Coordination and Sampling in Distributed Constraint Optimization

THÈSE N^{O} 5396 (2012)

PRÉSENTÉE LE 13 JUILLET 2012

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS
LABORATOIRE D'INTELLIGENCE ARTIFICIELLE
PROGRAMME DOCTORAL EN INFORMATIQUE, COMMUNICATIONS ET INFORMATION

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Brammert OTTENS

acceptée sur proposition du jury:

Prof. A. Wegmann, président du jury
Prof. B. Faltings, directeur de thèse
Prof. K. Aberer, rapporteur
Prof. K. Larson, rapporteur
Prof. M. Yokoo, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2012

"Reading furnishes the mind only with materials of knowledge;
it is thinking that makes what we read ours."
— John Locke

To Maria...

Acknowledgements

With the conclusion of this thesis, a long journey comes to an end. Many people have helped me in one way or another on this long but satisfying journey. Foremost there is Prof. Boi Faltings. Boi agreed to take me on as a PhD student, and through the years guided me with a soft hand through the small bumps and huge crevasses that constitute a PhD. Not all routes that we have taken have lead to satisfying results, but such is the nature of research. I am very grateful for all the times you walked into my office, suggesting me to read this or that paper and for the many the helpful comments on my writing.

I would like to thank my thesis committee members for taking the time to read my thesis, and for the comments that they gave that helped improve this thesis. I would also like to thank a couple of people in the lab, without whom my thesis would have looked very different. I owe a lot to Thomas Léauté, who was working on the same topic. I greatly enjoyed, and appreciated the many fruitful discussions we had over the years. Working on FRODO together was a lot of fun, and I feel that we can be proud of the result. Christos Dimitrakakis, although our time in the lab together has been short, I greatly enjoyed our many discussions on sampling and stochastic methods. Radek Szymanek, thanks for your help with helping designing the data structures for the O-DPOP algorithm. Ludek Cigler, thanks for being the great office mate that you have been! And of course many thanks to everybody else that has been part of the lab at some point in time: Marita Ailoma, Florent Garcin, Jason Li, Miroslav Melchiar, Adrian Petcu, Li Pu, Michael Schumacher, Radek Szymanek, Immanuel Trummer and Martin Vesely.

I would like to thank everybody that helped make my GSA experience an experience I will never forget. Thank you Maria Mateescu for getting me on board, and thanks Heather Miller for taking over the wheel after I left to start working on my thesis again. And many thanks to everybody that has been active in the GSA during my time there, you know who you are!

My life in Lausanne would not have been complete without the many friends that I made here. You are with too many to name you all, but some I would like to personally thank. I want to thank Mirjam for convincing me to move to Lausanne in the first place. I have to thank Ruud for generously introducing me into the social melting pot that was his lab, for the many enjoyable evenings watching crappy movies, and for being there in times of need. There is of course the rest of the Dutch crew, Jessica, Daan,

Joppe, Eline, Martijn, Wietske, Willem-Jan, Harm thank you all for keeping my Dutch in shape! As I said, there are too many of you to name you all, but I'll try anyway: Albrecht, Antti, Benoit, Celine, Denisa, Elena, Emile, German, Joao, Jose, Hillary, Kieran, Luis, Marius, Maya, Mayur, Pamela, Susanna, Sylvie, Tobi, Radjika, and many more, thank you all for the great time!
I would like to thank my parents for supporting me the past 29 years. Without them I would most definitely not be where I am right now! I would like to thank the rest of my family, and my friends back in the Netherlands for making my visits home worth while! And finally, I want to thank Maria. Haluan kiittää teitä siitä, että elämässäni, and for putting up with me these last few crazy months!

Lausanne, 24 Mars 2012 Brammert Ottens

Abstract

The Distributed Constraint Optimization (DCOP) framework can be used to model a wide range of optimization problems that are inherently distributed. A distributed optimization problem can be viewed as a problem distributed over a set of agents, where agents are responsible for finding a solution for their part of the problem. In the DCOP framework, the decisions the agents need to take are modeled using *decision variables*, and both the local problems and the inter-dependencies are modeled using *constraints*. In this thesis, we attack two problems related to the DCOP framework.

In the first part of this thesis, we look at coordination problems with complex, non-trivial preferences. In the DCOP literature, the notion of *complex* is used for problems where agents need to take multiple-decisions and thus own multiple decision variables. To express the fact that agents preferences are hard to obtain, we introduce the notion of a *non-trivial* local problem. The present DCOP literature has largely ignored the origin of preferences, i.e. algorithms assume that the constraints are known a-priori. In coordination problems with complex, non-trivial preferences this assumption no longer holds. In order to investigate the behavior of existing DCOP algorithms on such coordination problems, we first introduce a new DCOP model for such coordination problems, where we make an explicit distinction between the coordination part of the problem, and the local preferences of the agents. We then introduce two new benchmarks with complex, non-trivial local subproblems, and perform an evaluation of several complete and non-complete DCOP algorithms. Our results show that the O-DPOP algorithm, which elicits preferences from agents in a best-first order, outperforms both other complete approaches as also local search approaches. We show that, despite the fact that a best first order cannot always be guaranteed when dealing with non-trivial local problems, O-DPOP still finds solutions that are either better, or on par with, local search algorithms.

In the second part of this thesis, we introduce a new sampling based approach to solving DCOPs. Existing DCOP algorithms are either complete, but do not scale well, or scale well but are not guaranteed to find a solution. Our approach, inspired by the application of the UCB algorithm for the multi-armed bandit problem on trees, aims to provide method that scales well. Furthermore, given certain assumptions on the problem structure, our method is guaranteed to find good solutions. We introduce the Distributed UCT (DUCT) algorithm, that uses bounds similar to those used in the UCT algorithm, which is an adaptation of the UCB approach on trees. We introduce and

evaluate four different variations of the DUCT algorithm, DUCT-A, DUCT-B, DUCT-C and DUCT-D, that differ in the bounds that they use. To evaluate our algorithms, we compare them with existing DCOP approaches on three different types of problems, both with and without hard constraints. Our results show that, compared with other DCOP algorithms, the DUCT-D variant consistently performs well on the problems used, with respect to both solution quality and runtime. Furthermore, it consistently sends less information than the state-of-the-art DCOP algorithms, and is thus shown to be a suitable and practical algorithm for solving DCOPs.

Keywords: Distributed Constraint Optimization, Multi-agent Coordination, Sampling, O-DPOP, DUCT

Résumé

Le formalisme de l'optimisation distribuée sous contraintes (DCOP) peut être utilisé pour modéliser une large gamme de problèmes d'optimisation qui sont intrinsèquement distribués. Un problème d'optimisation distribué peut être vu comme un problème qui est distribué parmi un ensemble d'agents, qui sont responsables de trouver une solution à leurs parties respectives du problème. Dans le formalisme de DCOP, les décisions que les agents doivent prendre sont modélisées par des *variables de décision*, et leurs problèmes locaux ainsi que leurs inter-dépendances sont modélisés par des *contraintes*. Dans cette thèse, nous nous attelons à deux problèmes qui concernent le formalisme de DCOP.

Dans la première partie de cette thèse, nous considérons des problèmes de coordination avec des préférences complexes et non-triviales. Dans la littérature, l'adjectif *complexe* est utilisé pour décrire des problèmes dans lesquels chaque agent doit prendre des décisions multiples, et donc contrôle plusieurs variables de décision. Pour exprimer le fait que les préférences des agents sont difficiles à obtenir, nous introduisons le concept d'un problème local *non trivial*. La littérature actuelle a largement occulté l'origine des préférences, c'est à dire que les algorithmes supposent que les contraintes sont connues a priori. Cette hypothèse ne tient pas en présence de problèmes de coordination avec des préférences complexes et non triviales. Pour étudier le comportement des algorithmes de DCOP existants sur de tels problèmes de coordination, nous introduisons tout d'abord un nouveau modèle DCOP pour ces problèmes de coordination, dans lequel nous faisons la distinction explicite entre la partie du problème qui a trait à la coordination, et les préférences locales des agents. Nous introduisons ensuite deux classes de problèmes avec des sous-problèmes complexes et non-triviaux, sur lesquelles nous évaluons plusieurs algorithmes de DCOP complets ou non complets. Nos résultats montrent que les performances de l'algorithme O-DPOP, qui extrait les préférences des agents par ordre décroissant d'utilité, dépassent à la fois celles des autres approches complètes ainsi que des approches basées sur la recherche locale. Nos montrons que, malgré le fait que cet ordre d'extraction ne puisse pas toujours être garanti dans le cas de problèmes locaux non triviaux, O-DPOP trouve quand-même des solutions de qualité supérieure, ou au moins comparable, à celle des algorithmes de recherche locale.

Dans la deuxième partie de cette thèse, nous introduisons une nouvelle approche de résolution de DCOP basée sur l'échantillonnage. Les algorithmes de DCOP existants

sont soit complets, mais incapables de résoudre des problèmes de grande taille, soit sont capables de résoudre de tels problèmes, mais ne sont pas garantis de trouver une solution. Notre approche, inspirée de l'algorithme UCB pour la résolution de problèmes de machines à sous à plusieurs leviers basés sur des arbres, vise à produire une méthode qui permette à la fois de résoudre de grands problèmes, tout en étant garantie de trouver des solutions de bonne qualité, sous certaines hypothèses quant à la structure des problèmes. Nous introduisons l'algorithme distribué UCT, ou *DUCT*, qui utilise des bornes comparables à celles de l'algorithme UCT, qui est une adaptation de l'approche UCB à des arbres. Nous introduisons et évaluons quatre variantes de l'algorithme DUCT : DUCT-A, DUCT-B, DUCT-C et DUCT-D, qui se différencient par les bornes qu'ils utilisent. Pour évaluer nos algorithmes, nous les comparons à des approches DCOP existantes sur trois différentes classes de problèmes, tant avec que sans contraintes rigides. Nos résultats montrent que, comparée aux autres algorithmes de DCOP, la variante DUCT-D a systématiquement de bonnes performances sur les problèmes utilisés, tant en ce qui concerne la qualité de la solution que le temps de calcul. De plus, elle échange systématiquement moins d'information que les meilleurs algorithmes de DCOP existants, ce qui démontre son intérêt pratique pour résoudre des DCOPs.

Mots-clés : optimisation distribuée sous contraintes, coordination multi-agents, échantillonnage, O-DPOP, DUCT

Samenvatting

Het Distributed Constraint Optimization (DCOP) framework kan gebruikt worden voor het modelleren van een grote verscheidenheid aan optimalisatie problemen die een inherente gedistribueerde component hebben. Een gedistribueerd optimalisatie probleem kan men zien als een probleem dat verdeeld is over een verzameling van agenten, waar iedere agent verantwoordelijk is voor het vinden van een oplossing voor zijn deel van het probleem. In het DCOP framework wordt iedere beslissing die een agent kan nemen gemodelleerd door middel van een variabele. Bovendien worden zowel het locale probleem van iedere agent, alsook de inter-agent afhankelijkheden gevat in *constraints*. In deze dissertatie kijken we naar twee problemen gerelateerd aan het DCOP framework.

In het eerste deel van deze dissertatie kijken we naar coördinatie problemen met complexe, niet-triviale voorkeuren. In de huidige DCOP literatuur is het concept *complex* gereserveerd voor problemen waarin agenten meerdere beslissingen moeten maken, en dus de controle hebben over meerdere beslissings variabelen. Non-triviaal betekent in deze context dat het moeilijk is de voorkeuren van de agent te verkrijgen. De huidige DCOP literatuur heeft de oorsprong van voorkeuren grotendeels genegeerd, m.a.w. de meeste algoritmes nemen aan dat de voorkeuren a-priori bekend zijn. Bij coördinatie problemen met non-triviale locale problemen geld deze aanname niet meer. Om het gedrag van bestaande DCOP algoritmes op dit soort problemen te onderzoeken, introduceren we eerste een generiek DCOP model voor zulk soort problemen, waar we expliciet onderscheidt maken tussen het coördinatie deel van het probleem, en de locale problemen van de agenten. Daarna introduceren we twee nieuwe benchmarks met complexe, non-triviale problemen, en evalueren verschillende complete en niet-complete DCOP algoritmes op deze problemen. Onze resultaten tonen aan dat het O-DPOP algoritme beter presteert dan alle andere bestaande algoritmes. Hierbij is het vooral het feit dat O-DPOP voorkeuren opvraagt in een beste-eerst volgorde dat O-DPOP een voordeel geeft. Bovendien kan O-DPOP, ondanks dat deze beste-eerst volgorde niet altijd gegarandeerd kan worden, nog steeds goede oplossing vinden in vergelijking met lokale zoek algoritmes.

In het tweede deel van deze dissertatie introduceren we een nieuw DCOP algoritme dat gebruikt maakt van sampling. Huidige algoritmes lopen tegen verschillende problemen aan. Complete algoritmes schalen niet goed op naar grote problemen, terwijl incomplete algoritmes niet kunnen garanderen dat een oplossing gevonden wordt.

Wij introduceren het Distributed UCT (DUCT) algoritme, dat geïnspireerd is op het gebruik van UCB op bomen. UCB is een succesvol algoritme voor het oplossen van het multi-armed bandit probleem. Onder een aantal aannames op de structuur van het probleem vind DUCT gegarandeerd goede oplossingen. We introduceren vier verschillende variaties van DUCT, DUCT-A, DUCT-B, DUCT-C en DUCT-D, welke verschillen in welke limiet ze gebruiken. Voor de evaluatie van DUCT hebben we het aan de hand van drie verschillende problemen vergeleken met bestaanden DCOP algoritmes. Onze resultaten laten zien dat DUCT-D, vergeleken met andere DCOP algoritmes, beter presteert op zowel de kwaliteit van de gevonden oplossingen als op de snelheid waarmee deze oplossingen gevonden worden. Bovendien wisselt het minder informatie uit dan andere state-of-the-art DCOP algoritmes. DUCT-D is dus een goed alternatief voor zowel complete alsook incomplete DCOP algoritmes.

Sleutelwoorden: Distributed Constraint Optimization, Multi-agent Coordination, Sampling, O-DPOP, DUCT

Contents

Acknowledgements **v**

Abstract (English/Français/Nederlands) **vii**

List of figures **xv**

List of tables **xvii**

List of algorithms **xix**

List of symbols **xxvi**

1 Introduction **1**
1.1 Contributions of this Thesis 2
1.2 Outline of the Thesis 4

2 Distributed Constraint Reasoning **5**
2.1 Applications 5
2.2 Distributed Constraint Satisfaction (DCSP) 6
2.3 Complete DCSP Methods 8
2.3.1 Asynchronous Backtracking (ABT) 8
2.3.2 Improvements on ABT 9
2.3.3 Negotiation Methods 13
2.4 Incomplete DCSP Methods 14
2.4.1 Distributed Iterative Improvement (DII) 14
2.4.2 Distributed Breakout Algorithm (DBA) 15
2.5 Distributed Constraint Optimisation 15
2.6 Complete DCOP Methods 17
2.6.1 Synchronous Branch & Bound (SynchBB) 17
2.6.2 Asynchronous Distributed Optimization (ADOPT) 17
2.6.3 Dynamic Programming Optimization Protocol (DPOP) 20
2.6.4 Open DPOP (O-DPOP) 22
2.6.5 Asynchronous Forward Bounding (AFB) 23
2.6.6 No-Commitment Branch and Bound (NCBB) 24

2.7 Incomplete DCOP Methods 24
2.7.1 Local-Search Algorithms 24
2.7.2 Max-Sum Algorithm 25

3 Multi-Agent Coordination with Complex, Non-Trivial Local Preferences 27
3.1 Coordination Problems with Complex Local Preferences 28
3.1.1 DCOP Model of Coordination 31
3.2 Coordination Benchmark Problems 34
3.2.1 The Truck Task Coordination Problem 34
3.2.2 The Meeting Scheduling Problem 38
3.2.3 Contract Net Protocol 41
3.3 Experimental Evaluation 41
3.3.1 Experimental Setup 42
3.3.2 Solution Quality 43
3.3.3 Runtime 46
3.3.4 Preference Elicitation 49
3.3.5 Messages and Information 52
3.4 Conclusions 53

4 DUCT: A UCB-based Sampling Method 57
4.1 Distributed UCT 59
4.1.1 Random Sampling: The RANDOM algorithm 59
4.1.2 Termination 60
4.1.3 Normalization 61
4.1.4 Hard Constraints 62
4.1.5 Confidence Bounds: The DUCT algorithm 62
4.2 Theoretical Analysis 66
4.3 Experimental Evaluation 68
4.3.1 The Channel Allocation Problem 68
4.3.2 Experimental Setup 70
4.3.3 Experimental Results 72
4.4 Conclusions 74

5 FRODO 2 77
5.1 FRODO Architecture 77
5.1.1 Communications Layer 78
5.1.2 Solution Spaces Layer 78
5.1.3 Algorithms Layer 79
5.2 Experimental Setups 80
5.3 O-DPOP implementation 81
5.3.1 Processing received goods 82

6 Conclusions **85**
6.1 Coordination Under Complex Local Preferences 85
6.2 Solving DCOPs Using Sampling 86
6.3 Future Work 87

Bibliography **89**

Curriculum Vitae **99**

Personal Publications **101**

List of Figures

2.1 A constraint graph 7
2.2 Asynchronous Backtracking 8
2.3 A partial ordering of the variables 11
2.4 From a constraint graph to a pseudo tree 19
2.5 UTIL propagation in DPOP 22

3.1 Example of a TTC problem 35
3.2 Percentage of solutions found (1). 44
3.3 Percentage of solutions found (2) 45
3.4 Percentage of solutions found (3) 46
3.5 Distance to the optimal solution for the TTC problem 47
3.6 Solution quality for the meeting scheduling problem 48
3.7 Solution quality for the graph coloring problem 48
3.8 Simulated time in ms for the TTC problem 49
3.9 Simulated time in ms for the meeting scheduling problem (a, b, c) and the graph coloring problem (d) 50
3.10 NCCC count for the TTC problem(1) 51
3.11 NCCC count for the TTC problem(2) 52
3.12 NCCC count for the meeting scheduling and graph coloring problems . 53
3.13 Amount of information sent for the TTC problem(1) 54
3.14 Amount of information sent for the TTC problem(2) 55
3.15 Amount of information sent for the meeting scheduling and graph coloring problems 56

4.1 The constraint graph for $f_1(x_1, x_2, x_3) + f_2(x_2, x_4)$ and one of its possible pseudo-trees 58
4.2 And AND/OR graph with $\forall_i D_i = \{0, 1\}$ 58
4.3 Meeting Scheduling Results 70
4.4 Graph coloring 71
4.5 Channel allocation 72
4.6 Amount of Information Exchanged 73

5.1 General FRODO software architecture. 78

List of Tables

4.1 The DUCT variants 68

List of Algorithms

1 SynchBB for variable x_i . . . 18
2 DPOP . . . 21
3 O-DPOP UTIL propagation . . . 23
4 DSA . . . 24
5 MGM . . . 25

6 SAMPLING algorithm for variable k . . . 63
7 sample(a, k): random sampling . . . 63
8 sample(a, k): DUCT sampling . . . 66

9 SimpleJ.next() . . . 83
10 checkQ() . . . 83
11 chooseSource() . . . 83
12 SimpleJ.insert(a, list1, list2) . . . 84

List of Symbols

$\mathcal{A}$ A set of agents

a_i An agent

α The unit payment in the TTC problem

a A context

$\bar{a}$ The current context

a^* The best assignment found so far

a_k^t The context received by agent k at time t

Ap_i The access points close enough to access point i to cause interference

β Solution distribution parameter

$B_{a,d}^t$ A lower bound on the cost of choosing d, under context a at time t

C_i A coordination constraint

C The capacity of a channel

$\hat{d}_a$ The value with the lowest cost under context a

Δ The maximal channel for which interference occurs

δ Parameter for the termination bound of DUCT

δ_i The optimality gap for variable x_i

Dep A collection of dependencies

$\mathfrak{d}$ A dependency

$\mathcal{D}$ The product of the domains in $\mathscr{D}$

$\mathscr{D}$ A set of domains

D_i The domain of variable x_i

d_i	The maximal distance between the preferred time of agent a_i and the used time slot
d	A domain value
$d_{i,j}$	The distance between access point i and access point j
$\mathcal{E}$	A set of edges in a graph
ϵ	Error
$\mathcal{F}$	A set of factors
f^i	The preference reported by variable i
f_i	A factor
f	The global cost function
f^*	The optimal cost
γ	The fraction of infeasible solutions
Γ^+	The set of higher priority neighbors
Γ^-	The set of lower priority neighbors
$L^t_{a,d}$	A confidence interval
$\hat{\ell}^i$	The incorrect local preference reported by agent i
λ_a	The length of longest path
l_i	A preference constraint
list_L	The left list
list_R	The right list
ℓ^i	The sum of the constraints controlled by x_i
L	The left source
L^+	The maximal utility reported by the left source
L_-	The minimal utility reported by the left source
$\mathcal{M}$	A mapping from agents to a set of variables
m	Template upper bound
M	A set of meetings

m_i	A meeting
$\hat{\mu}^t_{a,d}$	The lowest cost found so far for value d under context a
$\hat{\mu}^t_a$	The lowest cost found so far under context a
N	The strength of the noise
n_T	The number of meeting slots
$\mathcal{P}_i$	The set of packets offered to t_i
$\mathcal{P}$	A set of packets in the TTC problem
p_i	A packet in the TTC problem
Pref	A collections of preferences
$p(x_i)$	A priority assignment to variable x_i
P_i	The signal strength of access point i
Q	An ordered list of joined assignments
$\mathbb{R}$	Real numbers
$\rho()$	The regret
R	The right source
R^+	The maximal utility reported by the right source
R_-	The minimal utility reported by the right source
S^t_a	The set of allowed domain values
S	The strength of a signal
$\mathcal{T}_i$	The set of trucks that are within customer range of p_i
t	A template
T_i	The preferred time of agent a_i
T	The number of samples taken
$\mathbb{T}$	The threshold
$\mathcal{T}$	A set of trucks in the TTC problem
t	A truck in the TTC problems

$\tau^t_{a,d}$	The number of times value d has been selected for variable x_k under context a
V	The nodes of a graph
W	The width of a channel
x^j_i	A coordination variable over coordination need j, owned by a_i
$\mathcal{X}$	A set of variables
x_i	A variable
y^t_k	The value of a sample received by agent a_k at time t

1 Introduction

In the early years of Artificial Intelligence, the field focussed on single machines, or single agents, performing tasks traditionally done by humans. The first programs operated in very stylistic and simplified worlds. The real world, however, is complex and messy. Among other things, this complexity comes about because agents are not alone in the world. At times, an agent must cooperate with other agents to solve a problem it cannot solve by itself. At other times, it has to compete with other agents over scarce resources. In this thesis, we assume that agents are cooperative, and share a common goal. The main focus of this thesis, is thus on the problem of *coordination*.

Examples of problems that require coordination abound. Take a logistics company where different trucks need to coordinate their actions to deliver goods from A to B. Or a meeting scheduling problem, where people have to agree on when to hold which meeting. These are problems where there is a common goal, that cannot be guaranteed to be obtained without the participants coordinating their actions. If truck drivers ignored the existence of other truck drivers, they might end up trying to pick up a packet that has already been picked up. Closer to home, when people would show up for meetings when they seem fit, chaos would ensue, and no meeting would ever be held.

The Distributed Constraint Reasoning (DCR) framework, has been introduced to tackle cooperative coordination problems as described above. Initially conceived for solving Distributed Constraint Satisfaction (DCSP) problems, the decisions agents can take are modeled using decision variables. An outcome is an assignment of values to these variables. The (in)feasibility of an outcome is captured by constraints on subsets of decision variables. A constraint assigns $true$ to feasible assignments and $false$ to infeasible assignments. In a more general approach, feasibility constraints are replaced by functions assigning costs or utilities to variable assignments, transforming the problem to an optimization problem. The latter approach is called Distributed Constraint Optimization (DCOP).

1.1 Contributions of this Thesis

The contribution of this thesis is twofold. The first part of this thesis focuses on what we call coordination problems with complex, non-trivial preferences. In the DCOP literature, the concept of "*complex*" is reserved for problems where agents own multiple decision variables. By *non-trivial* preferences, we mean preferences that are hard to obtain. In many coordination problems, a clear distinction can be made between the coordination constraints on the one hand, and preferences on the other, where each agent has a local problem that models its preferences. In many situations, this local problem is difficult to solve, and thus non-trivial. A good example of a problem with a non-trivial local problem can be found in the logistics setting. Agents, in this case truck drivers, need to coordinate over who is assigned which packets. The goal is to minimize the cost incurred when delivering the packets. To calculate the cost for a local packet assignment, however, agents need to solve a Vehicle Routing Problem (VRP) problem, which is known to be NP-complete. What is more, the driver also has preferences over which route to take. It can be that he prefers route A over route B because it takes him past his favorite restaurant for lunch. In general, such preferences are more difficult to formalize. We describe a DCOP model of coordination problems with preferences, that makes an explicit distinction between the preferences of the agents, and the inter-agent coordination constraints. This distinction allows the agents to use their own local solvers to generate their preferences.

We introduce two coordination problems with preferences. The first takes its inspiration from the logistics domain. Trucks must decide who picks up which packet, in order to maximize their combined profit. The local problem of each agent is both complex, and non-trivial. It is complex because an agent can be offered more than one packet, and non-trivial because calculating the utility for a particular packet assignment involves solving a VRP. The second benchmark is an adaptation from the Meeting Scheduling problem, that introduces preferences over combinations of meetings.

