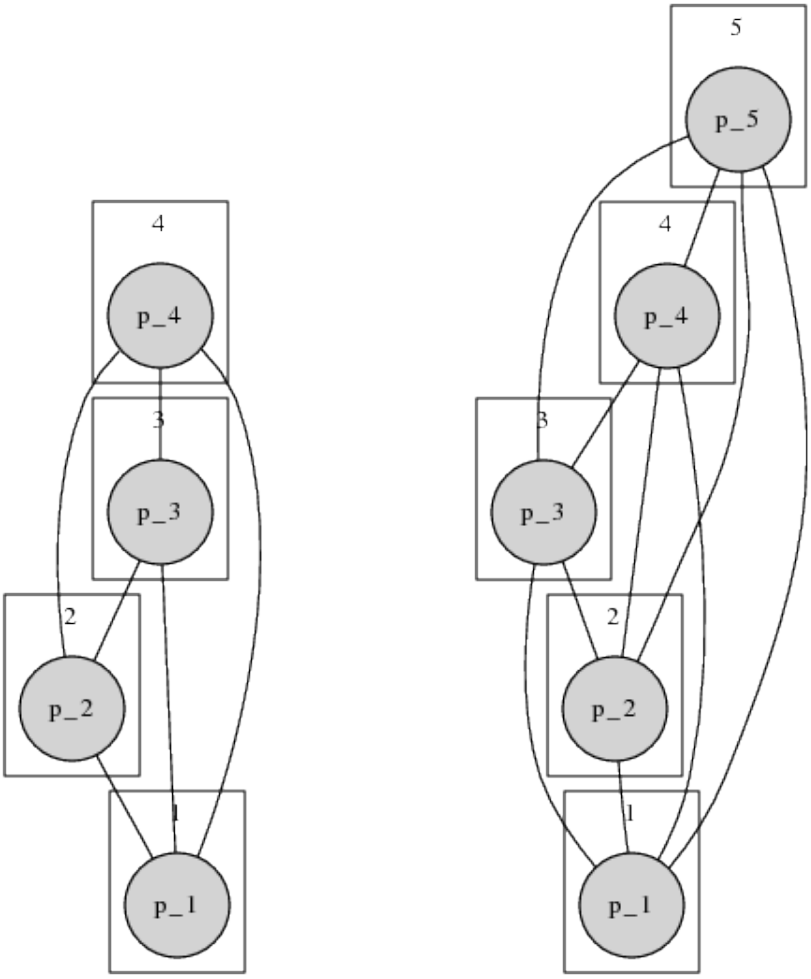

Figure 6: This figure shows the constraint graph before and after adding a variable. The boxes stand for the agents. In this case every variable is owned by a different agent.