In the DCOP literature, the difficulty of solving local problems has largely been ignored. Early on it had been recognized that agents often need to take multiple decisions, and thus own multiple decision variables, constituting a complex local problem. The values that represent the constraints, however, are assumed to be known a-priori. In many settings this is not a realistic assumption. When users report preferences, for example, it is not reasonable to expect them to report preferences for all possible outcomes. How an algorithm elicits preferences from agents is thus an important consideration when evaluating algorithms.

From the preference elicitation literature, it is know that for certain types of problems, incremental elicitation, i.e. asking agents to report outcomes in a best-first order, minimizes the the amount of computation the agents have to perform. Generating such an order, however, is an NP-complete problem for many different preference

representations. Agents often have limited resources, which makes it impossible for them to give correct preferences on all possible solution, let alone provide a best-first order.

In this first part, we study the influence of the existence of non-trivial local problems on the performance of existing DCOP algorithms. More specifically, we want to know whether incremental elicitation provides efficiency improvements in the presence of preferences, even when a best-first order cannot be guaranteed. To that end, we evaluate an existing incremental elicitation algorithm (O-DPOP) on three types of problems. The first has preferences that are non-trivial to obtain, the second has easily computable preferences, and the third has no preferences at all.

In the second part of this thesis, we present a novel approach to solving DCOPs. The currently available algorithms can be divided in search-based, inference-based, and local-search-based algorithms. The first two are guaranteed to find an optimal solution, but pay for it in memory usage or runtime. The local-search-based algorithms use less resources, but are not guaranteed to find a solution or even to converge. In this thesis, we introduce a fourth type of algorithm, based on sampling.

When searching for an optimal solution, the balance between exploration and exploitation is important. If the scale tips in the direction of exploration, the algorithm runs the risk of walking away from the optimal solution. When the scale tips in the direction of exploitation, algorithms may pay too much attention to bad solutions. None of the current DCOP algorithms is able to strike a proper balance between the two. The DPOP algorithm, for example, performs no search at all, but simply aggregates all possible solutions. The ADOPT-based algorithms perform search, but they only disregard an option when they can prove it is suboptimal. For these algorithms, the weight lies fully on exploitation of already known information. Local-search-based algorithms like DSA perform a certain level of exploration by choosing, with a certain probability, an inferior solution. They do not, however, explore the search space in a systematic fashion. A notable exception is the O-DPOP algorithm, that tries to get around the need for exploration by relying on the best first order. It might still, however, perform an exhaustive search.

We present an approach based on methods developed for solving the multi-armed bandit problem. In the multi-armed bandit problem an agent has to play a set of rounds, where in each round it must choose a gambling machine (one armed bandit). In every round, the agent obtains a reward by playing the chosen machine, and the goal is to choose bandits in such a way that the regret of the agent over the entire sequence of plays is minimized. With regret we mean the difference between total reward of the played sequence, and the reward given by the optimal sequence. The UCB algorithm, which is proven to provide an optimal strategy for such problems, is a sampling algorithm that makes use of confidence bounds to steer the search. It succeeds in

providing a good balance between exploration and exploitation. Generalizations of the algorithm to trees are shown to be successful in providing solutions to many problems, including the problem of playing Go. In this thesis, we introduce a new algorithm for solving DCOPs called DUCT, based on confidence bounds. Evaluations of different variants of the DUCT algorithm are provided, as well as comparisons with existing approaches.

1.2 Outline of the Thesis

Chapter 2 gives an overview of the most important developments in the field of Distributed Constraint Reasoning (DCR). The sub-fields of Distributed Constraint Satisfaction (DCSP) and Distributed Constraint Optimization (DCOP), are both formally defined. We outline the most important algorithms developed in both fields, and show how they are linked together.

Chapter 3 deals with DCOPs with complex, non-trivial local problems. We provide a novel DCOP model for coordination problems with preferences. Two benchmark problems with non-trivial local problems are introduced, and for both a DCOP model is given. We provide a thorough evaluation of existing DCOP approaches in the presence of non-trivial local problems.

Chapter 4 introduces a new, confidence-based algorithm for solving DCOPs, called DUCT. We give a motivation for the algorithm, and formally define it. We provide an experimental evaluation in which we compare different variants of our algorithm with state-of-the-art DCOP approaches.

Chapter 5 provides details on the FRODO platform. FRODO is a platform for distributed constraint optimization, specifically designed for experimental purposes. This Chapter discusses the motivations we had for designing the platform, and gives its general architecture. Furthermore, some more detailed implementation issues are discussed for some of the algorithms used in this thesis. Finally, Chapter 6 concludes.

2 Distributed Constraint Reasoning

The field of Distributed Constraint Reasoning (DCR) evolved from a particular subfield of Distributed AI called Cooperative Distributed Problem Solving (CDPS) (Durfee et al., 1989). A CDPS problem is a loosely-coupled distributed network of semi-autonomous problem-solving nodes that perform sophisticated problem solving, and cooperatively interact with other nodes to solve a *single* problem (Lesser, 1990). Lesser (1990) was the first to propose to view solving a CDPS problem as a distributed state search. Based on this Yokoo et al. (1992) attempted to model a subset of CDPS problems as Distributed Constraint Satisfaction problems. This chapter will focus on the most important developments in DCR, looking at both satisfaction and optimization problems.

2.1 Applications

The DCR paradigm can be used to model many different types of real world problems. Without going into much detail, we will discuss a few of them. The traditional problem that has been used to evaluate many DCR algorithms is the graph coloring problem. It consists of a graph, where the nodes must be assigned different colors. The number of colors is limited, and the nodes must be colored such that no two neighboring nodes have the same color. This is of course a rather abstract problem, but it can be used to model many real world problems. In (Marx, 2004), for example, different types of scheduling problems (aircraft scheduling, processor scheduling and frequency allocation) are modeled as graph coloring problems.

The DCR paradigm is also well suited to model distributed scheduling problems. The Nurses Time Tabling and Transportation (NTTT) problem is one instance of a scheduling problem that has been modeled as a DCR (Solotorevsky and Gudes, 1996). The goal is to schedule the shifts of nurses, such that their availability constraints, as well as the regulations on what type of nurse should be present at what time are satisfied. The meeting scheduling problem is another instance of a distributed scheduling problem

that lends itself well to the DCR paradigm. The basic problem consists of a set of agents, which need to schedule meetings between each other. Several different types of models have been studied in (Maheswaran et al., 2003).

Sensor networks are a much studied application in the field of multi agent systems. Vinyals et al. (2010) provide an overview of most of the work done on sensor networks. The DCR community has mostly focussed on sensor networks where the sensors are directional, i.e. they can only sense in a certain direction. (Modi et al., 2001; Béjar et al., 2005; Zhang et al., 2005) propose a sensing problem where sensors need to track objects. Zhang et al. (2005) assume that one sensor is enough to track an object, allowing a graph coloring model of the problem. In (Modi et al., 2001; Béjar et al., 2005), however, each object must be assigned at least 3 sensors. Béjar et al. (2005) define a model where the objects to be tracked are the agents, while Modi et al. (2001) model the sensors as agents. In (Zivan et al., 2009), a sensor network is used to completely cover a geographical area. Again, in their model the sensors are the agents.

Other applications are Combinatorial Auctions (Petcu, 2007), Take off and Landing Slot Allocation (Petcu, 2007) and the Distributed Multiple Depot Vehicle Routing Problem (DisMDVRP), as introduced in (Léauté et al., 2010).

2.2 Distributed Constraint Satisfaction (DCSP)

Distributed Constraint Satisfaction as defined in (Yokoo et al., 1992) is a first attempt to use Constraint Satisfaction Programming (CSP) (Dechter, 2003) techniques to model distributed multi-agent environments. A formal definition can be found in Definition 1.

Definition 1 (Distributed Constraint Satisfaction Problem)**.** *A Distributed Constraint Satisfaction Problem (DCSP) is a tuple* $\langle \mathcal{A}, \mathcal{M}, \mathcal{X}, \mathscr{D}, \mathcal{F} \rangle$.

- $\mathcal{A} \triangleq \{a_1, \ldots, a_k\}$ *is a set of agents;*
- $\mathcal{X} \triangleq \{x_1, \ldots, x_n\}$ *is a set of variables, where each variable is owned by a single agent;*
- $\mathcal{M} : \mathcal{A} \mapsto \mathcal{P}(\mathcal{X})$ *is a function such that* $\mathcal{M}(a_i) \cap \mathcal{M}(a_j) = \emptyset$ *whenever* $i \neq j$*, and* $\bigcup_{a \in \mathcal{A}} \mathcal{M}(a) = \mathcal{X}$*;*
- $\mathscr{D} \triangleq \{D_1, \ldots, D_n\}$*, is a collection of finite domains, with product space* $\mathcal{D} = \prod_{i=1}^{n} D_i$*, such that* $x_i \in D_i$*;*
- $\mathcal{F} \triangleq \{f_1, \ldots, f_m\}$*, with* $f_i : D_{i_1} \times \ldots \times D_{i_{n(i)}} \rightarrow \{true, false\}$ *is a function ranging over the variables* $\{x_{i_1}, \ldots, x_{i_{n(i)}}\}$*, assigning* $true$ *to feasible assignments, and* $false$ *to infeasible assignments.*

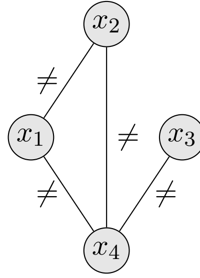


Figure 2.1: A constraint graph

A solution is an assignment to the variables $\boldsymbol{x} \in \prod_{D_i \in \mathscr{D}} D_i$ *such that* $\forall f_i \in \mathcal{F} \quad f_i([\boldsymbol{x}]_f) = true$*, where* $[\boldsymbol{x}]_f$ *is the projection of* $\boldsymbol{x}$ *to the variables in the scope of* f_i*.*

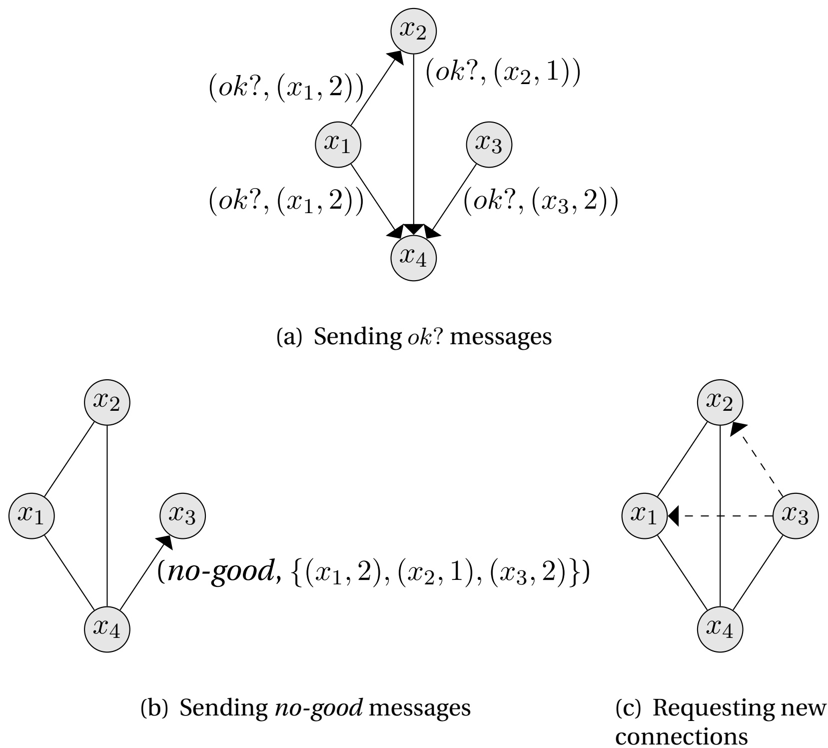
It must be said that solving a DCSP is not equivalent to parallel/distributed solving of a CSP. Solving a CSP using parallel/distributed methods is done from an efficiency point of view. The problem is distributed in such a way as to most efficiently solve the global problem. In a DCSP, however, the distribution is *fixed*, and cannot be changed. The main goal of the DCSP community is thus to investigate how to solve a problem, given a pre-defined distribution.

The agents in $\mathcal{A}$ are assumed to be *autonomous* entities that can only communicate with each other through the sending of messages. Importantly, they are assumed to have full knowledge of all the constraints they participate in. Messages between agents are delivered within a finite amount of time, and in the order in which they have been sent. For ease of exposition, most algorithms discussed in this section have added the assumption that each agent controls only a single variable, i.e. $\forall a \in \mathcal{A}\, |\mathcal{M}(a_i)| = 1$.

The structure of a DCSP is captured by the constraint graph. The constraint graph represents the connectivity of the problem, i.e. it shows which variables are directly connected to which other variables through constraints.

Definition 2 (Constraint Graph)**.** *Given a DCSP* $\langle \mathcal{A}, \mathcal{M}, \mathcal{X}, \mathscr{D}, \mathcal{F} \rangle$*, its constraint graph* $\langle \mathcal{X}, \mathcal{E} \rangle$ *is such that each node represents a variable, while there is an edge* $(x_i, x_j) \in \mathcal{E}$ *between two variables if they share a constraint.*

Take as an example a problem consisting of the variables $x_1 \in \{1, 2, 3\}$, $x_2, x_3 \in \{1, 2\}$ and $x_4 \in \{3, 4\}$, with the constraints $x_1 \neq x_2$, $x_1 \neq x_4$, $x_2 \neq x_4$ and $x_3 \neq x_4$. The corresponding graph is shown in Figure 2.1. The constraint graph is an important concept in the DCR literature, and many approaches use it to structure their search for a solution. The different approaches for solving DCSPs can be roughly divided into three different types: backtracking-based methods, negotiation-based methods and local-search methods. The first two are, in general, complete methods while the local-search methods are not necessarily guaranteed to find a feasible solution.



(a) Sending $ok?$ messages

(b) Sending *no-good* messages

(c) Requesting new connections

Figure 2.2: Asynchronous Backtracking

2.3 Complete DCSP Methods

In this Section we discuss several complete DCSP methods. A DCSP method is complete when it is guaranteed to find a solution, if it exists. We start with the Asynchronous Backtracking algorithm Yokoo et al. (1992). It is the first complete and distributed algorithm for solving DCOPs. The algorithm is based on the Backtracking (BT) method (Mackworth, 1987), and is in many ways the backbone for most of the DCSP algorithms that have been developed.

2.3.1 Asynchronous Backtracking (ABT)

ABT allows agents to run concurrently, and asynchronously make decisions on their variable's values. Each variable x is assigned priority value $p(x)$. These priority values define a total order on the variables, such that $x_i \succ x_j$ if $p(x_i) > p(x_j)$. Take, as an example, the problem defined in Figure 2.1, and assume the following order $x_1 \succ x_2 \succ x_3 \succ x_4$. Each agent starts by choosing a value for its variable, and reports it to all neighbors with a lower priority value using an $ok?$ message (Figure 2.2(a)).

An agent stores the values reported by the $ok?$ messages in its *agent_view*. Whenever an agent receives a new $ok?$ message, it checks, based on the constraints it controls,

whether its current value is consistent with its the current *agent_view*. A constraint is controlled by the lowest priority variable participating in the constraint. If the current value is consistent, it simply sends out an $ok?$ message containing this value to all its lower priority neighbors. Otherwise, it tries to find a new value for its variable that is consistent, and reports this new value to its lower priority neighbors using an $ok?$ message. If such a value does not exist, it means that one of the higher priority agents needs to change its value, i.e. the algorithm needs to *backtrack*. An agent initiates a backtrack by reporting its current *agent_view* as a *no-good* to the lowest priority agent in the *no-good* (Figure 2.2(b)).

When an agent receives a *no-good*, it considers it as a new constraint between it and the other variables in the *no-good*. If the *no-good* contains a variable that is not known by the receiving agent, i.e. there is no link between the variable controlled by the receiving agent and the variable in the *no-good*, a new link is added (Figure 2.2(c)). Over each new link an $ok?$ message is sent to update the agent's *agent_view*.

The algorithm was shown to be sound and complete. When an agent finds an empty *no-good*, it knows that there is no solution to the problem and the algorithm can terminate. When a solution has been found, none of the agents will be sending messages. In this case, a distributed termination algorithm (Chandy and Lamport, 1985) must be used to recognize the situation.

2.3.2 Improvements on ABT

The ABT algorithm has been the starting point for many of the subsequent algorithms developed for DCSP problems. This section looks at the different improvements that have been made.

Dynamic Orderings

The main disadvantage of the ABT algorithm is that its variable ordering is fixed. Because of the fixed order, the algorithm might have to perform a full search before it finds the variable that is responsible for the constraint violation. The Asynchronous Weak Commitment (*AWC*) algorithm (Yokoo, 1995), inspired by the Weak Commitment (WC) algorithm (Yokoo, 1994), tries to remedy this by introducing a dynamic order. It does this by dynamically changing the priory order $p()$ as follows; whenever a variable x_i discovers a *no-good*, it sets the variable's priority value to $k + 1$, where k is the maximal priority value among the variables occurring in its *agent_view*. In (Yokoo, 1995; Armstrong and Durfee, 1997), the AWC algorithm is extended to agents with complex local problems, where a complex local problem is a problem where agents own multiple variables.

Zivan and Meisels (2005) introduces a different approach to dynamically ordering the variables. Their algorithm *ABT_DO* allows agents to propose a reordering of lower priority variables. They tested different heuristics, and found that changing the order whenever a *no-good* is received provided the best results..

No-good Learning

Both ABT and AWC use a very crude way to generate *no-good*s. In (Yokoo and Hirayama, 2000), a new method for *no-good*-learning is introduced, called resolvent learning, that combines a set of *no-good*s to a single *no-good*, thus saving space. Whenever a variable x_i reaches a dead-end, it finds, for each possible value in its domain, a *no-good* that prohibits this value. It then aggregates these *no-good*s into a single *no-good*. If a value is prohibited by multiple *no-good*s, the smallest *no-good* is chosen, and ties are broken by choosing the *no-good* with the highest priority. The selected *no-good*s are combined by removing references to x_i, and taking the union of the remaining variable assignments.

It has been shown that resolvent-based learning not only speeds up AWC, but it also greatly reduces the number of constraint checks that must be performed. However, AWC with resolvent learning is still susceptible to what is called the *no-good*-explosion. For some problems, many *no-good*s must be generated, putting both a big burden on memory usage, and making it more difficult to evaluate new variable assignments. One way to reduce the burden, is to only store *no-good*s of a certain size. Which size works well, however, depends on the problem at hand. A smaller size might reduce the number of *no-good*s stored, but it can also increase the number of cycles needed to find a solution.

Preventing the Addition of Links

Whenever an agent in ABT receives a *no-good* that contains a variable it does not know, it must request a link with this variable to maintain consistency, adding to the complexity of the problem. Different attempts have been made to remove the need to make connections between agents that are not connected in the original problem. In (Hamadi, 1998), a partial ordering is used to overcome the need to add new links. Although the algorithm they describe (*DIBT*) has been shown not to be complete (Bessière et al., 2003), the idea of using a partial order is an important idea that has been used in subsequent algorithms, both for DCSPs and their optimization variants.

The partial order is constructed as follows. They start with a total ordering on the variables, and each variable splits its neighbors into a set with higher priority values (parents) Γ^+, and a set with lower priority values (children) Γ^-. When two nodes have

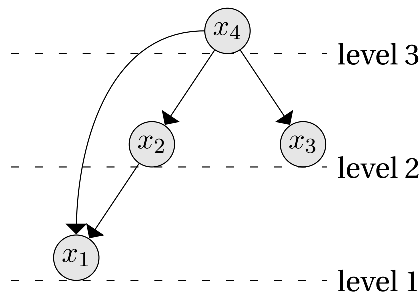


Figure 2.3: A partial ordering of the variables

the same priority value, their identifiers are used to break the tie. Next, each variable with $\Gamma^+ = \emptyset$ assigns itself to level 1. All other variables set themselves to the maximal level of their children plus 1. If we take $p()$ to be the number of connections a variable has, the problem as defined in Figure 2.1 is ordered as shown in Figure 2.3.

Note that nodes in the same level are independent from each other. DIBT then uses a backtracking mechanism without *no-good*-learning, which is memory efficient, but unfortunately makes the algorithm incomplete. The Distributed Dynamic Backtracking (*DDB*) algorithm (Bessière et al., 2003, 2005) attempts to fix this, by adding a *no-good* learning strategy. The basic idea is to create a *no-good* for every value that is inconsistent with the current *agent_view*. Whenever a variable reaches a dead-end, the *no-good*s are resolved into a single *no-good*, and reported to the lowest priority ancestor of the variable.

Concurrent Search

ABT allows concurrency search by allowing the agents to asynchronously change their values. In order to remain complete, agents need to store *no-good*s that are discovered while searching. In Zivan and Meisels (2006) another form of concurrency is introduced, in the form of the *ConcDB* algorithm. Multiple different search processes are run concurrently on the agents, exploring different parts of the search space. Each search process is represented by a single, consistent partial assignment (CPA). This CPA then moves through the constraint network as a token. This means that for each search process, only a single agent at a time is allowed to add a value. This removes the need to store *no-good*s, and experiments show that ConcDB uses less constraint checks and sends less messages than ABT.

Forward Checking

In backtracking-based algorithms, a constraint is evaluated by the lowest agent that participates in the constraint, for the simple reason that a constraint cannot be evaluated

until all variables have been assigned values. This means, however, that when an agent chooses a value, it has no way of knowing whether it will be compatible with variables further down the hierarchy. In order to assure completeness and termination, agents have to store *no-good*s. In the centralized case, the Forward Checking (FC) (Haralick and Elliott, 1979) algorithm has been developed to counter this. Distributed Forward Checking (*DIFC*) (Brito and Meseguer, 2003) attempts to use forward checking for solving DCSPs. DIFC uses the same ordering as in DIBT. However, it gives control of the constraints to the highest priority variable in the ordering, and instead of sending a value assignment for its own variable, it reports the influence of its decisions on domains of its lower priority neighbors. The main disadvantage of DIFC is that calculating the domains of lower priority variables is an expensive operation. Experiments have corroborated this.

Dynamic Ordering Distributed Forward Checking (*DODFC*) (Meisels and Razgon, 2001) takes another approach to using forward checking. It assumes, contrary to most other DCSP algorithms, that all agents communicate with all other agents from the start. One variable is selected, and the controlling agent is to function as a so called DECIDING_AGENT (DA) at the beginning of the algorithm. Whenever an agent becomes the DA, it sends the domain values that are consistent with its *agent_view* to all agents that share constraints with it. Each agent then computes, for each reported value, an evaluation, and reports this to the DA. The DA then aggregates these evaluations into a single number for each value, selects the best value and reports this to the other agents. Agents then refine the domains of their variables, and based on some predefined heuristic make a suggestion to the DA as to which variable should become DA next. An example of such a heuristic is to propose the variable with the smallest domain. The DA then selects the next variable, and makes this variable the new DA.

Asynchronous Forward Checking (*AFC*) (Meisels and Zivan, 2007) is very similar to DODFC, in that at any time there is only one agent that decides on the value of its variable. The idea is to maintain a CPA, which is used as a token that is sent over the network. An Initial Agent (IA), initializes the CPA, and based on a heuristic it then sends it to a first agent. Whenever an agent receives a CPA, it updates its *agent_view*, chooses a value and then sends a FC_CPA to all unassigned variables. The FC_CPA is an exact copy of the current CPA. When an agent receives an FC_CPA, it checks whether it is consistent with its constraints, and if not it returns a Not_Ok message and a new value is chosen. Whenever an agent is not able to choose a consistent value, the agent backtracks by sending the CPA up to the last agent that has assigned a variable. For bigger, more complex problems AFC outperforms both ABT and DDB both in terms of the number of constraint checks performed and the number of messages sent. Distributed Dynamic BackJumping (DDBJ) (Nguyen et al., 2004) improves on AFC by introducing more concurrency in the algorithm.

Private Constraints, Shared Variables

In the general DCSP model, the variables are controlled by the agents, while the constraints are shared. That is, each agent controls its own variables, but it has full knowledge of each constraint in which at least one of its variables participates. In (Silaghi et al., 2000a) it is argued that this assumption does not hold in negotiation settings, where it is the constraints that are private, while the decisions are not. They introduce the Asynchronous Aggregation Search (*AAS*) algorithm to deal with such situations. In AAS, each agent is responsible for a set of constraints, and different agents can propose values for the same variable.

AAS assumes an ordering over the agents, and each agent is only connected with agents with which it shares a variable. A backtrack search is performed, but instead of propagating single values, agents in AAS propagate aggregated tuples of values, where the messages are sent from agents with lower priorities to agents with higher priorities. These aggregated tuples of values are proposals for the domains of variables, and each agent's *agent_view* now contains a set of possible values for each variable the agent has to coordinate over. Whenever an agent finds that no combination of values for the variables satisfies its constraints, it reports a *no-good* to the relevant agents. As in ABT, when receiving a *no-good* an agent might add links to other agents whose variables it has become interested in. Three instantiations of the algorithm (AAS0, AAS1 and AAS2) are proposed, that differ in the way they deal with *no-good*s. AAS2 is similar to ABT in that it records all *no-good*s while AAS1 and AAS0 forget some *no-good*s when new information on variable domains is received. Furthermore, AAS0 performs a merge of all known *no-good*s by using a simple relaxation rule. Experiments, however, showed no significant difference with ABT. MHDC (Silaghi et al., 2000b) improves the AAS approach by combining it with a distributed bound-consistency approach.

2.3.3 Negotiation Methods

Mailler and Lesser (2003) introduced a mediation-based approach to solving DCSPs, called Asynchronous Partial Overlay (*APO*). In backtrack-based algorithms like ABT and AWC agents are only aware of their own constraints, although they can learn new constraints with non-neighboring variables through *no-good*s. They are thus to learn that certain value combinations are not allowed, but not why they are not allowed. In APO agents are allowed to gather more information on the constraints of their neighbors.

An order is assumed over the variables, although this order is not used in the same way as in ABT or AWC. Agents communicate the values for their variables to all their neighbors. Whenever a conflict arises that an agent cannot solve by changing its value, and this agent has the highest priority of all agents participating in the constraint,

the agent starts a *mediation* with all its neighbors to minimize the number of constraint violations. During the mediation, the agent that started the mediation collects information on the values of other variables. That is, each value of the variables in its *agent_view* must report whether it is consistent with the current *agent_view*, and if not, report at least one conflicting value of another variable. It then tries to find a solution that minimizes the number of constraint violations. Note that during mediation an agent can discover new variables that influence its own local solution. New links are created between such new agents and the mediator agent, which will be taken into account in the next mediation. In (Grinshpoun and Meisels, 2008) however, it is shown that APO is not complete. They propose a modified version, called CompAPO, that plugs the holes in APO.

2.4 Incomplete DCSP Methods

Incomplete methods are methods that are not guaranteed to find a solution to the problem. The main class of incomplete methods consists of local-search methods. Local-search methods are methods that try to find better solutions by making local changes based on only local information.

2.4.1 Distributed Iterative Improvement (DII)

Hirayama and Toyoda (1995) developed the Distributed Iterative Improvement (*DII*) algorithm. Each agent picks an initial value for its variable, and then informs its neighbors of this value. Agents collect value information in an *agent_view*, and check for constraint violations. When a constraint violation occurs, an agent starts a *negotiation* with neighboring agents in order to reduce the number of constraint violations. It does this by communicating both the current number of violations, and the minimal number of violations it can obtain. An agent is allowed to change the assignment to its variable when either none of its neighbors have constraint violations, or if the improvement it can obtain is the highest among all its neighbors (ties are broken through comparison of the variable identifiers).

DII basically performs a distributed version of hill climbing, and it is thus possible for DII to get stuck in a local minimum. Whenever DII encounters a situation where no agent is able to decrease the number of constraint violations by changing its value[1], it forms a coalition of agents whose variables participate in the violated constraints. One agent in the coalition is selected as *manager* of the coalition. The manager collects all variables, domains and constraints in the coalition and solves the resulting CSP using some strategy. They introduce two types of strategies, i.e. the selfish and the altruistic strategy. In the selfish strategy, a manager chooses an assignment that satisfies all of

[1] Note that this is a necessary, but not sufficient condition for being in a local minimum

its own constraints, and asks the other agents in the coalition if the new assignment is ok. The altruistic strategy, on the other hand, performs a search in the full CSP of the coalition, and chooses the assignment that minimizes the number of constraint violations. Experiments show that the altruistic strategy works better than the selfish strategy. However, when the local CSP becomes too complex, the altruistic strategy becomes too expensive. Note that no new links are created between agents, as would be the case in APO.

2.4.2 Distributed Breakout Algorithm (DBA)

The DII algorithm is able to jump out of local minima. However, when the coalitions become too big the algorithm breaks down. Distributed Breakout (*DBA*) (Yokoo and Hirayama, 1996) is inspired by the Breakout algorithm (*BA*) (Morris, 1993). In BA, pairs of variable assignments are given weights, and the goal is to minimize the sum of the weights of violated constraints. When a local minimum is encountered, the weights for the value pairs that violate a constraint are increased, in effect removing the local minimum.

In DBA, agents choose initial values and report them to their neighbors. Next, based on their current *agent_view*, they report the possible improvements they can provide by changing their values. As in DII, an agent is only allowed to change its value when its improvement is higher than the possible improvements of its neighbors. A local minimum is a property of the global assignment, and no single agent is able to recognize a local minimum. For that reason, the notion of a *quasi-local minimum* is defined. An agent is in a quasi-local minimum when it violates some constraint, and neither it, nor any of its neighbors, is able to improve the assignment. When an agent detects a quasi-local minimum, it increases the weights for the constraint violations.

To detect when to terminate, each agent has a *termination_counter*. Its value is increased from d to $d+1$ when both its value, and its neighbors' values are consistent, and the *termination_counter* of all its neighbors is equal to or bigger than d. Whenever d reaches the total number of variables in the problem, the agent knows it can terminate. It is thus assumed that each agent knows an upper bound on the number of agents present in the problem.

2.5 Distributed Constraint Optimisation

Distributed Constraint Optimization (DCOP) is a generalization of the DCSP paradigm, where constraints do not define which solutions are feasible or not, but attach a value to each outcome. Distributed Optimization has been first addressed in (Liu and Sycara, 1995). However, they focus on solving centralized problems in a distributed fashion,

while in the DCOP paradigm the focus is on solving inherently distributed problems. Yokoo (1993); Hirayama and Yokoo (2003) proposed extensions of the DCSP paradigm to over-constrained problems. In (Yokoo, 1993) this is done by assigning importance values to constraints, where the goal is to find a solution that minimizes the importance of the constraints that are violated. (Hirayama and Yokoo, 2003), on the other hand, extends the Partial Constraint Satisfaction (PCSP) domain, where there is a distance metric defined over subproblems of the original problem, to a distributed setting. Neither approach, however, is able to model a wide array DCOP problems.

The DCOP paradigm, as defined below, has been introduced in (Modi et al., 2003). Constraints are seen, not as predicates that define the feasibility of a variable assignment, but as functions assigning a value to each variable assignment.

Definition 3 (A DCOP problem)**.** *A discrete distributed constraint optimization problem is a tuple $\langle \mathcal{A}, \mathcal{M}, \mathcal{X}, \mathscr{D}, \mathcal{F} \rangle$ where*

- $\mathcal{A} \triangleq \{a_1, \dots, a_k\}$ *is a set of agents;*
- $\mathcal{X} \triangleq \{x_1, \dots, x_n\}$ *is a set of variables, where each variable is owned by a single agent;*
- $\mathcal{M} : \mathcal{A} \mapsto \mathcal{P}(\mathcal{X})$ *is a function such that* $\mathcal{M}(a_i) \cap \mathcal{M}(a_j) = \emptyset$ *whenever* $i \neq j$*, and* $\bigcup_{a \in \mathcal{A}} \mathcal{M}(a) = \mathcal{X}$*;*
- $\mathscr{D} \triangleq \{D_1, \dots, D_n\}$*, is a collection of finite domains, with product space* $\mathcal{D} = \prod_{i=1}^{n} D_i$*, such that* $x_i \in D_i$*;*
- $\mathcal{F} \triangleq \{f_1, \dots, f_m\}$ *is a set of constraints where each constraint* $f_i : \mathcal{D}_i \to \mathbb{R} \cup \{\infty\}$ *depends on* $n(i)$ *variables, with* $\mathcal{D}_i \triangleq D_{i_1} \times \dots \times D_{i_{n(i)}}$*. Let* $\boldsymbol{x} = \{x_1, \dots, x_n\}$*, then* $[\boldsymbol{x}]_{f_i}$ *is used to denote the projection of* $\boldsymbol{x}$ *to the subspace on which the* i*-th constraint is defined, and* $var(f_i)$ *to denote the variables in the range of* f_i*.*

A feasible assignment $\boldsymbol{x} \in \prod_{D_i \in \mathscr{D}} D_i$ *is an assignment to all variables such that*

$$f(\boldsymbol{x}) \triangleq \sum_{i=1}^{m} f_i\left([\boldsymbol{x}]_{f_i}\right) \neq \infty, \tag{2.1}$$

The objective is to find a feasible assignment that minimizes the sum of the constraints, with the minimum value being $f^*(\mathcal{D}) \triangleq \min_{\boldsymbol{x} \in \mathcal{D}} f(\boldsymbol{x})$.

Although Definition 3 defines a DCOP as a minimization problem, this in no way restricts the modeling power of the framework. A maximization problem can simply be transformed into a minimization problem by flipping the sign of all constraint values. Most DCOP algorithms, however, do take a more restricted view of DCOPs. Outside of

the algorithms in the DPOP family, all algorithms assume values between 0 and ∞, so as to exploit the monotonicity properties that come with this assumption. Without rescaling a problem, most algorithms are thus not able to handle problems with both costs and utilities.

2.6 Complete DCOP Methods

This section introduces the complete DCOP methods. All methods presented in this section are guaranteed to find the optimal solution.

2.6.1 Synchronous Branch & Bound (SynchBB)

SynchBB (Hirayama and Yokoo, 2003) has originally been developed for solving DCSPs, although it can easily be adapted to solving DCOP problems. SyncBB is a synchronous search algorithm, altogether not too different from AFC. It thus assumes that constraints relate costs to variable assignments. Variables are ordered using a linear order, and only a single agent is active at a time. The first agent in the ordering initiates the algorithm by sending a path to the next agent, containing a variable assignment to the variable of the first agent (line 8, Algorithm 1). Every subsequent agent that receives a partial path (equivalent to a CPA in AFC), chooses a value for its own variable, evaluates its constraints and sends the path, including the cost of the path, further down the ordering (lines 26-27). Whenever the final agent is able to assign a value to its variable, the cost of the entire path is broadcasted to all the agents as the best value found so far (lines 20-24). Agents now use this value as an upper bound to prune partial paths that already have a higher cost.

Whenever an agent is not able to assign a value to its variable, i.e. the costs of all values is above the upper bound, a BACK message is sent (lines 24 and 29).

2.6.2 Asynchronous Distributed Optimization (ADOPT)

The ADOPT algorithm (Modi et al., 2003) is the first asynchronous, complete algorithm that is able to solve DCOP problems. ADOPT is a distributed, asynchronous, search algorithm. Similar to SynchBB, ADOPT assumes that the values of all constraints are between 0 and ∞, and it tries to find the variable assignment that minimizes the sum of constraints.

ADOPT uses a variable hierarchy similar to what is used in DIBT. Given the constraint graph of a problem, ADOPT finds a so called *pseudo tree*, and uses this as a partial order on the variables.

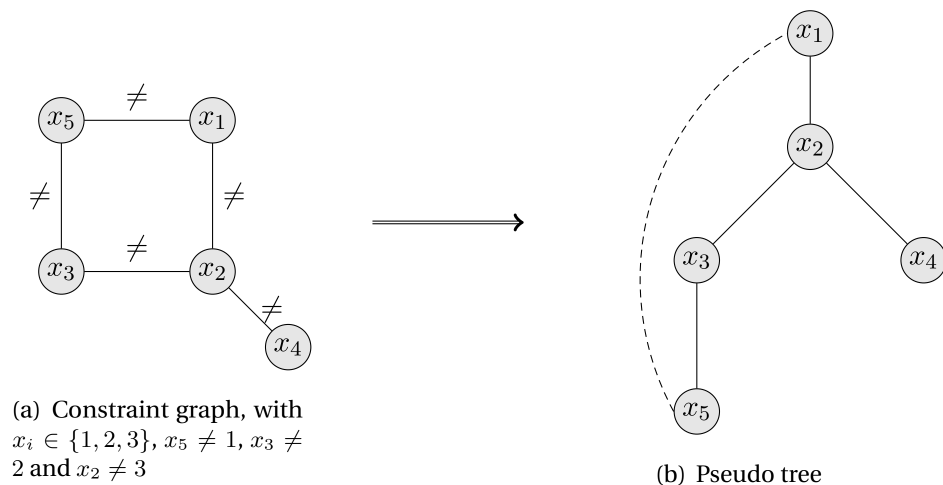
Definition 4 (Pseudo Tree)**.** *Given a graph $\langle V, \mathcal{E} \rangle$, a pseudo tree is a spanning tree $\langle V, \mathcal{E}' \rangle$*

Algorithm 1: SynchBB for variable x_i

```
input: A fixed, known linear order on the variables x_1 ≻ x_2 ≻ ... ≻ x_n
1  D'_i ← D_i;
   // The sum of the constraints controlled by i;
2  ℓ^i(x_i, ·) ← Σ_{f ∈ {f' ∈ F | x_i ∈ var(f') ∧ ∀x' ∈ (c') x' ≻ x_i}} f(x_i, ·);
3  f^i ← 0 // the cost of the current partial assignment;
4  f* ← ∞ // the cost of the best assignment found so far;
5  a* ← null // the best assignment found so far;
6  ā ← null // The current assignment;
7  if i = 1 then // x_1 starts the search
8  |   d_1 ← first element from D'_i that is feasible;
9  |   if d_1 ≠ null then send((CPA, d_1, ℓ^1(d_1))) to x_2;
10 |   else broadcast(INFEASIBLE)
11 while there is a message M to process do
12 |   if M is (UB, a, c) then
13 |   |   f* ← c, a* ← a;
14 |   else
15 |   |   if M is (CPA, a, c) then
16 |   |   |   D'_i ← D_i, ā ← a, f^i ← c;
17 |   |   else if M is BACK then
18 |   |   |   D'_i ← D'_i \ {d_i};
19 |   |   if i = n then
20 |   |   |   d_i ← arg min_{d' ∈ D_i} ℓ^i(d', ā);
21 |   |   |   a* ← (â, x_i), f* ← f^i + ℓ^i(a*);
22 |   |   |   broadcast((UB, a*, f*));
23 |   |   |   if f* = 0 then broadcast(TERMINATE);
24 |   |   |   else send(BACK) to x_{n-1};
25 |   |   else
26 |   |   |   d_i ← value from D'_i such that f^i + ℓ^i(x_i, â) < f*;
27 |   |   |   if d_i ≠ null then send((CPA, (a, d_i), f^i + ℓ^i(d_i, a)));
28 |   |   |   else if i = 1 then send(TERMINATE);
29 |   |   |   else send(BACK) to x_{i-1};
```

such that whenever $(a, b) \in \mathcal{E}$ *and* $(a, b) \notin \mathcal{E}'$, a *and* b *must be in the same branch.*

Using the pseudo tree, ADOPT can search different parts of the problem concurrently. An example is given in Figure 2.4. The solid lines represent the spanning tree of the original constraint graph, while the dotted lines represent *back edges*. A back edge is an edge that is present in the original graph, but not part of the spanning tree. A pseudo tree must be such that a back edge exists only between a node and one of its descendants or ancestors. The *separator* of a node i in the tree is the set of ancestor nodes that must be removed from the problem to completely separate the problem



(a) Constraint graph, with $x_i \in \{1,2,3\}$, $x_5 \neq 1$, $x_3 \neq 2$ and $x_2 \neq 3$

(b) Pseudo tree

Figure 2.4: From a constraint graph to a pseudo tree

rooted at i from the rest of the problem. The separator of x_3, for example, consists of $\{x_1, x_2\}$. The *width* of a pseudo tree is the size of the largest separator.

Algorithms for finding a pseudo tree can be found in (Hamadi, 1998; Léauté, 2011), and generally consist of two phases. The first phase selects a root, while the second phase performs a depth-first traversal of the constraint graph, starting from the root. During construction of the pseudo tree, the following information is gathered for each node i:

- $parent_i$, the parent of i;
- $children_i$, the children of i;
- $pseudo_children_i$, the pseudo children of i, i.e. all lower priority neighbors of i that are not children;
- $pseudo_parents_i$, the pseudo parents of i, i.e. all higher priority neighbors of i that are not its parent;
- sep_i, the separator of i.

Each variable in ADOPT starts the search at the same time, by selecting the domain value that minimizes the constraints it is responsible for. A variable is assigned the constraints in which it is the lowest participating variable. Each agent then sends this value to its children and pseudo children using a VALUE message. A VALUE message contains only a single variable assignment. Upon receiving a VALUE message, an agent responds by sending a COST message to its parent, that contains both a lower bound and an upper bound on the actual cost for the current agent view. Since all agents

change their values asynchronously, a COST message also contains the agent view for which it has been generated. Whenever an agent receives new VALUE or COST message, it resets all lower and upper bounds collected under an incompatible context. In order to prevent the algorithm from having to rediscover parts of the search space that are known not to be interesting, a *backtrack threshold* is used. This threshold is communicated between agents using a THRESHOLD message.

Using the upper bound and the threshold, ADOPT is able to recognize when the optimal solution has been found, and thus when it is to terminate. Although experiments have shown that ADOPT outperforms SynchBB, the asynchronous nature of ADOPT can still result in the generation of a large amount of messages. In fact, the number of messages exchanged by the ADOPT algorithm is exponential in the depth of the tree. Algorithms like BnB-ADOPT (Yeoh et al., 2010) and ADOPT-ng (Silaghi and Yokoo, 2006), improve ADOPT in different aspects.

2.6.3 Dynamic Programming Optimization Protocol (DPOP)

Where all previously described algorithms attempt to solve problems using search, DPOP (Petcu and Faltings, 2005a), see Algorithm 2, uses *Dynamic Programming* methods. DPOP is an adaptation of the general bucket elimination scheme (Dechter, 2003) to a distributed setting. Similar to ADOPT, it operates on a pseudo tree, allowing it to solve separate parts of the problem concurrently. After generating a pseudo tree, DPOP proceeds in two distinct phases. In the first phase, the UTIL propagation phase, cost information is collected in a bottom up fashion. In the second phase, the VALUE propagation phase, variables are assigned values in a top-down manner. Do note that, contrary to SynchBB and ADOPT, DPOP also supports maximization problems and puts no restrictions on the signs of the constraint values.

UTIL Propagation

In the UTIL propagation phase, agents collect cost information from their children, combine it with their own local problems, and then report this to their parents. This phases is started by the variables on the leaf nodes of the pseudo tree. They create UTIL messages by summing up (*joining*) all their constraints, and then *projecting out* their own variables (line 3). In general, if m_j is a UTIL message received by child j, then x_i calculates m_i as follows

$$m_i(\cdot) = \min_{d \in D_i} [\sum_{f \in \mathcal{F}_i} f(d, \cdot) + \sum_{j \in children_i} m_j(d, \cdot)]$$

where $\mathcal{F}_i = \{f' \in \mathcal{F} | x_i \in var(f') \wedge var(f') \cup children_i \cup pseudo_children_i = \emptyset\}$. As in ADOPT, each variable in DPOP evaluates only the constraints in which it is the lowest

Algorithm 2: DPOP

```
1  initialization
2      if children_i = ∅ then
3          m_i(·) ← min_{d∈D_i} Σ_{f∈F_i} f(d, ·);
4          send UTIL(m_i) to parent_i;
5  when received(UTIL(m_j))
6      Store m_j;
7      if all children reported a UTIL message then
8          m_i(·) ← min_{d∈D_i} [Σ_{f∈F_i} f(d, ·) + Σ_{j∈children_i} m_j(d, ·)];
9          if i is the root then
10             d* ← min_{d∈D_i} m_i(d);
11             send VALUE({x_i = d*}) to all children;
12         else
13             send UTIL(m_i) to parent_i;
14 when received(VALUE(a))
15     d* ← min_{d∈D_i} m_i(d, a);
16     to every child j, send VALUE([a ∪ {x_i = d*}]_{sep_j});
```

priority variable. An example can be found in Figure 2.5, based on the problem as defined in Figure 2.4(a). The size of a UTIL message sent by variable x_i depends on the size of its separator. In fact, the size of a UTIL message is exponential in the size of the separator. So even though DPOP sends only a linear number of messages, the sizes of these messages are exponential in the numbers of variables that occur in the messages. The complexity of DPOP is thus determined by the width of the pseudo tree it finds. Unfortunately, finding the pseudo tree that minimizes the tree width is an NP-complete problem. Hence, approximate methods as described in (Hamadi, 1998; Léauté, 2011) can be used to generate a pseudo tree.

VALUE Propagation

When the root node has received UTIL messages from all children (line 9), it knows, for each of its domain values, the minimal global cost the entire problem can achieve given the current *agent_view*. It chooses the value that minimizes the global cost, and reports this to its children using a VALUE message, in effect starting the VALUE propagation phase. Upon receiving a VALUE message, an agent assigns the value to its variable that minimizes the cost of its subproblem, and reports this, together with the context in the VALUE message it received, to all its children. This way all variables are able to make optimal decisions, and at the end of the VALUE propagation phase a solution has been found.

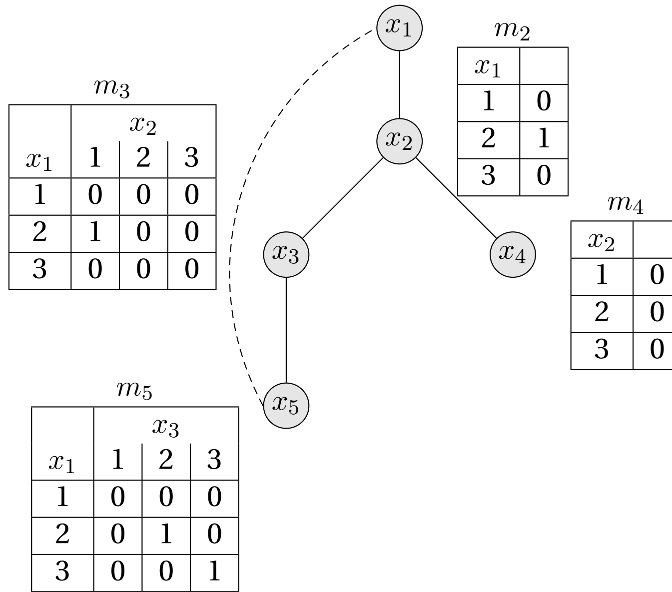


Figure 2.5: UTIL propagation in DPOP

2.6.4 Open DPOP (O-DPOP)

Although DPOP sends only a linear number of messages, the UTIL messages can still be large. Furthermore, many of the entries in the UTIL messages are not needed for finding the optimal solution. In (Petcu and Faltings, 2006), the O-DPOP algorithm is introduced. O-DPOP is based on the open constraint programming paradigm introduced in (Faltings and Macho-Gonzalez, 2005). The basic idea is that the agents do not need to have information on all possible assignments in their separators to find the optimal solution, if they receive assignments in a best first order. This best first order is used to calculate upper bounds on unseen assignments, which can then be used to recognize the optimal solution. Similar to DPOP, O-DPOP works with a pseudo tree and consists of a UTIL propagation phase, where utilities are propagated in a bottom up manner, and a VALUE propagation phase to find an optimal variable assignment. The VALUE propagation phase is equal to the VALUE propagation phase in DPOP.

UTIL Propagation

The UTIL propagation (Algorithm 3) phase is initiated by the root node. It starts by sending an ASK message to its children. Every node, whenever it receives an ASK message, either reports the best assignment to the variables in its separator it has not yet sent (lines 19 - 20), or, if it does not know what this assignment is, it sends an ASK message to all its children (line 24). When it has reported all assignments in its separator, it sends a DONE message to its parent (line 22).

Algorithm 3: O-DPOP UTIL propagation

```
1  initialization
2  |  replied = true;
3  |  initialize goodStore;
4  |  if x_i is the root variable then
5  |  |  send an ASK message to all children;
6  when received(ASK)
7  |  reply();
8  when received(GOOD(good, child))
9  |  goodStore.store(good);
10 |  if ¬replied then
11 |  |  reply();
12 when received(DONE(child))
13 |  goodStore.childFinished(child);
14 |  if ¬replied then
15 |  |  reply();
16 procedure: reply()
17 |  good ← goodStore.next();
18 |  if good ≠ null then
19 |  |  send GOOD(good, x_i) to parent;
20 |  |  replied ← true;
21 |  else if All feasible goods have been reported then
22 |  |  send a DONE(x_i) message to parent;
23 |  else
24 |  |  send an ASK to the children;
25 |  |  replied ← false;
```

A new assignment is reported using a UTIL message, containing a variable assignment and the corresponding utility the subtree rooted at the sending variable can obtain for that assignment. We call this a *good*. The goodStore is a data structure that stores the received goods, and is able to provide new goods in a best first order. In order to do this, it assumes that the variables' children report goods in a best first order. It can then compute a lower bound on the unreported assignments of each child, which can be used to calculate the lower bound on unseen, and partially seen assignments. Whenever a child is finished, i.e. it submitted all assignments, the goodStore is notified of this (line 13).

2.6.5 Asynchronous Forward Bounding (AFB)

AFB (Gershman et al., 2006a) is a adaptation of AFC to the DCOP setting. The search part is similar to the search of SynchBB. A partial assignment (CPA) is forwarded from

agent to agent, and only one agent at a time is in possession of the CPA. The forward checking, however, is performed in parallel. Using a lexicographic ordering on the variables, AFB is shown to outperform both SynchBB and ADOPT in terms of the number of concurrent constraint checks (Gershman et al., 2006b) and the number of messages. Olteanu et al. (2011), however, showed that when smarter ordering heuristics are used, AFB is outperformed by both SycnhBB and ADOPT. *ConcFB* (Netzer et al., 2010) is another FB algorithm. In ConcFB, different parts of the search space are investigated concurrently. This is achieved by allowing multiple CPAs to exist in the network at the same time.

2.6.6 No-Commitment Branch and Bound (NCBB)

NCBB is a branch and bound algorithm that allows variables to submit different variable assignments to different children. Unlike SynchBB, NCBB operates on the pseudo tree. It is shown to significantly outperform ADOPT in both the number constraint checks performed and the number of messages sent, although Yeoh et al. (2006) showed that BnB-ADOPT outperforms NCBB. DPOP outperforms NCBB in terms of runtime, but the advantage of NCBB over DPOP is that its space complexity is polynomial (as opposed to exponential for DPOP).

2.7 Incomplete DCOP Methods

This section deals with incomplete DCOP methods, that are thus not guaranteed to find the optimal solution.