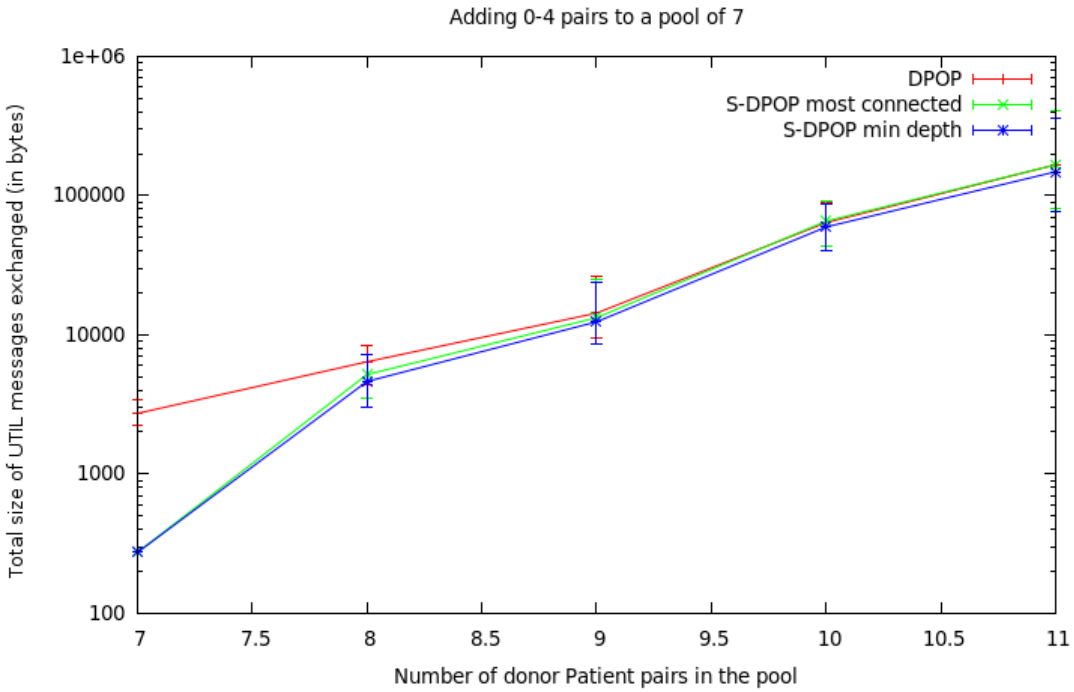

Figure 7: Median size of UTIL messages exchanged during a rerun after adding pairs, along with 95% confidence intervals for median for DPOP, S-DPOP and S-DPOP without the new heuristic (sample size > 100)

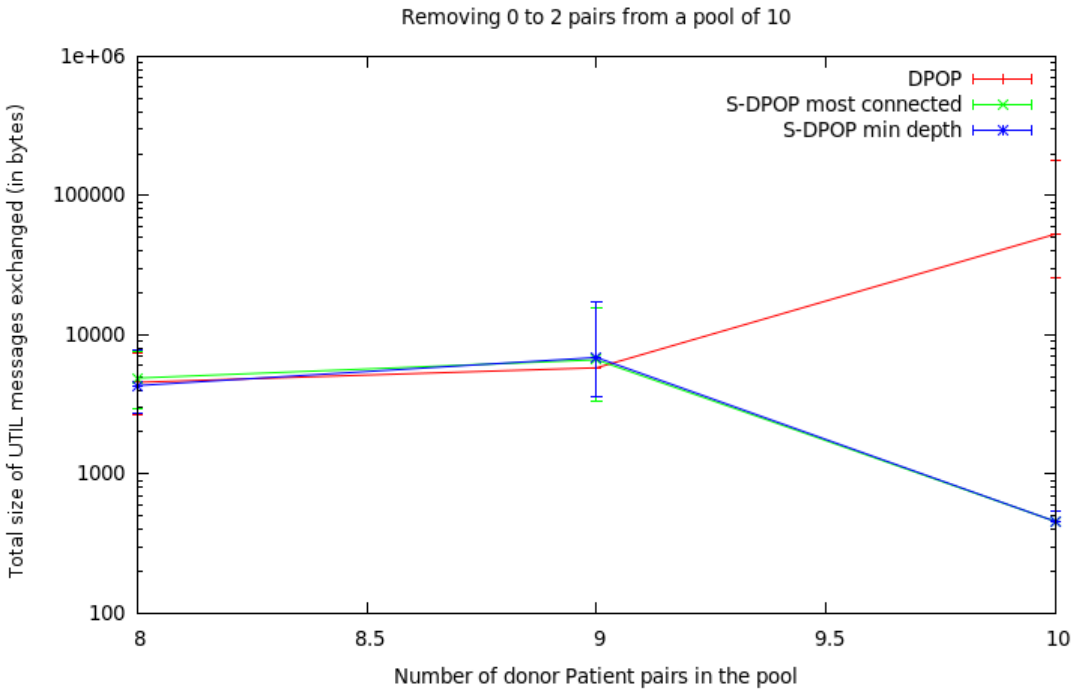

Figure 8: Median size of UTIL messages exchanged during a rerun after removing pairs, along with 95% confidence intervals for median for DPOP, S-DPOP and S-DPOP without the new heuristic (sample size > 100)

messages have to be exchanged because the new heuristic constructed a suboptimal pseudo-tree compared to the standard DFS heuristic.

It is important to point out that results only apply to small instances of the kidney exchange problem, and it is not unlikely that, for larger instances or for different problem classes, the performance gains of S-DPOP could be significant.

7 Conclusion

The main goal of this project was to develop a new heuristic for pseudo-tree regeneration to be used in S-DPOP, and comparing the performances of S-DPOP and DPOP. To do this, S-DPOP had to first be completely implemented and tested on the open source platform FRODO, before the heuristic could be created and validated. To compare the performance of S-DPOP and DPOP and test whether the new heuristic makes any difference, we formulated the kidney exchange problem as a DCOP and recreated a realistic donor-patient pair generator described in the paper of Saidman et al [9, 1] to create a new benchmark on the FRODO platform.