Algorithm 4: DSA

```
1 initialization
2 |  d ← a random value from D_i;
3 |  send VALUE(x_i, d) to all neighbors;
4 when received(VALUE(x_j, d))
5 |  add x_j = d to the agent_view;
6 |  if All neighbors have submitted a VALUE message then
7 |  |  d ←select a new value and assign it;
8 |  |  send OK(x_i, d) to all neighbors;
```

2.7.1 Local-Search Algorithms

As in the DCSP paradigm, local-search algorithms for DCOPs face the problem of getting trapped into a local minimum. The Distributed Stochastic Algorithm (*DSA*) (Fitzpatrick and Meertens, 2001), has originally been developed for Max-DisCSPs, and

Algorithm 5: MGM

```
1  initialization
2  |  d ← a random value from D_i;
3  |  send OK(x_i, d) to all neighbors;
4  when received(OK(x_j, d))
5  |  add x_j = d to the agent_view;
6  |  if All neighbors have submitted a OK message then
7  |  |  d̂ ←select a new value;
8  |  |  δ is the improvement x_i can obtain by selecting d̂;
9  |  |  send IMPROVE(δ) to all neighbors;
10 when received(IMPROVE(δ_j))
11 |  store δ_j;
12 |  if All neighbors have submitted an IMPROVE message then
13 |  |  if ∀j ∈ children_i δ_i > δ_j then
14 |  |  |  set own value to d̂;
15 |  |  |  send OK(x_i, d̂) to all neighbors;
```

is a very simple local-search algorithm that introduces stochasticity to escape local minima. Whenever an agent cannot perform a move that decreases its local cost, with probability p it chooses another value for its variable (several variants to this heuristic are discussed in (Fitzpatrick and Meertens, 2001)). Zhang et al. (2005), showed that both DSA and DBA can be modified to work in a DCOP setting. The general algorithm is shown in Algorithm 4. Zivan (2005) introduces a framework for local search algorithms that allows for anytime behavior, and to retain the globally best solution found so far. The framework requires a bottom-up cost propagation phase to collect all cost information, after which a top-down propagation allows agents to recognize their currently best assignment.

The Maximum Gain Message (*MGM*) algorithm (Maheswaran et al., 2004), see Algorithm 5, is a simplification of the DBA algorithm. *MGM2* is a generalization of MGM, allowing for 2-coordinated solutions. An agent is not only allowed to change its own value, but also to propose value changes to another agent's variable.

2.7.2 Max-Sum Algorithm

Where the above algorithms operate on the constraint graph of the DCOP, the Max-Sum algorithm (Teacy et al., 2006) operates on the *factor graph* representation of the problem. A factor graph is a bi-partite graph, containing both variable nodes and function nodes, where function nodes represent constraints. Each variable node is connected with every function node representing a constraint in which it participates, and each function node is thus connected with the variables in the scope of its constraint. The

Max-Sum algorithm is an instance of the family of Generalized Distributed Law algorithms (Aji and McEliece, 2000). It is complete for tree-structured factor graphs, but incomplete for factor graphs containing cycles. The algorithm is initiated by the variable nodes, which send a zero-utility function $q(x)$ over their domain to their neighboring function nodes. The function nodes combine this information with their own constraints into a function $q'(\mathbf{x})$, and send a message to each variable x, containing a function $r(x)$ representing the marginal, minimal cost obtainable for each assignment to x. These variable nodes collect the messages from the factor nodes, and create a new message for each of their neighbors. This process continues until it converges. For tree structured graphs, this process is guaranteed to converge, while for graphs containing cycles no guarantees can be given.

3 Multi-Agent Coordination with Complex, Non-Trivial Local Preferences

Coordination is an important aspect of modern day live. Due to globalization, many companies need to coordinate their actions with each other, or even within the same company. Take for example a logistics company that operates world wide. To ensure efficient pickup and delivery, the schedules of trucks, boats and airplanes from different parts of the world must be coordinated to make sure they fit well together. Another example of the need of coordination can be found in the electricity grid. When a failure occurs, badly coordinated local decisions can cause a blackout of the entire network. Scheduling meetings, both within one company and between several companies can quickly become a challenging task.

A common aspect of all problems mentioned above is their distributed nature. Information on the problem is distributed over agents, and agents often have no, or limited, information on the problems of the other agents. Furthermore, in many coordination problems, agents are not only in need of well coordinated solutions. They have preferences over the different solutions, and want to find a solution that maximizes their preferences. In the logistics problem, for example, the different agents (especially within a single company) want to have an allocation of tasks that minimizes their combined cost of driving. An agent's preferences are captured by its local problem. Such local problems are often difficult to solve. Furthermore, in many settings agents have only limited resources. They thus prefer to iterate over only a small subset of all possible solutions.

This Chapter deals with such coordination problems with preferences, where the agents local problems are non-trivial to solve. We assume that the agents have limited resources, and thus use local solvers that are not necessarily complete. Coordination problems can quite naturally be modeled using the Distributed Constraint Optimization (DCOP) framework. In fact, the main motivation for the field has been to model Cooperative Distributed Problem Solving (CDPS) problems (Durfee et al., 1989), of which many coordination problems are instances. In the DCOP framework decisions

are modeled as variables, and dependencies between decisions are modeled as functions over these variables, assigning a value to each possible variable assignment. How the values of the functions are obtained, however, is left open. This allows agents to use their own preferred method for solving their local problems. As argued by (Lesser, 1998), this is an important property to have, because it allows agents to adjust their local solvers to the environment they currently reside in.

Armstrong and Durfee (1997) and Yokoo and Hirayama (1998) are the first to recognize that agents have complex local problems, by which they mean that agents control multiple variables. However, they stay in the realm of constraint satisfaction, and thus do not deal with preferences. Solotorevsky and Gudes (1996) make a separation between the agents' local problems and the coordination problem, but use a separate agent for solving this coordination problem. Burke and Brown (2006) provide the first comparison of DCOP methods under the assumption of complex local problems. Again, however, the difficulty of solving local problems is not taken into account. Jain et al. (2009) and Taylor et al. (2010) look at situations where the utility values are not known a-priori, and need to be discovered online. They look at situations where complete algorithms are not applicable, and focus only on local search algorithms.

The main contribution of this Chapter is twofold. 1) First, we show how coordination problems with preferences are naturally modeled as DCOPs, creating an explicit distinction between the coordination part of the problem and the local preferences of the individual agents. We give a general model, and introduce two benchmark coordination problems. 2) Secondly, we show that on coordination problems with preferences, O-DPOP, using incremental elicitation, outperforms other complete methods in terms of runtime. Furthermore, even when the agents' local solvers cannot guarantee the best-first order, O-DPOP still finds better solutions than the incomplete local solvers. Finally, O-DPOP is shown to require the least amount of queries to the local problems to find a solution.

The rest of this Chapter is structured as follows. In Section 3.1 coordination problems are defined in more detail, and a DCOP model is presented. Section 3.2 takes two typical coordination problems, and shows how to model them as DCOP problems. Section 3.3 evaluates different DCOP algorithms on three different benchmarks, and Section 3.4 concludes.

3.1 Coordination Problems with Complex Local Preferences

Coordination is an inherent part of any multi-agent problem. Many definitions exist, but in this Chapter we use the following definition (Lesser, 1998)

Definition 5 (Coordination problem)**.** *Coordination of agent activities becomes necessary when there are interdependencies among agent activities.*

Such dependencies can take many different forms. However, in this Chapter we are interested in dependencies that create constraints between the different sub-problems of the different agents. To allow agents to adapt to the environment in which they need to solve their local problems, Lesser (1998) argues for a separation between the agents' local problems and the overall coordination problem. This separation not only allows agents to adapt the way they solve their local problems, but also enables better privacy protection (Léauté, 2011) and it allows agents to have preferences that are difficult to express formally. We thus define a coordination problem with preferences as follows:

Definition 6 (Coordination Problem with Preferences)**.** *A* coordination problem *consists of a set of agents $\mathcal{A}$, each owning one or more* decision variables $\mathcal{X}$. *A set of* dependencies *(Dep) between the decision variables determine the set of feasible outcomes, while each agent has* preferences *(Pref) over the different outcomes. A solution to the coordination problem is an assignment to the variables in $\mathcal{X}$ that maximizes some aggregation function of the individual agent preferences.*

When people solve coordination problems, they generally go through only a limited number of solutions before agreeing on a global solution. Furthermore, they start with proposing solutions that are best for them. Only when their own, most preferred solutions are not attainable do they suggest less preferred solutions. For example, when scheduling a meeting, people propose their most preferred times, and only when not everybody agrees do they start looking for other solutions. In a combinatorial auction, nobody will generate a large amount of alternate price quotes. They will only generate a set of most preferred quotes. It is thus natural for agents to provide preferences in a best-first order. In many problems, the preferences are provided by real people. To make sure not too big a burden is put on them, it is important to pay attention to how a coordination algorithm elicits preferences from them.

The field of *preference elicitation* deals with the problem of when to ask which question to which agent so as to minimize the number of preferences elicited from the user. Preferences exist in many different types of problems and forms. In decision support systems (Chen and Pu, 2004), users report their preferences over different outcomes to a decision maker. In Combinatorial Auctions (Conen and Sandholm, 2001), bidders reveal their preferences in the form of bids, while in Constraint Programming they take the form of soft constraints (Meseguer et al., 2006). However, in all settings a combinatorial explosion of possible outcomes exists, and hence one has to be careful when which preferences are elicited from the user.

To simplify the elicitation procedure, one can make assumptions on the preference structure. For example, one can assume additive independence (Keeney et al., 1979), where the utility of an outcome is the sum of the utilities of the individual attributes of the solution. In many problems, including coordination problems, this assumption does not hold. Alternatively, one can use the conditional additive independence

assumption (Boutilier et al., 1997, 2004), where the utility of certain attributes X are independent of other attributes Y, given an assignment of attributes Z. Such relations can, for example, be captured using a CP-net (Boutilier et al., 1997). In Combinatorial Auctions, restrictions on the preference structure are introduced through the use of bidding languages (Nisan, 2006). A bidding language restricts the types of bids a bidder can make, thereby simplifying the elicitation process. Coordination problems are often loosely coupled, meaning that dependencies occur only between subsets of all the agents. This fits with the conditional additive independence assumption.

Assumptions on the preference structure generally reduce the number of queries that are posed to the agents, but the question of which queries are used to elicit an agent's preferences is as important. The following types of queries can be distinguished:

- rank queries: "What is your second most preferred outcome?";
- order queries: "Is outcome A preferable over outcome B?";
- bound queries: "Is the utility of outcome A at least m?";
- value queries: "How much do you value outcome A?"

Recently, preference elicitation has received attention from the Constraint Programming community (Lamma et al., 1999; Faltings and Macho-Gonzalez, 2005; Gelain et al., 2010). Lamma et al. (1999) were the first to address the problem of elicitation in a constraint satisfaction setting. Their goal, however, was to minimize runtime and not the number of queries posed to the agents. Faltings and Macho-Gonzalez (2005) introduced the concept of Open Constraint Optimization, where the solver is not assumed to have full knowledge of the problem it is solving. Information is requested in a best-first manner, i.e. only rank queries are used. In the case of optimization, they report a two- to fivefold decline in the number of queries made to the local problem. Gelain et al. (2010) looked at Soft CSPs with incomplete preferences. A branch and bound approach to solving these problems is investigated, were who elicits what and at what stage of the algorithm are parameters of the system. They show that, for weighted CSPs, letting the user report preferences in a best-first order minimizes user effort. It thus seems that reporting preferences in a best-first order does not only come naturally to people, it also appears to result in efficient problem solving. Using the best-first order, however, does come with a price. In (Brafman et al., 2010), the complexity of finding the next best solution is investigated for different preference representations. They show that for most problems, this is an NP-complete problem.

Coordination problems can be solved using social conventions (Shoham and Tennenholtz, 1992), learning (Stone and Veloso, 2000) or communication between agents. Although each method comes with interesting problems, the scope of this Thesis is limited to methods using communication. One of the first attempts to solve coordination problems using communication is the Contract Net protocol (Smith, 1988;

Sandholm, 1993). The Contract Net protocol is a negotiation-based approach and part of the family of market-based mechanisms (Wellman, 1996). The inability of the Contract Net protocol to backtrack decisions, however, makes it unable to guarantee the optimal outcome. The Generalized Partial Global Planning (GPGP) protocol (Decker and Lesser, 1995) is another instance of a negotiating protocol that allows agents to coordinate their decisions. It consists of different modules, that deal with different types of coordination needs. The DCOP framework fits well to the types of coordination problems described above. Many algorithms are designed to make use of the conditional independency property, and the methods allow for agents to use their own local solvers. In the next Section, a general model for coordination problems with preferences is introduced.

3.1.1 DCOP Model of Coordination

We introduce a new DCOP model for coordination with preferences, where there is a clear distinction between the coordination problem and the local problems of the agents. We assume that the local problems are complex, both in their structure (agents own multiple variables), and in solving them. As described in Burke and Brown (2006), there are three main ways of dealing with local problems ranging over multiple variables. One can decompose the problem, i.e. run a virtual agent for each variable. One can compile the local problems by combing all variables belonging to a single agent in one variable, or one can create a specialized algorithm. In this thesis, as in most of the DCOP literature, we make use of the decomposition approach. The main reason being that the compilation approach fails to scale to more complex local problems. Furthermore, we are interested in developing a general framework for coordination, in effect excluding the creation of a specialized algorithm for particular instances.

It is important to note that we define the problem as a maximization problem. If $\langle \mathcal{A}, \mathcal{X}, \mathit{Dep}, \mathit{Pref} \rangle$ is a coordination model, we assume the following structure on the set of decision variables. Let j represent some *coordination need*, where such a need can deal with some shared resource (computing time), task (delivering a packet in a logistics problem), or common goal (a meeting), that agents need to coordinate over. Then for every agent $a_i \in \mathcal{A}$, and every coordination need j it is involved in, there exists a variable $x_i^j \in \mathcal{X}$, with D_i^j its domain. A mapping $\mathcal{M}$ from agents to a set of variables it owns can be defined as $\mathcal{M}(a_i) \triangleq \mathcal{X}_i = \{x_i^j | a_i \text{ is involved in } j\}$.

As stated in Definition 6, dependencies determine sets of feasible, or allowed, outcomes. In the DCOP framework, such dependencies can be modeled as constraints. If $\mathfrak{d}_j \in \mathit{Dep}$ represents a coordination need j, then $C_j : D_{j_1}^j \times \ldots \times D_{j_m}^j \mapsto \{0, -\infty\}$ is a function that assigns 0 to feasible assignments, and $-\infty$ to infeasible assignments. More formally,

given $\mathbf{x}_j = (x^j_{j_1}, \ldots, x^j_{j_m})$

$$C_j(\mathbf{x}_j) = \begin{cases} -\infty & \text{if } \mathbf{x}_j \text{ is not allowed according to } \mathfrak{d}_j; \\ 0 & \text{otherwise} \end{cases} \tag{3.1}$$

The preferences of an agent a_i can be represented with a utility function l_i, that ranges over all variables that i owns.

$$l_i : \prod_{x^j_i \in \mathcal{M}(a_i)} D^j_i \mapsto \mathbb{R} \tag{3.2}$$

A coordination problem with preferences can now be modeled as follows

Definition 7 (DCOP Model of Coordination)**.** *Given a coordination problem* $\langle \mathcal{A}, \mathcal{X}, \mathit{Dep}, \mathit{Pref} \rangle$*, its DCOP model is given by* $\langle \mathcal{A}, \mathcal{M}, \mathcal{X}, \mathscr{D}, \mathcal{C} \cup \mathcal{L} \rangle$ *where*

- $\mathcal{M}(a_i) \triangleq \mathcal{X}_i = \{x^j_i | a_i$ *is involved in* $j\}$*;*
- $\mathscr{D} \triangleq \{D^j_i | D^j_i$ *is the domain of* $x^j_i \in \mathcal{X}\}$*;*
- $\mathcal{C} \triangleq \{C_j | C_j$ *represents the dependency* $\mathfrak{d}_j \in \mathit{Dep}\}$*;*
- $\mathcal{L} \triangleq \{l_i | l_i$ *represents the preferences* $\mathfrak{p}_i \in$ *Pref of agent* $a_i \in \mathcal{A}\}$.

The objective is to find a feasible assignment **x** *that* ***maximizes*** *the sum of the preferences*

$$f(\boldsymbol{x}) \triangleq \sum_{j=1}^{|\mathcal{C}|} C_j\left([\boldsymbol{x}]_{C_j}\right) + \sum_{j=1}^{|\mathcal{L}|} l_j\left([\boldsymbol{x}]_{l_j}\right), \tag{3.3}$$

In order to determine its preferences, every agent is in possession of a *local solver*. This solver can take many different forms, but in general it will not be a complete solver, i.e. it will not be able to find the exact preferences for every set of possible decisions that the agent can take. Take as an example a logistics problem, where agents must decide which packets to service. For each packet assignment, the agent has to solve a vehicle routing problem in order to determine its utility. The vehicle routing problem is NP-complete, and no local solver exists that will always be able to give the proper utility, in a reasonable amount of time.

Gelain et al. (2010) showed that eliciting preferences in a best-first order minimizes the user effort in terms of the amount of queries. At the same time, however, obtaining

this best-first order is generally also an NP-complete problem. Any algorithm solving coordination problems must thus be able to cope with incomplete solvers that are not able to guarantee a correct best-first order. A simple way of dealing with this is the following. Whenever a solver, while generating solutions in a best-first order, comes across a solution that has a higher utility than the last utility reported, the utility of the new solution is capped, i.e. the new solution gets the (lower) utility of the previous solution. This introduces an error in the final solution. However, due to the additivity of the global preference function, this error can be shown to be at most the sum of the maximal caps[1].

More formally, the error introduced can be characterized as follows. Let ℓ^i be the true utility function for agent a_i, and $\hat{\ell}^i$ its reported reported utility function. $f = \sum_{a_i \in \mathcal{A}} \ell^i$ is the true global utility function, and $\hat{f} = \sum_{a_i \in \mathcal{A}} \hat{\ell}^i$ the reported global utility function. Furthermore, let $\delta_i = max_{\mathbf{x}'} (\ell^i(\mathbf{x}') - \hat{\ell}^i(\mathbf{x}'))$ be the *optimality gap* for local problem of agent a_i.

Proposition 1. *Let* $\boldsymbol{x}^*$ *be the optimal solution and* $\boldsymbol{x}$ *the solution given the misreported values. Then* $f(\boldsymbol{x}^*) - f(\boldsymbol{x}) \leq \sum_{a_i \in \mathcal{A}} \delta_i$.

Proof. First we show that $f(\mathbf{x}^*) - \hat{f}(\mathbf{x}^*) \leq \sum_{a_i \in \mathcal{A}} \delta_i$.

$$\begin{aligned} f(\mathbf{x}^*) - \hat{f}(\mathbf{x}^*) &= \sum_{a_i \in \mathcal{A}} \ell^i(\mathbf{x}^*) - \hat{\ell}^i(\mathbf{x}^*) \\ &\leq \sum_{a_i \in \mathcal{A}} max_{\mathbf{x}'} (\ell^i(\mathbf{x}') - \hat{\ell}^i(\mathbf{x}')) \\ &= \sum_{a_i \in \mathcal{A}} \delta_i \end{aligned} \tag{3.4}$$

We also have that

$$\hat{f}(\mathbf{x}^*) \leq max_{\mathbf{x}'} \hat{f}(\mathbf{x}') = \hat{f}(\mathbf{x}) \leq f(\mathbf{x}) \tag{3.5}$$

Hence, from (3.4) and (3.5) we can conclude that

[1] Do note, however, that in the worst case the least good solution is generated first. This means that all other assignments will be capped to this worst utility

$$f(\mathbf{x}^*) - f(\mathbf{x}) \leq \sum_{a_i \in \mathcal{A}} \delta_i$$

□

The intuition behind Proposition 1 is the following. Whenever agents report information from their local problems in a best-first order, the mistakes in the order made by one agent do not influence the orders of other agents. Therefore, a mistake made by one agent is not enhanced by the mistakes made by other agents. That is, a utility that is misreported by δ_i, results in an error in the final solution that is at most $\sum_{a_i \in \mathcal{A}} \delta_i$.

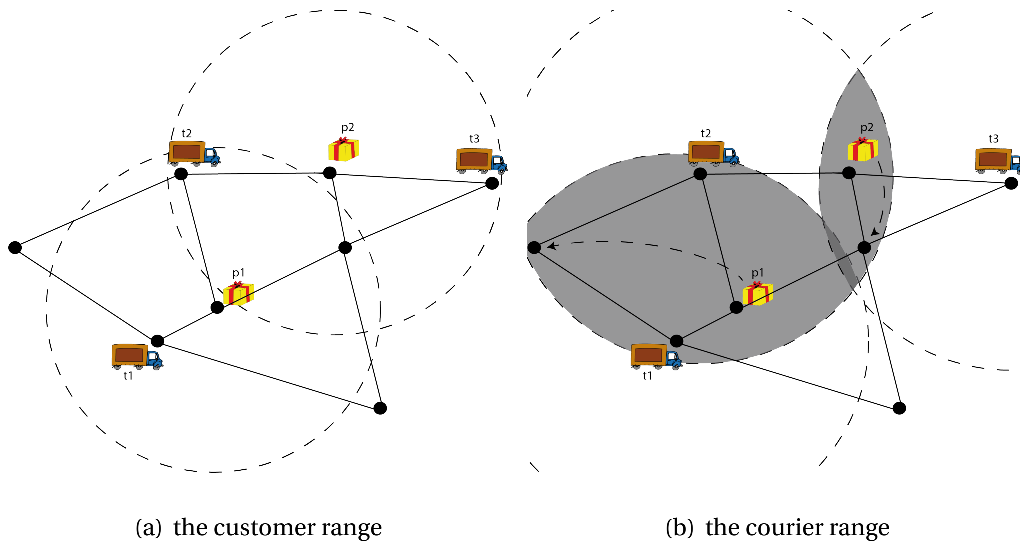
3.2 Coordination Benchmark Problems

In this section we introduce two Benchmarks problems, the Truck Task Coordination (TTC) problem, and an adaption of the standard meeting scheduling problem. Both problems have complex local problems, i.e. agents own multiple decision variables. In addition, solving the local problems of TTC instances involve solving Vehicle Routing Problems (VRP). Solving a VRP is know to be NP-complete. For each benchmark we give a general definition, the DCOP model, and details on the local solver.

3.2.1 The Truck Task Coordination Problem

In the Truck Task Coordination problem, we take the perspective of a single company, consisting of a group of couriers, dispersed over a geographical area. The problem is an adaptation of the logistics problem introduced in (Ottens and Faltings, 2008). Each courier has its own garage, from which it operates. Customers offer packets for pickup and delivery to the company, but there are restrictions on which couriers are allowed to service them: packets will only be offered to couriers whose garages are within a certain range of the pickup and delivery locations. Furthermore, the range of the couriers will also be limited. These two restrictions together make that not all possible packet assignments are feasible. The main differences with standard VRP problems (Toth and Vigo, 2001; Léauté et al., 2010) are that not all couriers are able to service all packets and that the goal is to maximize utility, not to minimize driving distance. Utility is defined as the payment obtained from delivering a set of packets, minus the cost incurred by driving.

Definition 8 (The TTC problem)**.** *The Truck Task Coordination problem consists of a map* $Map = \langle Cities, Roads \rangle$ *with the cities in* $Cities$ *dispersed over the euclidian plane and* $Roads \subseteq Cities \times Cities$*, a set of couriers* $\mathcal{T} = \{t_1, \ldots, t_n\}$ *and a set of packets* $\mathcal{P} = \{p_1, \ldots, p_m\}$*. The couriers and packets are dispersed over the different cities. Each packet can only be offered to couriers whose garages are within customer range of the*



(a) the customer range (b) the courier range

Figure 3.1: Example of a TTC problem

packet's home cities. Couriers are only allowed to accept packets if both the pickup and delivery city are within courier range distance from the courier. The customer range is the same for all packets, and the courier range is the same for all couriers. Each packet has a weight, and couriers have a maximal capacity. For each weight unit delivered, a truck receives a certain reward. The goal is to find an assignment that maximizes the total utility of the system, while at the same time satisfying both the range and capacity constraints.

An example of a problem instance can be found in Figure 3.1. Figure 3.1(a) shows the couriers that lie in the range of the packets. In this particular example both packets can be offered to only two out of three couriers. Figure 3.1(b), shows the courier range of all couriers. In this particular example, all couriers can service all packets that they are offered.

We assume that all couriers have private utility information, but are still cooperative, i.e. they are willing to participate in the algorithm and will always tell the truth. However, their utilities are assumed not to be known a priori. That is, the solver must query the couriers for the utility of every combination of packet assignments individually. The idea is that individual preferences depend not only on the profit obtained from delivering packets, but also on the preferences of the individual couriers: they might want to stop by a shop or want to have lunch in a certain restaurant. Such preferences make the local problem of each courier hard to formalize; hence the assumption on querying the user for all needed preference information. Note that what preference information is needed highly depends on the solver at hand.

DCOP Model

The TTC problem can be modeled as a DCOP problem, where every courier is an agent. The decisions couriers have to coordinate over, are on who picks up which packet. Following the DCOP model given in Definition 7, each packet $p_j \in \mathcal{P}$ is a coordination need. Let $\mathcal{P}_i \subseteq \mathcal{P}$ be the set of packets that has been offered to courier t_i, and $\mathcal{T}_j$ be the set of trucks that are offered packet p_j. Then for each courier t_i and every packet p_j such that $p_j \in \mathcal{P}_i$ and $|\mathcal{T}_j| > 1$, t_i owns a binary variable x_i^j with domain $D_i^j = \{0, 1\}$. If $x_i^j = 1$, then t_i will service packet p_j, and if $x_i^j = 0$ it will not service it. In the example given in Figure 3.1, there are three couriers t_1, t_2 and t_3. Packet p_1 is the packet in the bottom left, and is offered to t_1 and t_2 ($\mathcal{T}_1 = \{t_1, t_2\}$). Packet p_2 is situated on the upper right, and is offered to t_2 and t_3 ($\mathcal{T}_2 = \{t_2, t_3\}$). We have thus that, $\mathcal{P}_1 = \{p_1\}$, $\mathcal{P}_2 = \{p_1, p_2\}$ and $\mathcal{P}_3 = \{p_3\}$. This gives us the variables x_1^1, x_2^1, x_2^2 and x_3^2.

Coordination is needed to ensure that no two trucks decide to deliver the same packet. For each packet p_j, this can be modeled using a set of binary constraints between couriers. Given two couriers $t_i, t_{i'} \in \mathcal{T}_j$

$$C_{i,i'}^j(x_i^j, x_{i'}^j) = \begin{cases} -\infty & \text{if } x_i^j + x_{i'}^j > 1; \\ 0 & \text{otherwise} \end{cases} \tag{3.6}$$

The local utility of an agent depends on the set of packets it is assigned, and on the route that it will take. The cost of the route is provided by the local solver, which is detailed below. A packet assignment is infeasible if it either contains a packet whose pickup or delivery city is outside of the courier range, or when the capacity constraint cannot be met. This is indicated by setting the cost to ∞. Only packets that are within a certain distance are to be serviced. Let $distance(t_i, p_i) = true$ when both the pickup and delivery cities of packet p_i are within courier range, and $false$ otherwise. Then $ownPackets_i = \{p_i | distance(t_i, p_i) = true \wedge |\mathcal{T}_j| = 1 \wedge w_{p_i} \leq \text{capacity}\}$ is the set of packets that only courier t_i can service, and $coordinationPackets_i = \mathcal{P}_i \setminus ownPackets_i$ is the set of packets courier t_i needs to coordinate over. For every delivered packet p_i, a courier gets paid αw_{p_i}, where w_{p_i} is the weight of packet p_i and α is the payment per unit weight, hence it is guaranteed a payment of $pay = \sum_{p \in ownPackets_i} \alpha w_p$.

$$l_i(ownPackets_i, x_i^{j_1}, \ldots, x_i^{j_m}) \;=\; pay + \sum_{k=1}^{m} x_i^{j_k} \alpha w_{p_{j_k}} - \text{cost of the route} \tag{3.7}$$

where $m = |coordinationPackets_i|$. A DCOP model for the TTC problem is defined as follows

Definition 9 (DCOP Model for the TTC problem)**.** *A TTC problem* $(Map, \mathcal{T}, \mathcal{P})$ *can be modeled as* $\langle \mathcal{T}, \mathcal{X}, \mathcal{M}, \mathcal{D}, \mathcal{C} \cup \mathcal{L} \rangle$*, with*

- $\mathcal{X} \triangleq \{x_i^j | t_i \in \mathcal{T} \text{ and } p_j \in \mathcal{P}_i\}$;
- $\mathcal{M} \triangleq \mathcal{M}(t_i) = \{x_i^j | x_i^j \in \mathcal{X}\}$;
- $\mathcal{D} \triangleq \{D_i^j | x_i^j \in \mathcal{X}\}$ *with* $D_i^j = \{0, 1\}$;
- $\mathcal{C} \triangleq \{C_{i,i'}^j | p_j \in \mathcal{P} \wedge t_i, t_{i'} \in \mathcal{T}_j\}$;
- $\mathcal{L} \triangleq \{l_i | t_i \in \mathcal{T}\}$.

Problem Generation

Problems are randomly generated according to the following paradigm. First, a grid of predefined size is created. On the grid cities are randomly dispersed. Then, a network of roads between cities is grown by first selecting one city in the map, and then iteratively adding the city closest to the graph. Next, a predefined number of couriers is each assigned a unique starting city. Finally, the packets are randomly dispersed over the cities, with randomly chosen destination cities.

Local Solver

Every time the optimization algorithm needs the utility of a certain variable assignment, the local solver is queried. The local solver used for the TTC problem uses a local-search algorithm with random restarts to determine the proper utility for a certain packet assignment (Ottens and Faltings, 2008). A fair comparison between algorithms is ensured by making the seed for the random generator used dependent on the set of packets carried.

Generating the best-first order for this problem amounts to solving a VRP problem for all possible packet assignments. Since this is not desirable, some assumptions are made that make it easier to generate an order. The correctness of the order, however, cannot be guaranteed. It is assumed that every packet delivery is profitable, and hence that the most preferred option is to deliver all packets. Secondly, given the previously released solution, new solutions are generated by removing a single packet from the list of accepted packets. Utilities for all possible new solutions are calculated and put in an ordered list. If a newly created solution is found to have a higher utility than the last reported utility, the utility for the new solution is capped at this utility value.

3.2.2 The Meeting Scheduling Problem

Meeting scheduling problems (Franzia et al., 2002; Maheswaran et al., 2003) have been much used in the evaluation of DCOP methods (Maheswaran et al., 2003; Petcu and Faltings, 2005a; Yeoh et al., 2010). In a meeting scheduling problem, a group of people needs to schedule meetings amongst themselves in such a way that no agent has two overlapping meetings, while maximizing the sum of the preferences of the different agents. The meeting scheduling problem has been intensively studied in the literature. In (Abdennadher and Schlenker, 1999; Bakker et al., 1993), the problem is modeled as a CSP, and solved using centralized methods. Sen (1997) uses the Contract Net protocol to find a schedule that satisfies all availability constraints of the participating agents. However, commitment to a certain schedule might result in some meetings not being scheduled even though in principle they could have been scheduled. Garrido and Sycara (1995) introduce a different negotiation approach that does not only take the availability constraints of different agents into account, but also tries to find a meeting time that best satisfies the agents' preferences. However, their system is only able to deal with one meeting at a time and is not able to guarantee a schedule when multiple meetings are to be scheduled. Franzia et al. (2002) also use negotiation to find a meeting time, but again only focus on single meetings with multiple agents. Hassine et al. (2004, 2005) are able to deal with multiple meetings, using the Distributed Reinforcement of Arc Consistency (DRAC (Hassine and Ghédira, 2002)) approach. In their approach, each meeting is instigated by a single agent, but only this agent's preferences are maximized when finding a meeting time, thus ignoring the preferences of the other participants. Their global objective is thus different from ours.

In the DCOP literature, all models used for DCOP evaluations of meeting scheduling problems have either ignored the preferences of individual agents, or deemed them trivial to obtain. As in the TTC problem defined above, in general this is not the case. People's preferences can depend on where they live, i.e. how far they need to travel, whether they have private engagements or simply a preference on the morning or the afternoon. All models found in the DCOP literature model the preferences as preferences over single meetings, where there is no dependency between the meetings. In this Chapter we extend this to allow agents to have preferences over combinations of meetings.

DCOP Model

Let $\mathcal{A} = \{a_1, \ldots, a_n\}$ be a set of agents, and $M = \{m_1, \ldots, m_k\}$ the set of meetings. Let $A_j = \{a_{i_1}, \ldots, a_{i_{n(j)}}\}$ be the set of agents that participate in the meeting m_j, and $M_i = \{m_j | a_i \in m_j\}$ be the meetings that agent a_i participates in. Furthermore, there are n_T different time slots available for each meeting. The agents need to agree on a meeting time for each meeting. Hence each agent a_i has a variable x_i^j for each meeting

m_j it participates in. The need for agreement between agents can be modeled using an equality constraint.

$$C_{i,j}^k(x_i^k, x_j^k) = \begin{cases} 0 & \text{if } x_i^k = x_j^k; \\ -\infty & \text{otherwise} \end{cases} \tag{3.8}$$

Each agent has a preference over different meeting schedules. When dealing with real people, preferences for different meeting times depend on which meeting is held when. We want to capture that dependency while not making the model too involved. Each agent a_i has a most preferred time, T_i, at which it wants to schedule its meetings. Its goal will be to minimize the total distance between its most preferred time and the actual meeting times. We can thus define its preference function as

$$l_i(\mathbf{x}_i) = \begin{cases} -\infty & \text{if } \exists x_i^k, x_i^{k'} \text{ s.t. } k \neq k' \wedge x_i^k = x_i^{k'} \\ \alpha \sum_{k=1}^{|A_i|} [d_i - |T_i - x_i^k|] & \text{otherwise} \end{cases} \tag{3.9}$$

where $d_i = \max(T_i, T - T_i)$ is the maximal distance to the most preferred time, and $\mathbf{x}_i = (x_i^{j_{i_1}}, \ldots, x_i^{j_{|A_i|}})$ a meeting schedule. Note that an agent will not accept any schedule in which two different meetings it participates in have the same slot.

A DCOP model for the meeting scheduling problem is defined as follows

Definition 10 (DCOP Model for the Meeting Scheduling Problem)**.** *A meeting scheduling problem $\langle \mathcal{A}, M \rangle$ can be modeled as $\langle \mathcal{A}, \mathcal{X}, \mathcal{M}, \mathcal{D}, \mathcal{C} \cup \mathcal{L} \rangle$, with*

- $\mathcal{X} \triangleq \{x_i^j | a_i \in \mathcal{A}$ *and* $m_j \in M_i\}$;
- $\mathcal{M} \triangleq \mathcal{M}(a_i) = \{x_i^j | x_i^j \in \mathcal{X}\}$;
- $\mathcal{D} \triangleq \{D_i^j | x_i^j \in \mathcal{X}\}$ *with* $D_i^j = \{1, \ldots, T\}$;
- $\mathcal{C} \triangleq \{C_{i,j}^k | m_k \in M \wedge a_i, a_j \in A_k\}$
- $\mathcal{L} \triangleq \{l_i | a_i \in \mathcal{A}\}$

Problem Generation

The problem generator as used in FRODO (Léauté et al., 2009) takes as input a number of agents, a number of meetings, the number of agents per meeting and the number of slots n_T. The meetings are then randomly generated, as also the preferred time of each agent.

Local Solver

The local solver used in our experiments is able to generate a best-first order on the possible schedules for a single agent without generating all possible schedules before hand. Given a schedule, one can count the number of meetings with a certain distance. For example, given a schedule $m_1 = 4, m_2 = 3, m_3 = 7$, with in total 7 available slots and 3 the most preferred meeting time, its count is $[1, 1, 0, 0, 1]$, i.e. one meeting with distance 0 (m_2), one meeting with distance 1 (m_1) and one meeting with distance 4 (m_3). Such a count can be seen as a schedule template. All schedules satisfying the same template have the same preference value. Given such a template, a new template with a lower preference value can be generated by moving one meeting to a slot with a bigger distance. Given the template $[1, 1, 0, 0, 1]$, a possible successor is $[0, 2, 0, 0, 1]$. One can define a lexicographic ordering over possible schedule templates that ranges from left to right, i.e. the more meetings close to the preferred time the better. Each template $\mathbf{t}$ has a value $val(\mathbf{t})$, that indicates how far away the slots are from the preferred slot. The value of a template $\mathbf{t}$ is defined as $val(\mathbf{t}) = \sum_{i=1}^{|\mathbf{t}|} i \times t_i$. The form of a template is restricted. For example, there can be only one slot with distance 0, i.e. the most preferred slot. Then, depending on the position of the most preferred slot, all the other entries in the template have an upper bound of either 1 or 2. For example, if there are 7 possible slots, and the most preferred time is 3, then the upper bounds on the values of the entries is given by $\mathbf{m} = [1, 2, 2, 1, 1]$. A successor $succ(\mathbf{t})$ of $\mathbf{t}$ can be defined as any template $\mathbf{t}'$ such that $val(\mathbf{t}') = 1 + val(\mathbf{t})$, $\forall i \in (1, |\mathbf{x}|)\ t_i \leq m_i$ and there is only one i such that $t'_i = t_i - 1$ and $t'_{i+1} = t_{i+1} + 1$.

Definition 11 (Template Generation)**.** *Starting from the template with the lowest value, a sequence of templates $\boldsymbol{t}_1, \ldots, \boldsymbol{t}_k$ can be generated as follows. Given a template $\boldsymbol{t}$, generate all templates such that $\boldsymbol{t}' = succ(\boldsymbol{t})$ and $\forall \boldsymbol{t}''$ s.t. $\boldsymbol{t}' = succ(\boldsymbol{t}'')$ it is the case that $\boldsymbol{t} \succeq \boldsymbol{t}''$ and add them to the sequence.*

Proposition 2. *Let $\boldsymbol{t}_1, \ldots, \boldsymbol{t}_m$ be a sequence of templates generated as described in Definition 11. Then the sequence satisfies $\forall i \neq j :\ \boldsymbol{t}_i \neq \boldsymbol{t}_j$ and $\forall i, j :\ i > j \rightarrow val(\boldsymbol{t}_i) \geq val(\boldsymbol{t}_j)$.*

Proof. First, we prove the claim that each template occurs only once in the sequence. Towards a contradiction, let $\mathbf{t}_i$, $\mathbf{t}_j$ be the first templates such that $\mathbf{t}_i = \mathbf{t}_j$, while $i \neq j$. Let $\mathbf{t}'_i$ and $\mathbf{t}'_j$ be such that $\mathbf{t}_i$ ($\mathbf{t}_j$) is generated by $\mathbf{t}'_i$ ($\mathbf{t}'_j$). Then by definition $\mathbf{t}'_i \neq \mathbf{t}'_j$, $\forall \mathbf{t}'' \in \mathcal{X} :\ \mathbf{t}_i = succ(\mathbf{t}'') \rightarrow \mathbf{t}'_i \succeq \mathbf{t}''$ and $\forall \mathbf{t}'' \in \mathcal{X} :\ \mathbf{t}_j = succ(\mathbf{t}'') \rightarrow \mathbf{t}'_j \succeq \mathbf{t}''$. From this it follows that $\mathbf{t}'_i = \mathbf{t}'_j$, hence $\mathbf{t}_i \neq \mathbf{t}_j$.

To prove that the templates are generated in the correct order, assume, towards a contradiction, that $\mathbf{t}_i$ and $\mathbf{t}_j$, with $i > j$, are the first templates such that $val(\mathbf{t}_i) < val(\mathbf{t}_j)$. Let $\mathbf{t}_i$ ($\mathbf{t}_j$) be generated by $\mathbf{t}'_i$ ($\mathbf{t}'_j$). Since $i > j$, it must by assumption be that $val(\mathbf{t}'_i) \geq val(\mathbf{t}'_j)$. Since $val(\mathbf{t}_i) = val(\mathbf{t}'_i) + 1$ and $val(\mathbf{t}_j) = val(\mathbf{t}'_j) + 1$, we have that $val(\mathbf{t}_i) \geq val(\mathbf{t}_j)$. Hence the templates are generated in the correct order. □

For each template, the schedules that satisfy the template can be generated in any order, while still generating schedules in a best-first order given the preference function (3.9). Furthermore, generating schedules in a best-first order can be done cheaply

Proposition 3. *Generating a sequence $\boldsymbol{t}_1, \ldots, \boldsymbol{t}_m$ of templates as done in Proposition 2 can be done in $\mathcal{O}(md)$, where $d = |\boldsymbol{m}|$.*

Proof. Given a template **t**, generating its successors is done by iterating over all positions in the template, shifting a used slot one place to the right (if possible), and then checking whether no smaller template could have generated this successor. This takes $2d$ steps at most. This must be done for each template in the sequence, arriving at complexity of $\mathcal{O}(md)$ □

3.2.3 Contract Net Protocol

The Contract Net protocol (Smith, 1988) has been used extensively to solve coordination problems. We use the Contract Net protocol to solve the TTC problem as introduced in Section 3.2.1.The Contract Net protocol is a coordination protocol for distributed task allocation. Simply put, agents can propose tasks, and other agents can bid for performing such tasks. In our experiments, we modeled the problem using two types of agents. Every task (or customer) and courier is represented by a single agent. A customer offers its task to all couriers within customer range. Upon receiving an offer, a courier calculates its marginal utility for accepting the task using the local solver introduced in Section 3.2.1, and reports the marginal utility as its bid. The customer then gives its task to the courier with the highest bid.

3.3 Experimental Evaluation

The main goal of the experiments presented in this section is to investigate the influence of non-trivial local sub-problems on the performance of different DCOP algorithms. We use three different benchmarks: the TTC problem and the meeting scheduling problem introduced in Section 3.2, and the graph coloring problem.

The TTC problem involves complex local problems that are non-trivial to solve. The meeting scheduling problem deals with complex local problems that are easy to compute, while the graph coloring problem is a standard algorithm used in DCOP evaluations. To simulate the non-triviality of the local problems for meeting scheduling, we run the same algorithms with no penalty and a 30 second penalty per query to the local solvers. Performance is measured using both simulated time (Sultanik et al., 2007) and Non Concurrent Constraint Checks (NCCC) (Gershman et al., 2006b) as also the amount of information exchanged. Because our experiments run on a single machine, looking at the total runtime of the algorithm does not provide a good picture of the

actual distributed performance of the algorithms. This is due to the fact that operations that can be performed in parallel in a distributed run, cannot be performed in parallel when run on a single machine. Hence the total run time on a single machine will be an overestimation of the distributed runtime. For the NCCC count, only queries to the local solver are counted. Queries of the coordination constraints are assumed to be trivial and are thus ignored. Furthermore, due to the non-optimality of the local solver, the assumptions made for the local solver used in O-DPOP and the non-completeness of the local-search approaches, the solution quality should be compared as well. DPOP, O-DPOP, SynchBB, AFB, ADOPT, DSA, MGM and MGM2 are evaluated on different types of problems. In addition, for the TTC problem we also evaluate the Contract Net approach[2].

3.3.1 Experimental Setup

For the TTC experiments we used a map size of 1000 x 1000, 100 cities and 10 couriers. The number of packets was set to 40. Each courier has a capacity of between 4 and 10 weight units, while each packet weighs between 1 and 4 units. Both the customer range and the courier range have been taken from $\{300, 310, 320, 330, 340, 350, 360, 370\}$. Lower ranges resulted in trivial problems while higher ranges make the problems too complex. The reward for delivering a package is either 200 or 500. For each combination of parameters, 99 instances have been created.

For the meeting scheduling problem, we created problems with 30 agents, between 10 and 16 meetings and 3 agents per meeting. The scaling factor α was set to 100. For each parameter value, 99 instances have been generated. In order to evaluate the influence of having a non-trivial local problem, we run the algorithms both with a 30 second penalty per constraint check of the preference constraints, and without a penalty.

For the graph coloring problem, we used a 3-coloring problem with the number of nodes taken from $\{10, 12, 14, 16, 18, 20, 22, 24\}$, and a density of 40%(i.e. the number of edges is 40% of the maximal number of edges). Again, 99 instances have been generated for every number of nodes.

The local-search algorithms do not have a termination condition, but are given a number of rounds. However, to have a fair comparison we gave them as much time as O-DPOP. Furthermore, local-search algorithms operate poorly in the presence of hard constraints. Hence, for the local-search algorithms, every hard constraint that is violated results in a penalty instead of $-\infty$ utility.

For every evaluation metric used, we plot the median value together with 95% intervals. All algorithms have been written in Java, and have been implemented in the

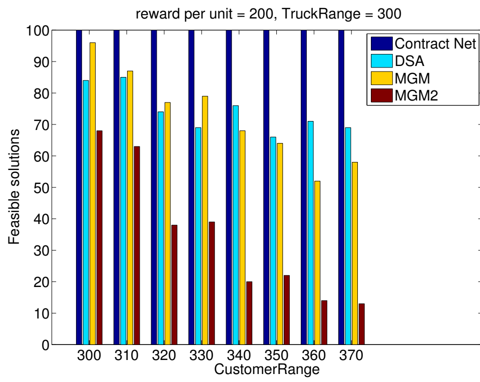
[2]The Contract Net approach does not fit well in either the meeting scheduling and graph coloring domains

FRODO (Léauté et al., 2009) framework. Contrary to the implementation used in (Sultanik et al., 2007), the FRODO implementation uses a single queue, where all messages are ordered using timestamps. Messages are released one by one (as opposed to batch), and the next message is released only after the previous message has been processed by the relevant agent. All experiments have been run on a 64 core Intel xeon 3 Ghz machine running linux, and each run was allocated 2 Gb of internal memory. Each run was given a time-out of 30 minutes.

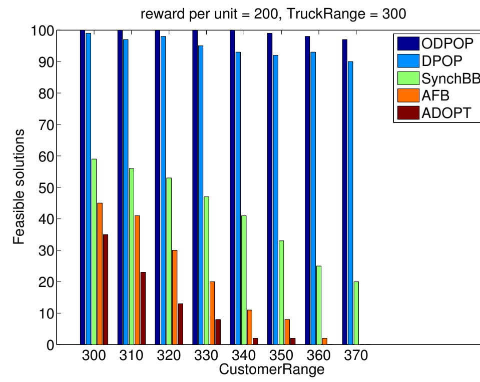
Not all algorithms have been able to find a solution for all problems. Therefore, Figures 3.2, 3.3 and 3.4 show, for each of the algorithms and each type of problem, the number of feasible solutions that have been found. From the figures, you can see that SynchBB, AFB and ADOPT solve only very few instances on the TTC and meeting scheduling problems, and hence they are omitted from the rest of the analysis of these problems. On the graph coloring problems their performance is better, and they are thus still included. Also note that the Contract Net is able to find a solution for all TTC instances. This is because the Contract Net requires only a single query to the local problem per packet, and thus never runs into the time out. However, as we will see later, in terms of the solution quality, the Contract Net does not perform well for all problems.

3.3.2 Solution Quality

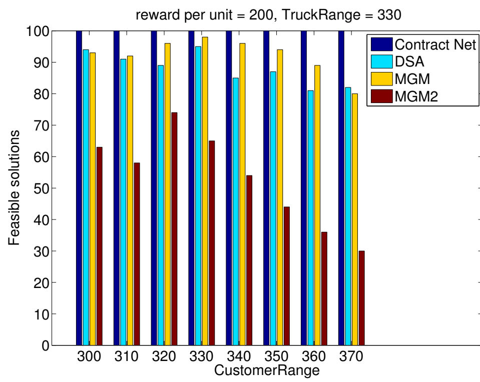
Figure 3.5 shows the difference in solution quality between the optimal solution and the found solution for the TTC problem, where we use the solution found by the complete algorithm DPOP as a reference point. We show the results for a courier range of 300 and 330. It turns out that when the unit reward is set to 500, even with capping utility values, O-DPOP is able to find the optimal solution. MGM, MGM2 and DSA perform worse. Interestingly, MGM is consistently better than DSA, while MGM2 is only able to find solutions for the smaller problems. For the problem instances with a reward of 500, the Contract Net finds solutions that are of the same quality as the local search algorithms, while its performance for instances with a reward of 200 is significantly worse. The main cause of this is the value for the reward. If this value is higher than the costs that an agent incurs by delivering a package, this reward is the dominating factor in the final quality of the solution. Figure 3.5(c) and 3.5(d) show the solution quality for a unit reward of 200. One can clearly see that in this case the Contract Net algorithm performs worse than the local-search algorithms. Interestingly, O-DPOP is not able to find the optimal solution when the reward is 200. Remember that the local solver generates assignments under the assumption that more packets is always better. However, with a action radius of 300/330, resulting in a maximal cost of 600/660, a reward of 200 makes that some packages are too expensive to deliver. Hence, the "the more the better" assumption does not hold anymore and the local solver used by O-DPOP is not able to generate a correct best-first order.