After implementing the kidney exchange benchmark on FRODO, we noticed that the kidney-exchange problem is highly connected (Figure 4 illustrates this well) and thus not easily solvable by DPOP. The pseudo-trees generated from the problem description tend to have a very high induced width, which grows almost linearly with the number of variables, thus leading to exponential growth in the size of the messages. Due to this fact, the performance of DPOP and S-DPOP could only be evaluated on problems of up to 12 pairs or less.

The results of the tests carried out on the kidney exchange problem showed that S-DPOP offers no significant gains in terms of message sizes exchanged compared to DPOP. On average, S-DPOP performed no worse than DPOP in any of the simulations. This is likely due to the fact that the problem sizes were too small and even minor changes affected the whole pseudo-tree because of the high connectivity. Also, because the way S-DPOP works is very similar to DPOP, it is impossible to draw statistically significant conclusions on this kind of problem.

7.1 Summary of Achievements

- Implemented S-DPOP on FRODO
- Invented and tested a heuristic for pseudo-tree regeneration
- Formulated the kidney-exchange problem as DCOP and implemented a kidney-exchange benchmark for DPOP and S-DPOP on FRODO that can also be used for other algorithms

- Showed that the new heuristic and implementation of S-DPOP do not offer any advantage over DPOP when tested on the kidney exchange problem

7.2 Future Work

Even though the tests did not indicate any performance gains for S-DPOP on the kidney exchange problem, there are several things worth investigating further with both S-DPOP and the min-depth and no-change heuristics.

Some possible steps for the future could include:

- Investigating why the kidney exchange problem is so hard to solve with DPOP and how it could be simplified. One promising option is:
  - Changing the DCOP model to assign *-infinity* utilities to assignments corresponding to n-ary exchanges with $n > 3$. Currently, they are not disallowed but always suboptimal, which can lead to an overhead in message sizes. This could then be used to reduce any variable's domain to remove values that are always infeasible. For instance: giving to a patient whose donor cannot give to anyone in return.
- Improving the performance of S-DPOP by swapping stored messages to disk. S-DPOP needs to remember all UTIL messages sent and received, in order to compare them with the ones sent in the following run. Currently, they are all kept in memory, which sometimes causes the Java VM to run out of memory. The same thing could be done for the optimal conditional assignments output by the projection operator.
- Using H-DPOP [3] instead of DPOP, i.e. utility diagrams instead of hypercubes, to reduce the sizes of UTIL messages. If the problem leads to sparse messages, this could be a significant gain in performance.
- Finding a constantly changing problem easier to solve with DPOP
- Implementing RS-DPOP [7] on FRODO to see how it performs in the case of the kidney exchange problem
- Looking into ways for improving the current heuristic. One possibility would be to choose the leader depending on the size of messages previously exchanged in the subtrees of the variables to achieve a more balanced tree where possible.

References

[1] Pranjal Awasthi and Tuomas Sandholm. Online stochastic optimization in the large: Application to kidney exchange. In Craig Boutilier, editor, *Proceedings of the Twenty-First International Joint Conference on Artificial Intelligence (IJCAI'09)*, pages 405–411, Pasadena, California, USA, July 11–17 2009.

[2] Anton Chechetka and Katia Sycara. No-commitment branch and bound search for distributed constraint optimization. In Hideyuki Nakashima, Michael P. Wellman, Gerhard Weiss, and Peter Stone, editors, *Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'06)*, pages 1427—1429, Hakodate, Japan, May 8–12 2006. ACM Press.

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

[4] Thomas Léauté, Brammert Ottens, and Radoslaw Szymanek. FRODO 2.0: An open-source framework for distributed constraint optimization. In *Proceedings of the IJCAI'09 Distributed Constraint Reasoning Workshop (DCR'09)*, pages 160–164, Pasadena, California, USA, July 13 2009. `http://liawww.epfl.ch/frodo/`.

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

[6] 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.

[7] Adrian Petcu and Boi Faltings. RS-DPOP: Optimal solution stability in continuous-time optimization. In *Proceedings of the 2005 Workshop on Distributed Constraint Reasoning (IJCAI '05 - DCR)*, Edinburgh, Scotland, July 31 – August 5 2005.

[8] 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.

[9] Susan L. Saidman, Alvin E. Roth, Tayfun Sönmez, M. Utku Ünver, and Francis L. Delmonico. Increasing the opportunity of live kidney donation by matching for two- and three-way exchanges. *Transplantation*, 81(5):773–782, March 15 2006.

[10] Peter Sestoft. Java performance – Reducing time and space consumption, April 13 2005.