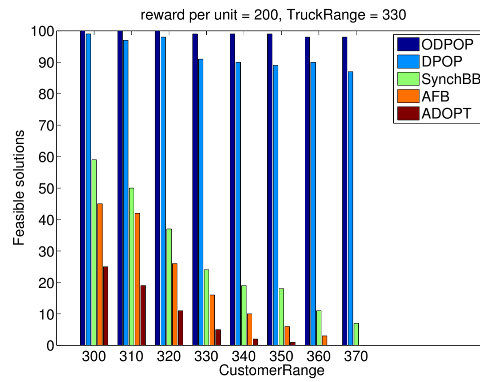
(a) TTC problem



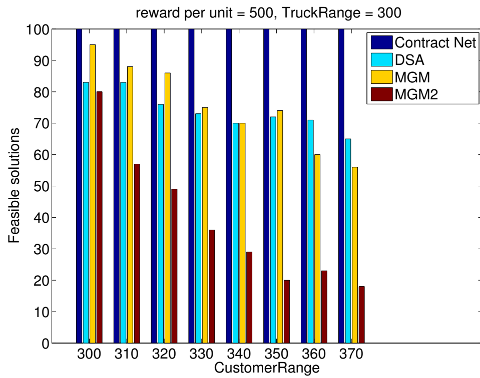
(b) TTC problem



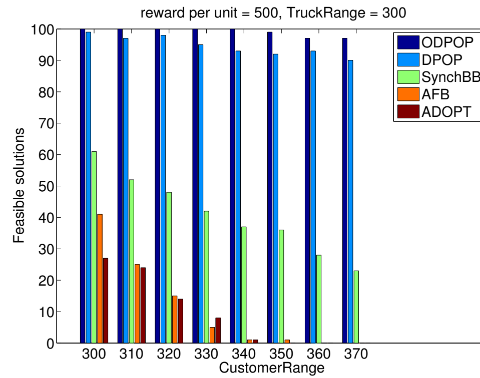
(c) TTC problem



(d) TTC problem

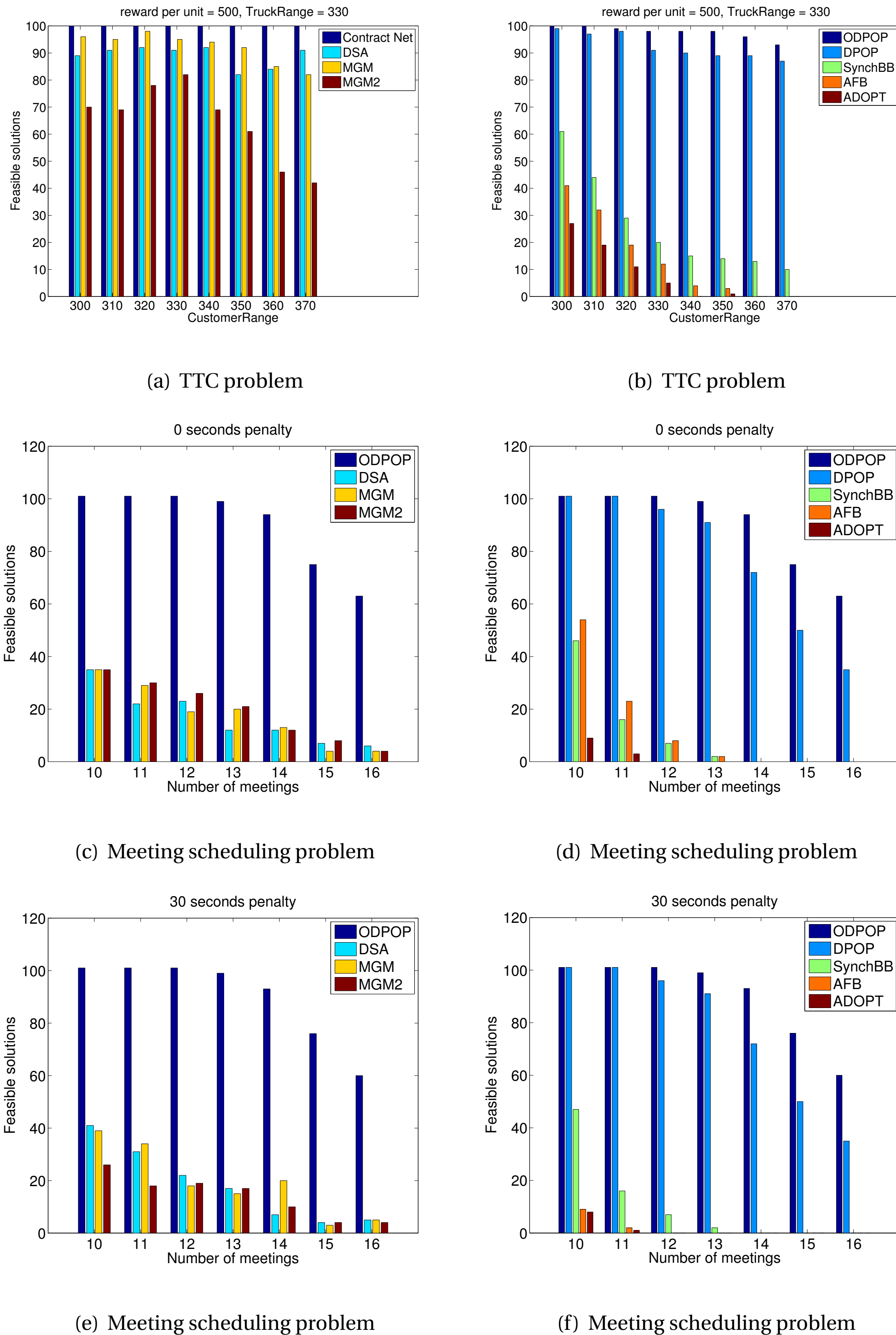


(e) TTC problem



(f) TTC problem

Figure 3.2: Percentage of solutions found (1).



(a) TTC problem

(b) TTC problem

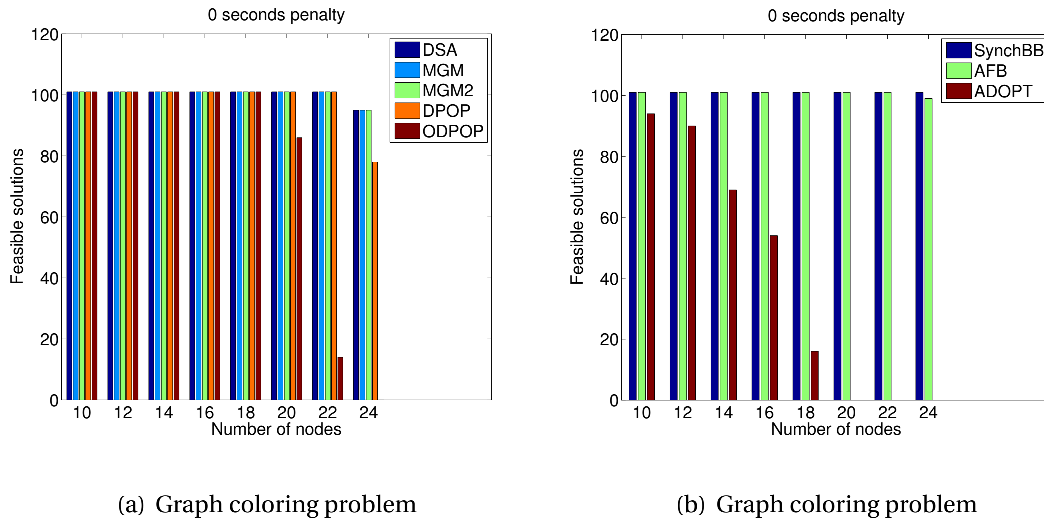
(c) Meeting scheduling problem

(d) Meeting scheduling problem

(e) Meeting scheduling problem

(f) Meeting scheduling problem

Figure 3.3: Percentage of solutions found (2)



(a) Graph coloring problem

(b) Graph coloring problem

Figure 3.4: Percentage of solutions found (3)

The local-search algorithms do not perform well on the meeting scheduling problems. From Figures 3.3(c), 3.3(d), 3.3(e) and 3.3(f), one can see that the local-search algorithms are not able to find a feasible solution to most of the problems, when given the same amount of time as O-DPOP. Furthermore, it holds for all algorithms that the more meetings there are, the fewer problems they are able to solve given either the available memory or the available time. Do note that this drop is more severe for DPOP than for O-DPOP. Since the local-search algorithms cannot find a solution for most problems, Figures 3.6(a) and 3.6(b) show, for the local problems, the median solution quality over the solved problems. Their median solution quality is significantly lower than the optimal utility found by O-DPOP and DPOP. The slight drop for DPOP is caused by the inability for DPOP to solve most of the larger problems. Note that it was possible for O-DPOP to generate the accurate best-first order, and hence it always finds the optimal solution.

When looking at the results for the graph coloring problem (Figure 3.7), the advantage of MGM2 over MGM and DSA becomes clear. While DSA and MGM consistently find a sub-optimal solution, MGM2 is able to find the optimal solution most of the time.

3.3.3 Runtime

Figure 3.8 shows the simulated runtime results in ms for the TTC problem. It is clear that the Contract Net algorithm is very efficient in terms of runtime. O-DPOP runs consistently faster than DPOP. We do not show the runtime for the local-search algorithms because they were allocated the same amount of time as O-DPOP.

While the customer range influences the number and size of the constraints of the

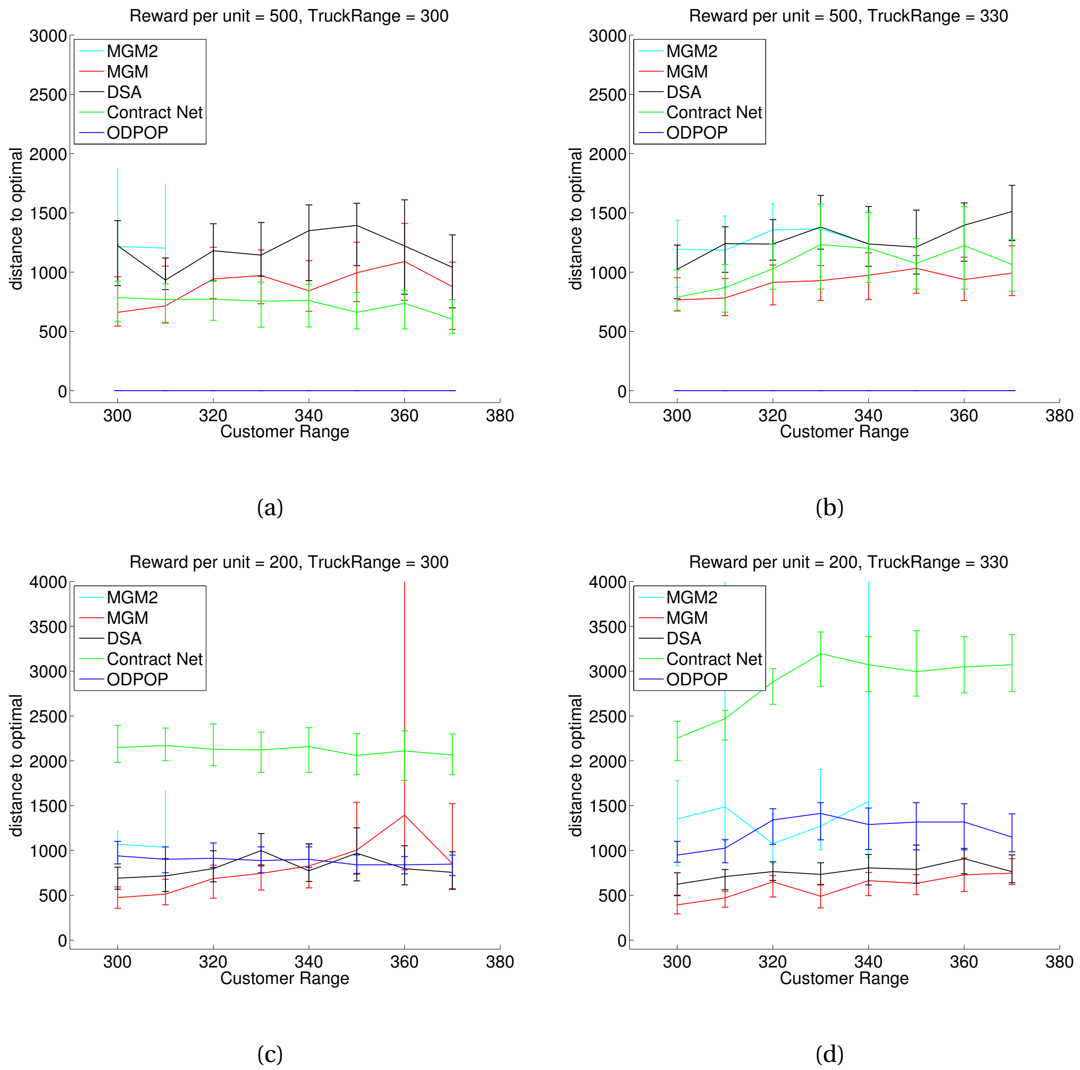
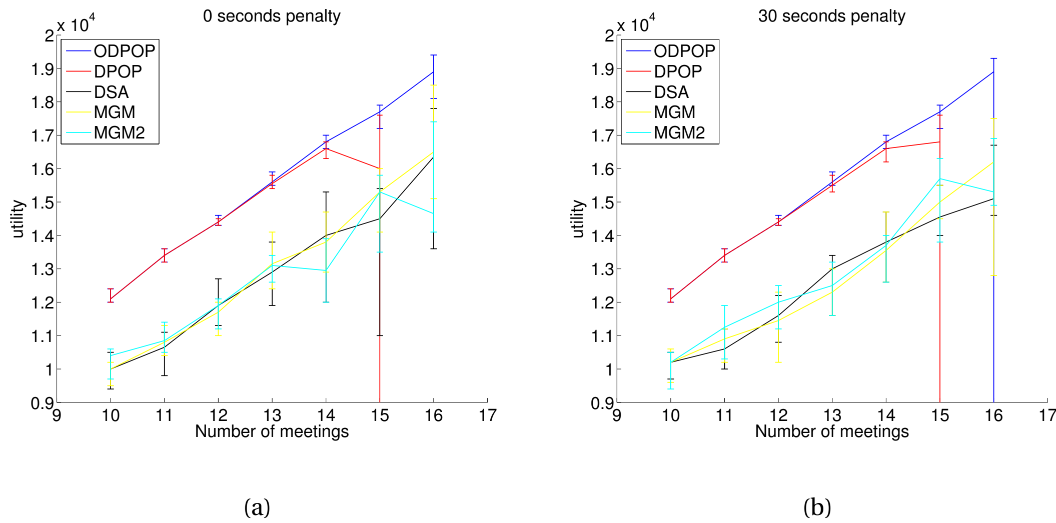


Figure 3.5: Distance to the optimal solution for the TTC problem

problem, the courier range influences the amount of coordination needed. The higher the courier range, the more coordination is needed. This is illustrated by the runtimes incurred by O-DPOP. The median runtime for a courier range of 330 is higher than the median runtime for a courier range of 300. Interestingly, the runtime for DPOP is also higher with a higher courier range, despite the fact that there is no change in the constraint graph or the size of the UTIL messages. A higher courier range means that a courier is able to accept more packages, which in turn means that a bigger part of its local problem is feasible. Therefore, even though nothing changes in the constraint structure of the problem, DPOP needs more time when it is faced with a higher courier range. It simply needs to query a bigger part of its local problem, and thus solve more non-trivial local problems.



(a) (b)

Figure 3.6: Solution quality for the meeting scheduling problem

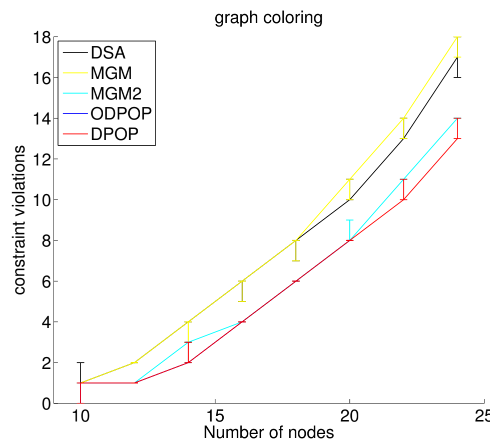


Figure 3.7: Solution quality for the graph coloring problem

Figure 3.9 shows the simulated runtime results for the meeting scheduling problem. Already with no penalty for querying the local problem, O-DPOP scales better than DPOP, and this effect is only enlarged when the penalty increases. Figure 3.9(c) shows that when there is no complex local problem, the DPOP algorithm significantly outperforms O-DPOP. In the graph coloring problem, an assignment either satisfies a constraint or not. There are thus no preferences, and O-DPOP is not able to take advantage of the best-first order. SynchBB and AFB perform similar to DPOP, although DPOP is a bit faster. ADOPT is only able to solve instances up to 16 nodes, and thus does not scale to bigger problems.

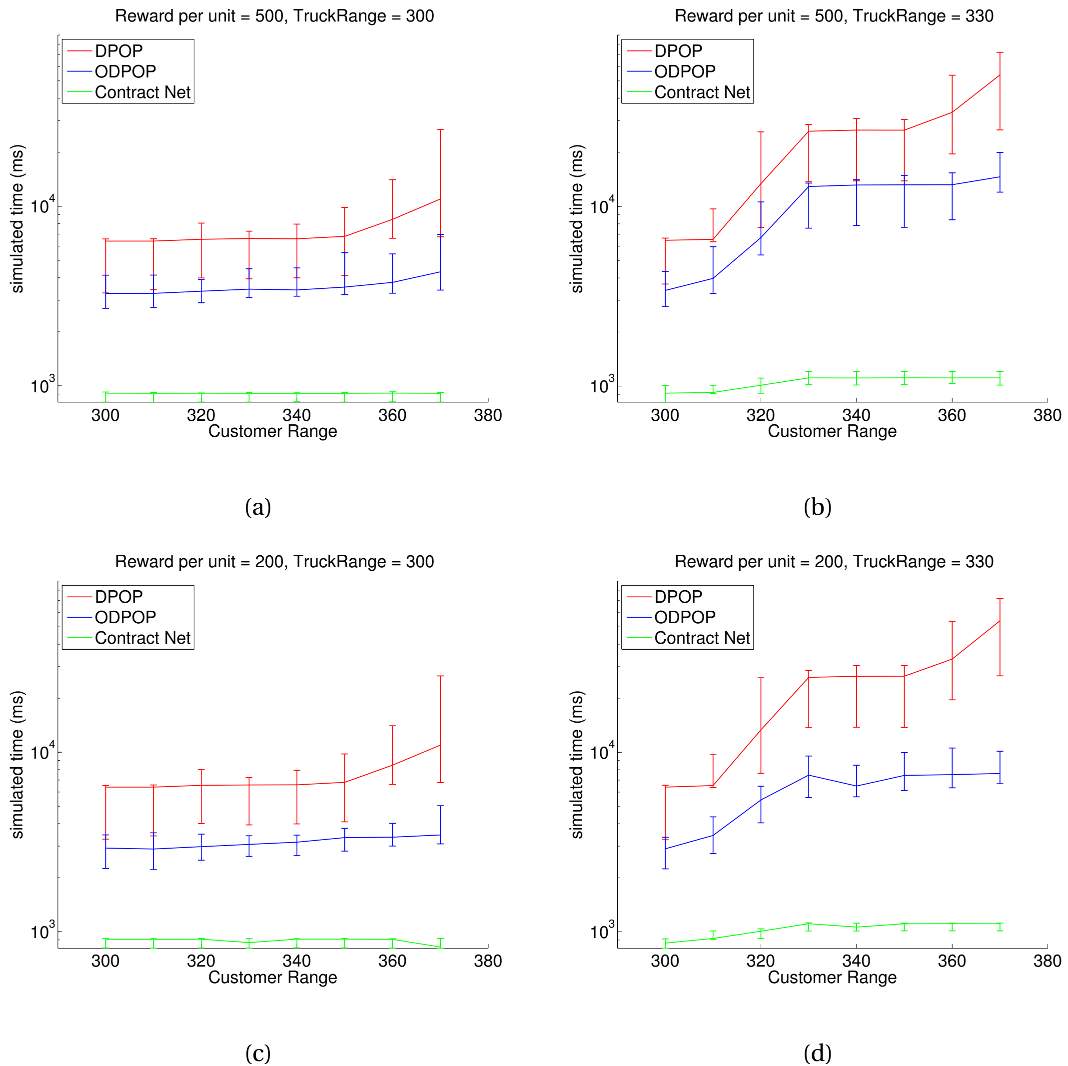


Figure 3.8: Simulated time in ms for the TTC problem

3.3.4 Preference Elicitation

The number of times a local problem is queried is captured by the NCCC measure. It measures the non-concurrent number of queries to the local problem. Figures 3.10, 3.11 and 3.12 show the results for the TTC problem, the meeting scheduling problem and the graph coloring problem. When looking at the TTC problem, notice that the Contract Net algorithm, where every agent needs to query its local problem only once per packet it is offered, shows a very low number of preferences elicited. Compared with the other algorithms, O-DPOP consistently performs the smallest number of queries to the local solver. Interestingly, the difference between O-DPOP and the other algorithms is greater for the instances with a reward of 200. This is because, with a reward of 200 the best-first order is not guaranteed any more, and O-DPOP thus believes it needs to

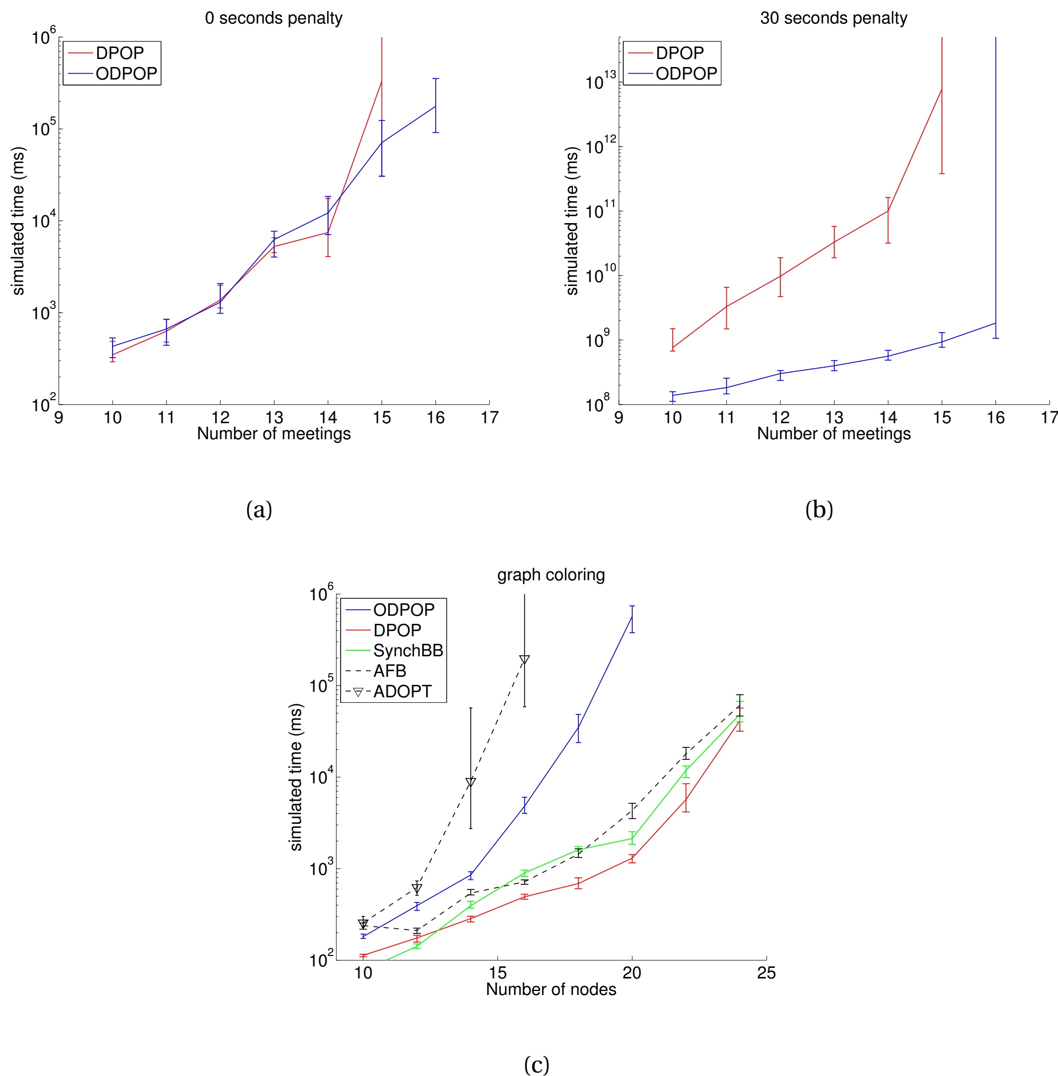


Figure 3.9: Simulated time in ms for the meeting scheduling problem (a, b, c) and the graph coloring problem (d)

collect less information from the trucks than it actually would have to.

The courier range determines the amount of coordination needed, i.e. the higher the courier range, the more possible conflicts. This can be seen from the behavior of O-DPOP. For a small courier range, the locally optimal solutions are more likely to be globally optimal. Thus for smaller courier range, O-DPOP will need to go through a smaller number of local solutions. Interestingly, although the local-search algorithms had the same amount of time as O-DPOP, they perform a higher number of constraint checks. Of the local-search algorithms, MGM2 performs the most number of constraint checks.

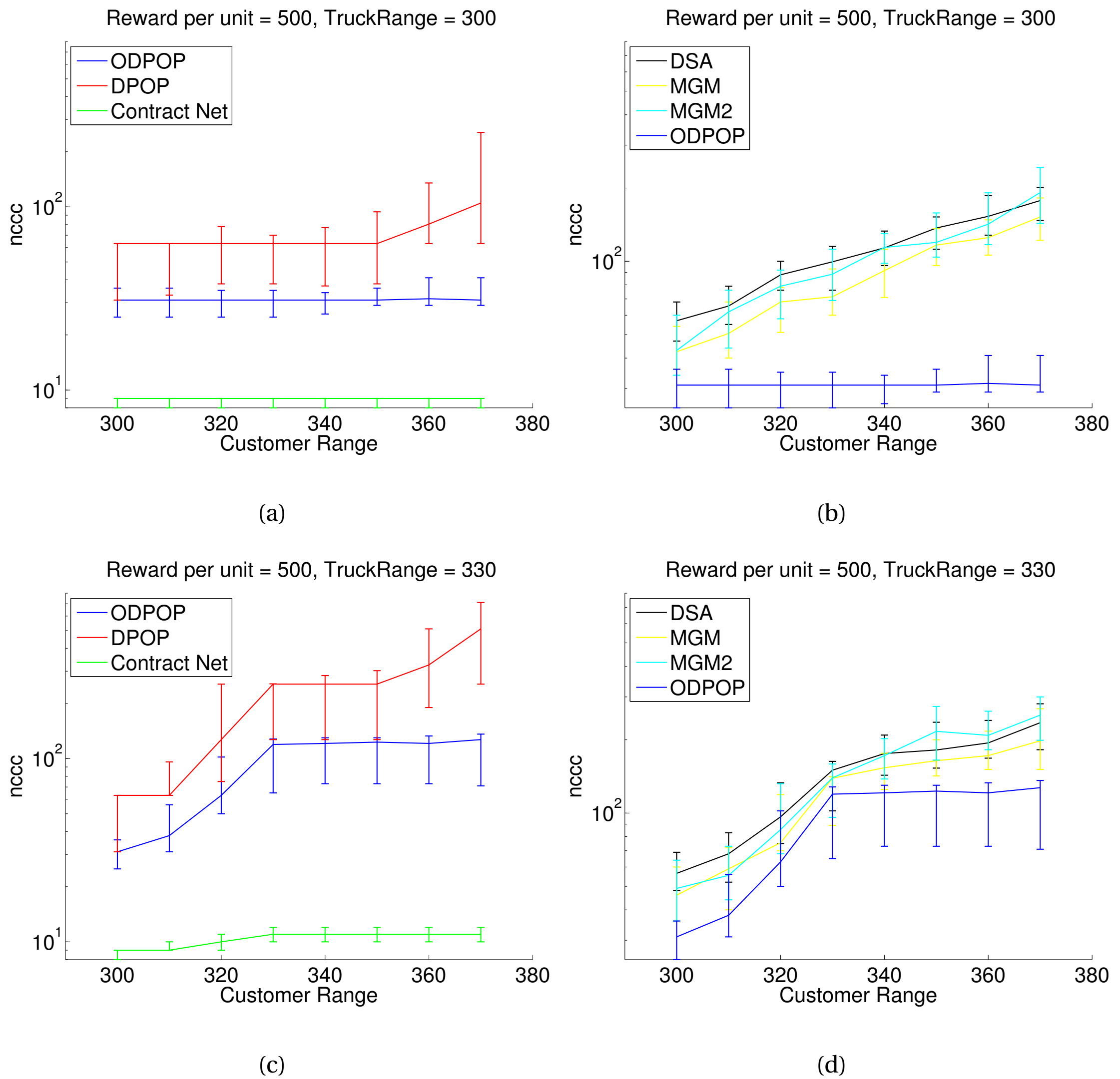


Figure 3.10: NCCC count for the TTC problem(1)

That incremental elicitation results in a smaller number of queries to the local problem can clearly be seen from the graph coloring and meeting scheduling results. For both problems, O-DPOP needs significantly less constraint checks to find an optimal solution. The reason that the runtime for O-DPOP is still higher than DPOP for the graph coloring problems is that O-DPOP needs to generate a best-first order. In the case of the graph coloring problem, this has to be done by ordering all possible assignments, giving O-DPOP a bigger overhead than DPOP. ADOPT performs a much larger number of NCCCs than all the other algorithms, while AFB, SynchBB, DPOP and the local search algorithms perform equally well.

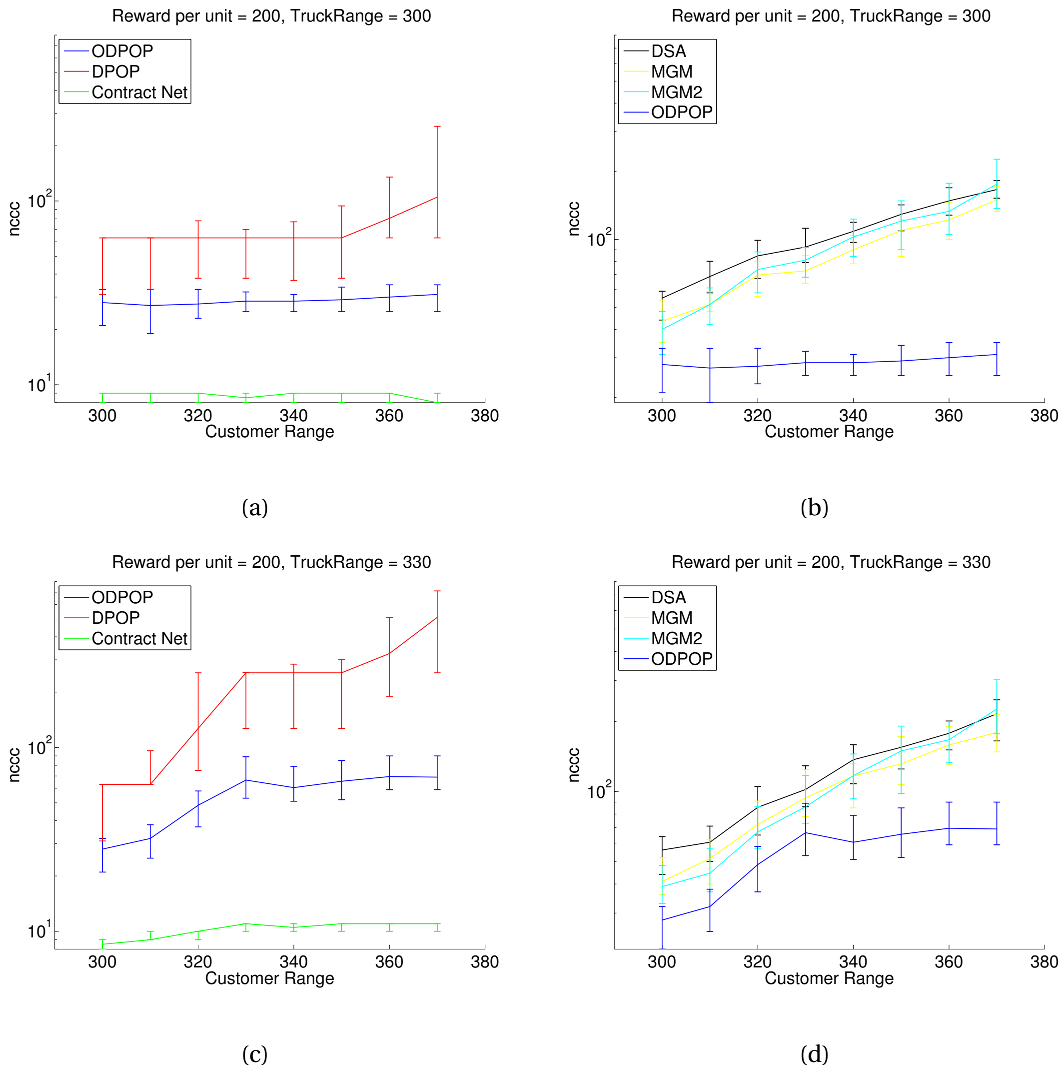


Figure 3.11: NCCC count for the TTC problem(2)

3.3.5 Messages and Information

Finally, when looking at the amount of information transmitted by the different algorithms, there is not much difference between them for the TTC problem(Figures 3.13 and 3.14). Interestingly, despite the fast runtime and the low number of constraint checks for the Contract Net, it still sends an amount of information that is similar to the other algorithms. For the meeting scheduling problem, it is interesting to note that O-DPOP and DPOP send the same amount of information, even though the number of NCCCs is much smaller for O-DPOP. This is due to the fact that O-DPOP must transmit not only the utility of an assignment, but also the assignment itself. The local-search algorithms transmit less data when the local problem takes more time, because they have time for fewer rounds. The spike in the information transmitted by DPOP for the

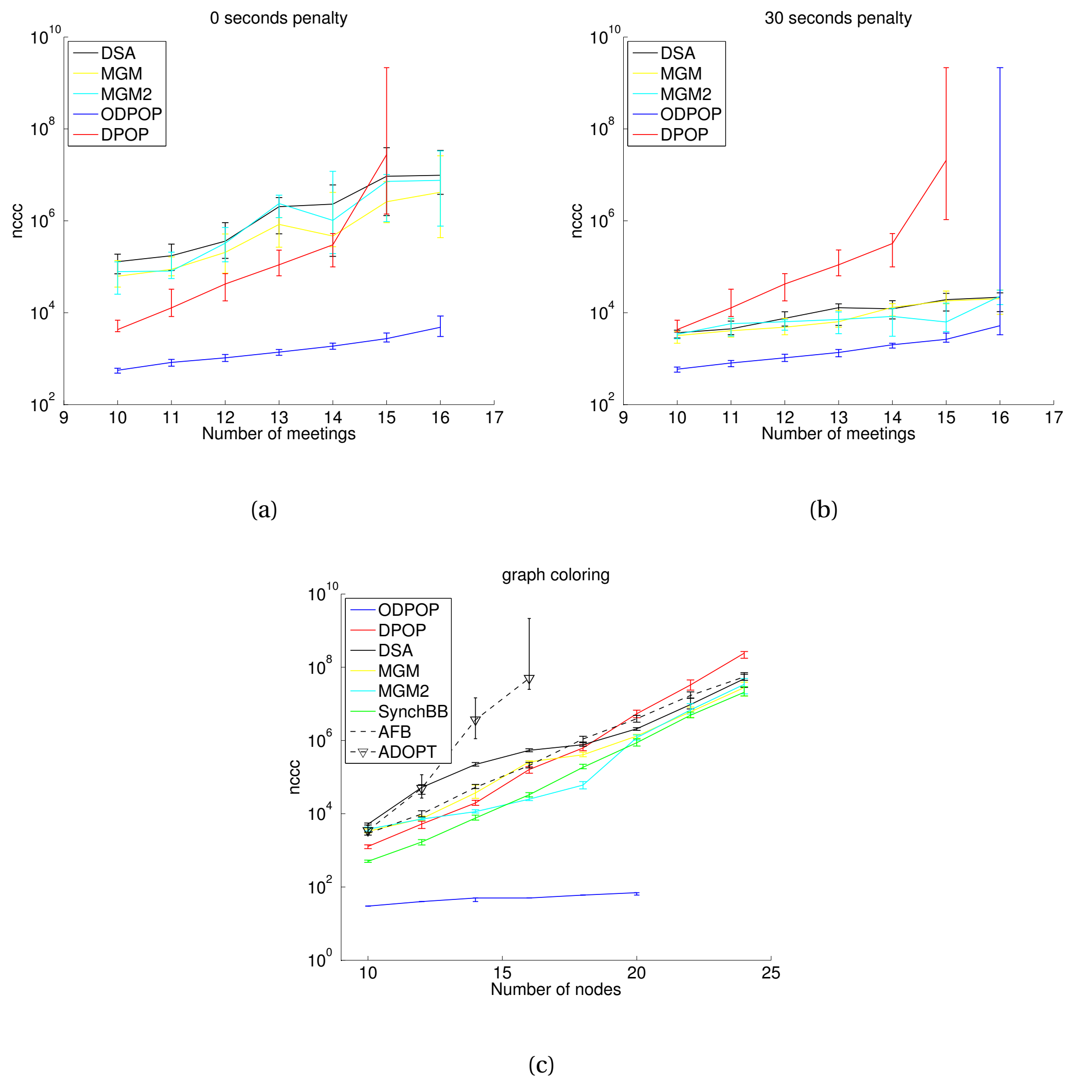


Figure 3.12: NCCC count for the meeting scheduling and graph coloring problems

meeting scheduling problems is due to the time out.

3.4 Conclusions

Coordination problems are ubiquitous in today's world. They come in many different shapes and forms, and there exist as many different ways of solving them. This Chapter deals with coordination problems that can be captured using constraints, both constraints between agents, but also internal constraints of the agents themselves.

Negotiation methods, like the Contract Net, have been designed to solve such problems. However, the Contract Net is only applicable in situations where there are services

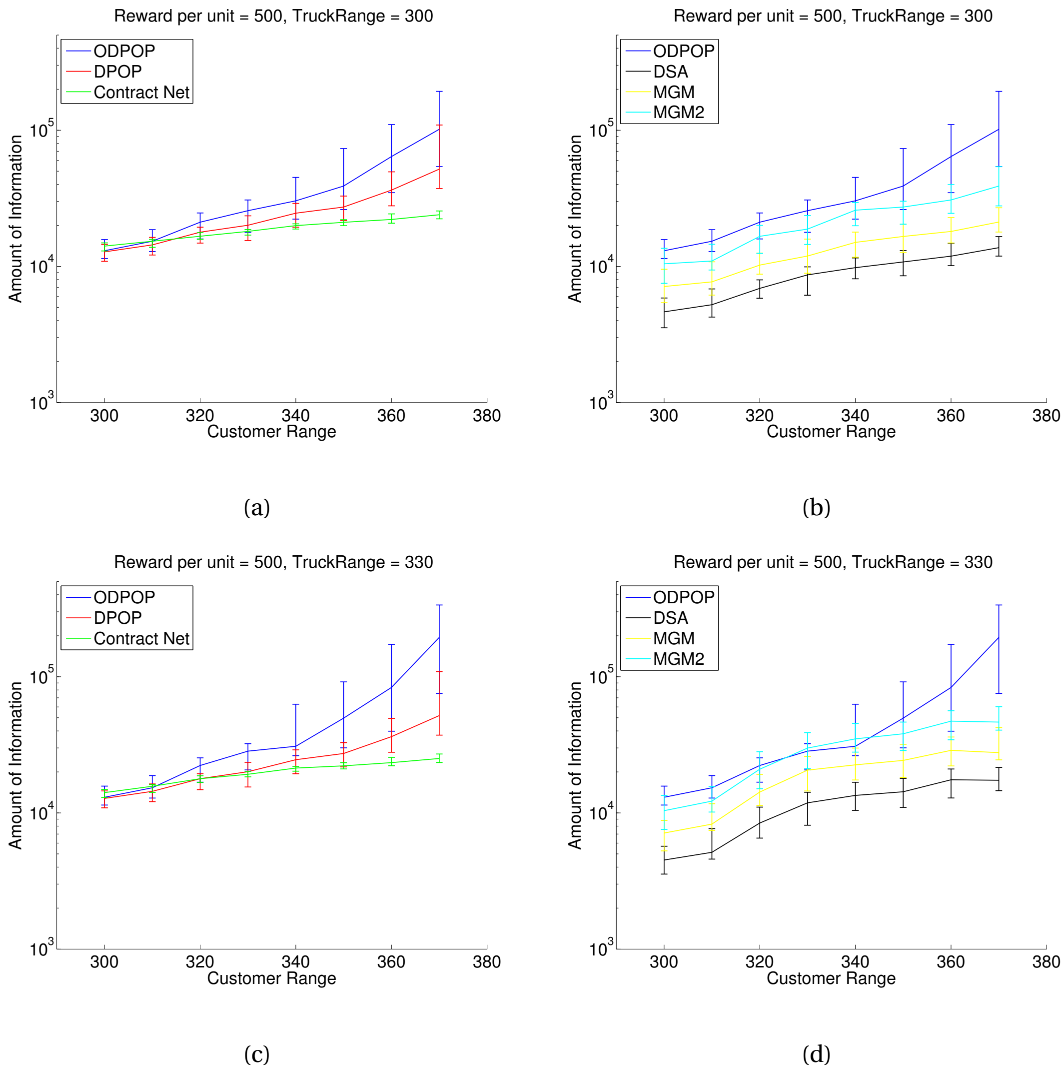


Figure 3.13: Amount of information sent for the TTC problem(1)

on offer. A task allocation setting is thus well suited for the Contract Net, or other market-based approaches for that matter. In scheduling settings, however, it is not clear who should offer what, and who bids for what. Hence, market-based approaches are not well suited for such domains.

We have shown that the DCOP paradigm provides an intuitive model for coordination problems. DCOP algorithms, however, have been designed with trivial local problems in mind. Agents that participate in coordination problems generally have non-trivial local problems. This Chapter investigated the behavior of existing DCOP methods when confronted with such non-trivial local problems. We have introduced the TTC problem, where each local problem is a vehicle routing problem, and evaluated a range of both complete and incomplete algorithms. Furthermore, we also evaluated the

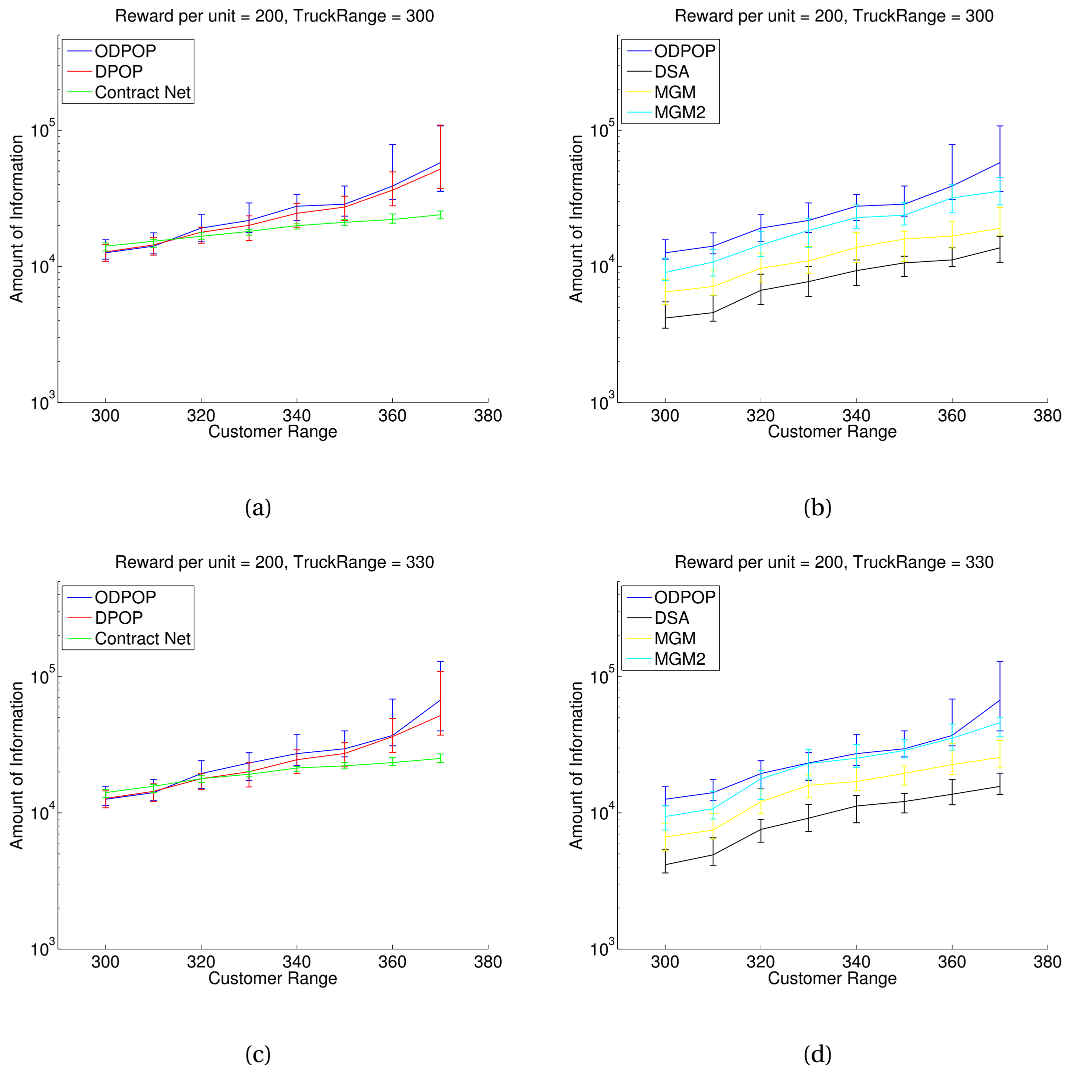


Figure 3.14: Amount of information sent for the TTC problem(2)

same set of algorithms on meeting scheduling problems and graph coloring problems that are traditionally used for DCOP evaluation. The difficulty of solving local problems was simulated using a penalty for querying the local problem.

Previous work has shown that, in a centralized setting, using incremental elicitation of preferences, i.e. reporting of preferences in a best-first order, results in efficient solving of problems with preferences. The experiments on meeting scheduling problems and graph coloring problems show that, when preferences are introduced in a DCOP setting, the O-DPOP algorithm, which uses incremental elicitation, significantly outperforms both complete algorithms in terms of runtime, and local-search algorithms in terms of solution quality. In many problems, however, obtaining a best-first order is not trivial. In the TTC problem, as introduced in Section 3.2.1, for example, generating the correct

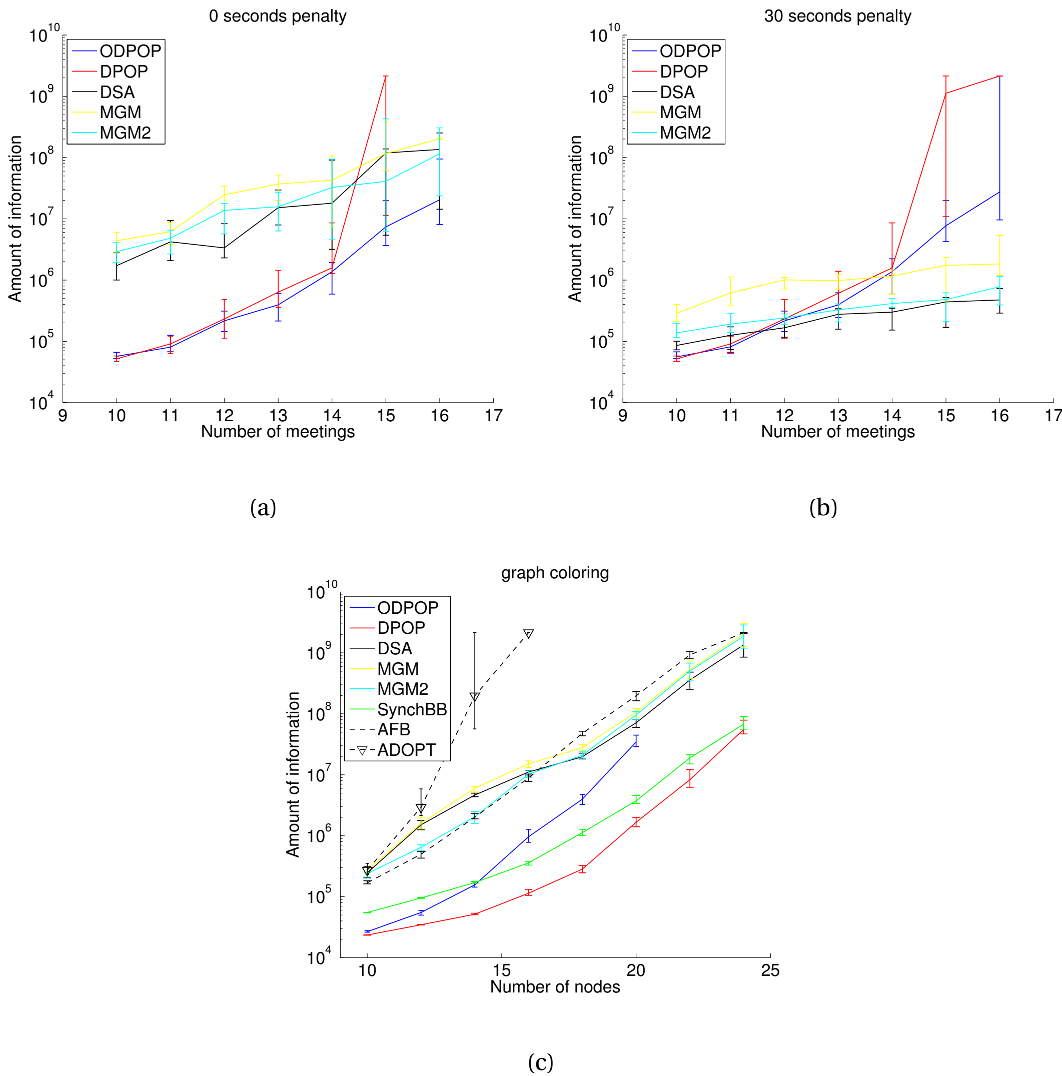


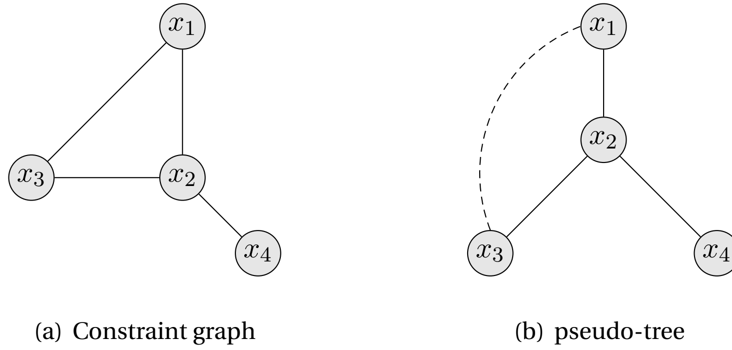
Figure 3.15: Amount of information sent for the meeting scheduling and graph coloring problems

best-first order requires solving a VRP problem for each possible packet assignments. Making some assumptions on the order of solutions, we introduced a local solver that was able to efficiently generate solutions, albeit not in a guaranteed best-first order. Experiments showed that O-DPOP, using incremental elicitation, outperformed other complete algorithms in terms of runtime, while still finding solutions that are close to optimal and significantly better than solutions found by local-search approaches.

4 DUCT: A UCB-based Sampling Method

Existing search algorithms for solving DCOPs systematically go through the search space, focussing on promising parts of the search space until they are certain that no good solution can be found in that part of the space. This can, however, lead to an exhaustive search. In searching for an optimal solution, it is thus important to have a good balance between exploration of unseen parts of the search space, and exploitation of already seen solutions. In this chapter, we introduce an algorithm that attempts to do just that. The approach is inspired by the confidence-based approaches from the multi-armed bandits literature. A natural starting point is the UCB algorithm (Auer et al., 2002) and its variants for tree-structured problems, such as UCT (Kocsis and Szepesvári, 2006) and HOO (Bubeck et al., 2011). Such algorithms have been shown to be very successful in playing Go (Gelly and Silver, 2005). These algorithms, as well as the confidence bounds they use, assume stochasticity and smoothness. While this is not the case in our setting, we nonetheless show that UCB-based search on DCOPs not only significantly outperforms local-search in terms of solution quality, but is also able to provide good feasible solutions for problems where optimal methods (in particular, DPOP) are too complex. In addition, we provide a theoretical analysis of the proposed algorithm, based on weak assumptions on the problem structure.

We first introduce a simple sampling algorithm, that randomly chooses paths in the search space. Based on the random sampling algorithm, we then introduce a new bound-based sampling algorithm that attempts to provide a good balance between exploitation and exploration. We use bounds on the solutions seen so far, that are inspired by the Hoeffding bound (Hoeffding, 1963) as used in the UCB algorithm. The Hoeffding bound can be used to bound the probability that the difference between a set of observations and its mean is smaller than some error. The bounds are then used to determine when the algorithm can terminate, i.e. when the error is small enough. Although the assumptions underlying the Hoeffding bound do not necessarily hold in our setting, experiments show they still give a good indication of when to terminate.



(a) Constraint graph (b) pseudo-tree

Figure 4.1: The constraint graph for $f_1(x_1, x_2, x_3) + f_2(x_2, x_4)$ and one of its possible pseudo-trees

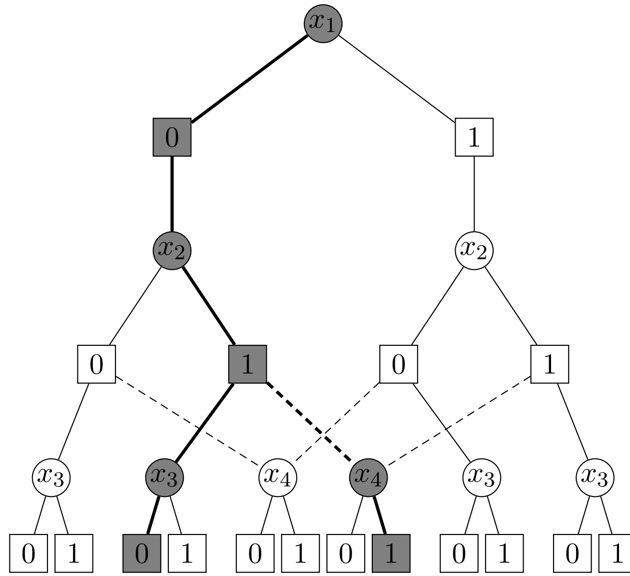


Figure 4.2: And AND/OR graph with $\forall_i D_i = \{0, 1\}$

To the best of our knowledge, no previous work uses sampling and confidence bounds for solving DCOPs. However, sampling has been used for solving both Constraint Satisfaction Problems (CSP) (Gogate and Dechter, 2003) and Quantified CSPs (Satomi et al., 2011). Furthermore, in (Léauté and Faltings, 2011b) sampling is used as a preprocessing method when dealing with stochastic problems, while (Stranders et al., 2012) introduces MAB-DCOPs, where stochasticity in a utility function is modeled as a multi-armed bandit. They introduce the HEIST algorithm, which uses a UCB inspired bound to minimize the expected regret.

The rest of this Chapter is organized as follows. Section 4.1 explains confidence bounds and describes the DUCT algorithm. Complexity results are given in Section 4.2, the experimental evaluation can be found in Section 4.3 and Section 4.4 concludes.

4.1 Distributed UCT

The problem of solving a DCOP is NP-complete. Thus, although complete methods work well for small, well structured problems, they will break down when problems become too big. As we have seen in Chapter 2, the main objective when solving a DCOP is to optimize a global function f. This function f is decomposable in a set of factors, i.e. $f \triangleq f_1 + \ldots + f_m$, allowing for a concurrent search of separate parts of the problem. The structure of the search space can be captured by an AND/OR graph (Dechter and Mateescu, 2004). Such a graph consists of alternating AND nodes and OR nodes, where OR nodes represent alternative solutions to the problem, and AND nodes represent a problem decomposition. Given a constraint graph and a possible pseudo tree (Figure 4.1), a corresponding AND/OR graph of the pseudo-tree is shown in Figure 4.2, with the squares and circles representing AND and OR nodes respectively. Note that since the subproblem rooted at x_4 does not depend on the value of x_1, we merge the two subgraphs corresponding to $x_1 = 0$ and $x_1 = 1$ into a single subgraph. A *path* represents an assignment to all variables. At AND nodes, the path *branches* to all the children, while at an OR node, the path *chooses* only a single child. The bold lines in Figure 4.2 constitute a path, and represent the assignment $\{x_1 = 0\,, x_2 = 1\,, x_3 = 0\,, x_4 = 1\}$.

To prevent confusion between the pseudo-tree and the AND/OR graph, nodes in the pseudo-tree are from here on called agents, while nodes in the AND/OR graph are nodes. The algorithm described in this chapter is designed for *minimization* problems, but can easily be converted to work for maximization problems.

4.1.1 Random Sampling: The RANDOM algorithm

DCOP search algorithms, such as ADOPT, operate by sending *context* messages down the pseudo-tree. A context is an assignment to all the variables in the receiving variable's separator. Each constraint is enforced by the lowest priority variable that participates in the constraint. Based on the received context, variables check the constraints that they enforce, systematically choose a value for themselves, and in this way explore the search space. A different, but simple, approach would be to randomly sample the search space. Lets call this algorithm RANDOM. A description of the algorithm can be found in Algorithm 6.

The root agent starts sampling by randomly selecting a value for its variable, and sending a CONTEXT message containing this variable assignment to all its children. Every time an agent k receives a CONTEXT message, containing a context a, it randomly chooses a value $d \in D_k$ for its variable, appends this to a, and sends this enlarged context to its children using a CONTEXT message. This process stops when the leaf agents are reached. At this time, the algorithm has selected a path through the AND/OR

tree. Based on the received context a, the leaf agents now calculate the minimal value the sum of the constraints they enforce can take:

$$y_k^t = \min_{d \in D_k} \ell^k(a, d) \tag{4.1}$$

where t denotes that this is the t-th sample taken by agent k, and $\ell^k(a, d)$ is the sum of the values of the constraints enforced by agent k. y_k^t is sent up the path using a COST message. Every subsequent agent calculates its own COST message using the following equation:

$$y_k^t = \ell^k(a, d) + \sum_{k' \in children_k} y_{k'}^t \tag{4.2}$$

Here $children_k$ denotes the children of agent k. The cost is thus the sum of the samples reported by its children plus the value of its local problem.

4.1.2 Termination

As a DCOP is distributed, a local termination condition is necessary. To this end, each agent k stores, for each context a it has received, the best value for y_k it has seen so far. More formally, let a_k^t be the context received by agent k at time t, and x_k^t be the value chosen by agent k at time t, then each agent stores the following:

- $\hat{\mu}_{a,d}^t$: the lowest cost found for value d under context a:

$$\hat{\mu}_{a,d}^t \triangleq \min \left\{ y_k^l \,\middle|\, l \leq t : a_k^l = a, x_k^l = d \right\}, \tag{4.3}$$

- $\hat{\mu}_a^t$: The lowest cost found under context a, $\hat{\mu}_a^t \triangleq \min_d \hat{\mu}_{a,d}^t$ and the value with the lowest cost $\hat{d}_a \triangleq \arg\min_d \hat{\mu}_{a,d}^t$

An agent terminates, when the difference between the expected optimal solution and the currently found best solution is 'small' enough. In order to determine this difference, a bound very similar to the Hoeffding (Hoeffding, 1963) bound is used. It is stressed, however, that this bound is used heuristically. The Hoeffding bound can be used to bound the error on the mean of a set of observations. Given n observations, with probability $1 - \delta$ an upper bound on the is given by

$$\sqrt{\frac{ln \frac{2}{\delta}}{n}} \tag{4.4}$$

Experiments have shown that this bounds also performs well in a DCOP setting. Hence, taking as n the number of times a value has been sampled under a context a, repre-

sented by

- $\tau^t_{a,d}$: The number of times value d has been selected for variable x_k under context a:

$$\tau^t_{a,d} \triangleq \sum_{l=1}^{t} \mathbb{I}\left\{a^l_k = a \wedge x^l_k = d\right\} \tag{4.5}$$

an agent k now terminates when the following two conditions are met:

- Its parent has terminated (this condition trivially holds for the root agent);
- The following equation holds for the last context a reported by its parent:

$$\max_{d \in D_k} \hat{\mu}^t_a - (\hat{\mu}^t_{a,d} - \sqrt{\frac{\ln \frac{2}{\delta}}{\tau^t_{a,d}}}) \leq \epsilon \tag{4.6}$$

  where ϵ and δ are parameters of the algorithm. Equation (4.6) holds when the biggest gap between the currently best value and the minimal lower bound on all values is smaller than ϵ. Note that, although the square root term has the form of a Hoeffding bound, it is used here heuristically.

When an agent k terminates, it adds $x_k = \hat{d}_a$ to the context a, and sends a F-CONTEXT message to its children, to signal its termination. Note that, since $\tau^t_{a,d}$ always increases, the algorithm is guaranteed to satisfy this condition in a finite number of steps. After an agent has terminated, its children continue sampling until their termination condition is met. By setting its value to $\hat{d}_a$, upon termination the agents have reconstructed the best global assignment seen so far.

4.1.3 Normalization

The above assumes that the range of the *global* cost is $[0, 1]$. In general DCOP problems this is not the case. Hence a normalization procedure must be carried out beforehand. An upper bound on the global cost can be found by summing up all the maximal values of all the individual local cost functions. This can be done in a bottom-up procedure, where each agent determines the highest value for its local problem, adds this value to the sum of the values received by its children, and then reports the new value to its parent. As soon as the root of the pseudo-tree has received this upper bound, UB, every local constraint is shifted to the left by subtracting its minimal value, and then divided by UB using a top-down procedure to disseminate the upper bound to all the agents.

4.1.4 Hard Constraints

Hard constraints, where there is some infeasible set $\mathcal{D}_I \subset \mathcal{D}$ such that $\forall \mathbf{x} \in \mathcal{D}_I \; f(\mathbf{x}) = \infty$ cause problems in two ways. Firstly, they break the normalization procedure. Secondly, the infeasible parts of the space should not be sampled.

The most straight-forward solution to both problems is to adjust the cost function such that all costs remain finite, while ensuring that $f(\mathbf{x}) < f(\mathbf{x'})$ for every $\mathbf{x} \notin \mathcal{D}_I$, $\mathbf{x}' \in \mathcal{D}_I$, i.e. feasible assignments have lower cost than infeasible assignments. However, normalization will squeeze the set of feasible costs into a small range, making optimization harder. As we verified experimentally, replacing infeasible costs with a penalty resulted in the algorithm minimizing constraint violations instead of looking for good feasible solutions.

A more promising approach is to use the infeasibility information to prune the search space. For this reason, firstly we only normalize so that $f(x) \in [0,1]$ for $x \notin \mathcal{D}_I$. Secondly, all infeasible parts of $\mathcal{D}$ are *ignored* as soon as their infeasibility is evident, and a search for the next feasible solution is started. Infeasible assignments can easily be recognized by the variable that enforces the infeasible constraint. Thirdly, when a node a does not have an infeasible local problem, but all its children reported to be infeasible, a is considered to be infeasible as well. The resulting sampling method (see Algorithm 7) randomly draws elements from D_k until it has found a feasible value. Furthermore, lines 28-31 of Algorithm 6 detail that when a child reports an infeasible value, the value of the local problem corresponding to the current context a and the sampled value is set to ∞, after which a new (feasible) value is chosen and reported to the children.

4.1.5 Confidence Bounds: The DUCT algorithm

When an agent receives a context, one can say the search algorithm has reached an OR node in the AND/OR graph. In RANDOM, the next branch (variable value) on the path to the leaf nodes is chosen randomly. However, we could instead focus the search on more promising choices. One way to do that is by constructing a confidence bound B such that the best value for any context is at least B, and sampling the choice with the least bound. In fact, the problem each OR node faces is similar to a *multi-armed bandit problem*, for which confidence-bound-based algorithms such as UCB (Auer et al., 2002) are nearly optimal solutions. UCB has been extended to tree-structured domains in the form of the UCB on trees (UCT) algorithm (Kocsis and Szepesvári, 2006) and its variants BAST (Coquelin and Munos, 2007) and HOO (Bubeck et al., 2011). The main assumptions in the UCT variants are that the space is metric, the cost function is stochastic and its expected value satisfies a Lipschitz smoothness condition with respect to the metric. Our setting is quite dissimilar, mainly because it is hard to justify

Algorithm 6: SAMPLING algorithm for variable k

```
1  initialization
2    if root then
3      parentFinished = true;
4    else
5      parentFinished = false;
6  when received(CONTEXT(a)) from parent
7    if variable k is a leaf node then
8      ℓ_min = min_{d∈D_k} ℓ^k(a, d);
9      send(parent, COST(ℓ_min, ℓ_min))
10   else
11     d = sample(a, k);
12     for each child k' do
13       send(k', CONTEXT(a ∪ {x_k = d}))
14 when received(F-CONTEXT(a)) from parent
15   parentFinished = true;
16   if Eq. (4.6) satisfied then
17     for each child k' do
18       send(k', F-CONTEXT(i ∪ {x_k = d̂_a}))
19   else
20     d = sample(a, k);
21     for each child k' do
22       send(k', CONTEXT(a ∪ {x_k = d}))
23 when received(COST(y^t_{k'}), B^t_{k',}) from child k'
24   if received cost message from all children then
25     if parentFinished ∧ Eq. (4.6) satisfied then
26       for each child k' do
27         send(k', F-CONTEXT(i ∪ {d̂_i}))
28     else if parentFinished ∨ y^t_k = ∞ then
29       if y^t_k = ∞ then
30         ℓ^k(a, a^t_k) = ∞;
31       d = sample(a, k);
32       for each child k' do
33         send(k', CONTEXT(a ∪ {x_k = d}))
34     else
35       send(parent, COST(y^t_k, B^t_{k,}))
```

Algorithm 7: sample(a, k): random sampling

```
1 d = random value from D_k;
2 while ℓ^k(a, d) = ∞ do
3   d = random value from D_k;
4 return d
```

any smoothness assumption on typical DCOPs. However, under a different set of assumptions, we can construct a similar type of confidence bound.

When solving a general DCOP problem, little is known about the distribution of the solutions over the space. In order to still put bounds on the optimal solution of a given subspace, we need to assume some form of smoothness, i.e. when slightly changing the solution we don't change the global value too much. To this end, we put the following bound on the number of sub-optimal choices

Assumption 1. *Let λ be the counting measure on $\mathcal{D}$ and let $f^*(A) \triangleq \min\{f(\boldsymbol{x}) \mid \boldsymbol{x} \in A\}$, for all $A \subset \mathcal{D}$. There exists $\beta > 0$ and $\gamma \in [0,1]$ s.t. $\forall A \subset \mathcal{D}$, $\epsilon \geq \gamma^{1/\beta}$:*

$$\lambda\left(\{\boldsymbol{x} \in A \mid f(\boldsymbol{x}) > f^*(A) + \epsilon\}\right) \leq \lambda\left(A\right)\gamma\epsilon^{-\beta}. \tag{4.7}$$

This assumption is quite weak, as it does not guarantee how many choices will be arbitrarily close to the optimal. Nevertheless, it can be used to obtain bounds on the value $f^*(A)$ of any set A. Assume that we have taken T samples from A, with values $(y_k^1, \ldots, y_k^T)$. In the worst-case, these are the T worst values, and thus an upper bound on the gap between the best-found value and the optimal value in A is:

$$\epsilon \leq \left(\frac{\gamma\lambda\left(A\right)}{T}\right)^{1/\beta}. \tag{4.8}$$

For a given problem, the values of γ and β are unknown. Taking $\beta = 2$, and letting γ slowly increase over time will result in these bounds eventually holding. This results in the following UCB style confidence bound: (Auer et al., 2002)

$$L_{a,d}^t = \sqrt{\frac{2\lambda_a \ln \tau_a^t}{\tau_{a,d}^t}}, \tag{4.9}$$

where

- τ_a^t: the number of times context a has been received:

$$\tau_a^t \triangleq \sum_{l=1}^{t} \mathbb{I}\left\{a_k^l = a\right\} \tag{4.10}$$

- λ_a: the length of the path to the deepest leaf node.

The length of the path can be obtained during the normalization phase, when the upper bounds on the factors are propagated up. Putting $\lambda_a = 1$ transforms Eq. (4.9)

into the standard UCB bound. However, such a bound treats a node close to the root in the same way as a node close to the leaf nodes, while the space left to explore is much bigger for the former than for the latter. A similar observation has been made for other optimization settings in (Coquelin and Munos, 2007; Kleinberg, 2005). We therefore set λ_a to the length of the maximal path from a to a leaf node, i.e. the higher in the tree, the more samples a node must collect before reaching the same confidence level. Also note that when a value d is not sampled for a while, the bound slowly increases. As a result, no region of the search space will be ignored and a good balance between exploitation of currently known good branches and exploration of unknown parts of the search space is achieved.

Given a context, we use the following bound, inspired by the standard UCB bound as used in (Kocsis and Szepesvári, 2006). The bound combines the local factors with the best value seen so far and a confidence interval

$$B^t_{a,d} \triangleq \ell^k(a,d) + \hat{\mu}^t_{a,d} - L^t_{a,d} \tag{4.11}$$

In the case of a leaf agent, $B^t_{a,d} = \ell^k(a,d)$. Due to the deterministic and finite nature of the DCOP problems, at a certain point in time all nodes of a particular subtree have been sampled. When such a point is reached, further sampling of the space will not yield any new information. The bound as described in Eq. (4.11) is not able to recognize this situation. For the parent of a leaf node, by definition this situation already occurs after sampling it once, i.e. after sampling it once its confidence bound is reduced to 0. The following recursive bound, which is similar to the one employed in HOO (Bubeck et al., 2011), makes use of this information to allow agents to recognize when a part of the tree does not need to be sampled anymore

$$B^t_{a,d} \triangleq \ell^k(a,d) + \max\{\hat{\mu}^t_{a,d} - L^t_{a,d}, \sum_{k' \in children_k} B^t_{k'}\}, \tag{4.12}$$

where $B^t_{k'} = \min_{d' \in D_{k'}} B^t_{a',d'}$ is the bound reported by agent k' for context $a' = a \cup \{x_k = d\}$. For leaf agents, $B^t_{a,d} = \ell^k(a,d)$.

Bound (4.12) is an optimistic estimate of the optimal value for the sub-tree rooted at agent k for context a and choice d. Note that, when $B^t_{a,d} = \ell^k(a,d) + \hat{\mu}^t_{a,d}$, the optimal cost for the subtree rooted at agent k, for context a and choice d, has been found. After this occurs, d should be ignored under context a. More precisely, let $S^t_a = \{d \in D_k | B^t_{a,d} \neq \ell^k(a,d) + \hat{\mu}^t_{a,d}\}$ be the set of allowed sample values. After trying each value at least once (since otherwise Eq. (4.9) is undefined), the agent samples according to:

$$x^t_k \triangleq \arg\min_{d \in S^t_a} B^t_{a,d}, \tag{4.13}$$

with ties broken randomly. Also, the termination condition is now:

$$\max_{d \in S_a^t} \hat{\mu}_a^t - (\hat{\mu}_{a,d}^t - \sqrt{\frac{\ln \frac{2}{\delta}}{\tau_{a,d}^t}}) \leq \epsilon. \tag{4.14}$$

The description of DUCT follows the algorithm skeleton as provided in Algorithm 6. However, it uses the sampling procedure as detailed in Algorithm 8.

Algorithm 8: sample(a, k): DUCT sampling

```
1 if all values in D_k have been sampled at least once then
2 |  return arg min_{d∈S_a^t} B_{a,d}^t
3 else
4 |  d = random unsampled value from D_k;
5 |  while ℓ^k(a, d) = ∞ do
6 |  |  d = random unsampled value from D_k;
7 |  return d
```

4.2 Theoretical Analysis

A general problem-independent analysis of the DUCT algorithm for DCOP is not possible. One can always construct a counterexample such that there exists only one optimal solution $\mathbf{x}^*$, with all remaining choices of $\mathbf{x}$ being either infeasible or far from optimal.

Definition 12. *Let $\rho(T)$ be the regret after T steps:*

$$\rho(T) \triangleq \min \left\{ y_0^t \,\middle|\, t = 1, \ldots, T \right\} - f^*(\mathcal{D}). \tag{4.15}$$

Thus, an algorithm that samples all of $\mathcal{D}$ in fixed arbitrary order has regret bounded by Eq. (4.8) with $A = \mathcal{D}$, since an adversary could manipulate the problem so that the worst values are seen first. On the contrary, the regret of RANDOM, which selects assignments uniformly, i.e.: $\mathbf{x}_t \sim \text{Uni}(\mathcal{D})$, does not depend on the size of $\mathcal{D}$:

Theorem 1. *The expected regret of the* RANDOM *algorithm is $\mathbb{E}\, \rho(T) \in O(1/\beta T + \gamma^{1/(\beta+1)})$.*

Proof. If Assumption 1 holds, then the probability that the regret of RANDOM after T samples exceeds ϵ is bounded by:

$$\mathbb{P}\left(\bigwedge_{t=1}^{T} [f(\mathbf{x}_t) > f^*(\mathcal{D}) + \epsilon] \right) \leq \left(\gamma \epsilon^{-\beta} \right)^T. \tag{4.16}$$

It follows from Eq. (4.16) and the fact that the regret and probabilities are bounded that $\forall\epsilon$, $\mathbb{E}\,\rho(T) \leq \left(\gamma\epsilon^{-\beta}\right)^T + \epsilon$.

$$\mathbb{E}\,\rho(T) = \mathbb{E}\,\rho(T|T > \epsilon)\,\mathbb{P}\,(T > \epsilon) + \mathbb{E}\,\rho(T|T \leq \epsilon)\,\mathbb{P}\,(T \leq \epsilon) \tag{4.17}$$

$$\leq \left(\gamma\epsilon^{-\beta}\right)^T + \mathbb{E}\,\rho(T|T \leq \epsilon)\,\mathbb{P}\,(T \leq \epsilon) \tag{4.18}$$

$$\leq \left(\gamma\epsilon^{-\beta}\right)^T + \epsilon \tag{4.19}$$

Setting the derivative to zero, we find that $\epsilon_0 = \left(\beta T\gamma^T\right)^{1/(\beta T+1)}$. Replacing to find the tightest bound:

$$\mathbb{E}\,\rho(T) \leq \gamma^T(\beta T\gamma^T)^{-\frac{\beta T}{\beta T+1}} + (\beta T\gamma^T)^{\frac{1}{\beta T+1}} \tag{4.20}$$

$$\leq \frac{1}{\beta T} + (\beta T\gamma^T)^{\frac{1}{\beta T+1}} \tag{4.21}$$

The second term can be bounded as follows. Note that $\max_x x^{1/(1+x)} < \max_x x^{1/x}$ and that

$$\frac{d}{dx}x^{1/x} = \frac{d}{dx}e^{\frac{1}{x}\ln x} = e^{\frac{1}{x}\ln x}x^{-2}(1 - \ln x),$$

which has a root $x = e$. Consequently, $(\beta T)^{1/(\beta T+1)} < e^{1/e}$. Finally, $\gamma^{T/(\beta T+1)} \leq \gamma^{1/(\beta+1)}$. □

Let us now consider DUCT. Since we have no knowledge of β, γ, we slowly increase our bounds until they hold. Then, the algorithm focuses on nearly optimal branches.

Theorem 2. *Consider two disjoint sets $A_1, A_2 \subset \mathcal{D}$ with $f^*(A_1) = f^*(A_2) + \Delta$, $A \triangleq A_1 \cup A_2$. If τ is the number of times A is sampled then the number of times the sub-optimal set A_1 is sampled under* DUCT *is* $O(\Delta^{-2}\ln\tau + (\gamma/2)^{\beta/(2-\beta)} + e^{\gamma/2})$.

Proof. Let $\hat{\mu}_i^t$ be the best-measured value in A_i at time t. For our bounds to hold, we must sample A at least $e^{\gamma/2}$ times and A_i at least $\tau_i = (\gamma/2)^{\beta/(2-\beta)}$ times. Then, we have:

$$f^*(A_i) \in [\hat{\mu}_i^t - L_{i,t}, \hat{\mu}_i^t + L_{i,t}],$$

Since we select A_1 rather than A_2, we have: $f^*(A_2) \geq \hat{\mu}_2^t - L_{2,t} \geq \hat{\mu}_1^t - L_{1,t}$. But $\hat{\mu}_1^t \geq f^*(A_1) - L_{1,t}$. Consequently, in order to select A_1 it is necessary that $f^*(A_2) \geq f^*(A_1) - 2L_{1,t}$. From the definition, we obtain $\tau_i \leq \left(\frac{4}{\Delta}\right)^2 \ln\tau$. □

To complete the analysis, we prove a lower bound on the expected regret of any algorithm.

algorithm	λ_a	sampling
DUCT-A	$\lambda_a = 1$	Eq. (4.11)
DUCT-B	$\lambda_a =$ path length	Eq. (4.11)
DUCT-C	$\lambda_a = 1$	Eq. (4.12)
DUCT-D	$\lambda_a =$ path length	Eq. (4.12)

Table 4.1: The DUCT variants

Theorem 3. $\mathbb{E}\,\rho(T) \in \Omega(\gamma^{1/2\beta+T/2})$.

Proof. It is sufficient to consider a function g in only one variable, for which Assumption 1 holds with equality. Assume a random bijection $h : \mathcal{D} \to \mathcal{D}$. Then function $f(\mathbf{x}) = g(h(\mathbf{x}))$ also satisfies Assumption 1 and any algorithm is equivalent to RANDOM. Then $\mathbb{E}\,\rho(T) \geq \gamma^T \epsilon^{1-\beta T}$ as Eq. (4.7) holds with equality. Selecting $\epsilon = \gamma^{1/2\beta}$ completes the proof. □

4.3 Experimental Evaluation

The introduced algorithms are evaluated on the *meeting scheduling* domain introduced in (Maheswaran et al., 2003), on randomly generated *graph coloring* problems and on the channel allocation problem introduced below. We evaluate 4 different versions of the DUCT algorithm, which are detailed in Table 4.1

The different sampling algorithms are compared against both optimal algorithms (DPOP, O-DPOP, ADOPT, SynchBB and AFB) and non-optimal algorithms (DSA, MGM and MGM2). For the meeting scheduling and channel allocation problems ADOPT, SynchBB and AFB could not solve the majority of the problems within the given time, while the performance of DPOP was consistently better than the performance of O-DPOP. Furthermore, for the graph coloring problem, DPOP dominated all other complete algorithms. Hence, to increase the readability of the graphs, we omitted these algorithms from the discussion of the experimental results. Furthermore, RANDOM is used as a baseline sampling algorithm. Comparisons are made both on solution quality, amount of information transmitted and runtime, where runtime is measured using the *simulated time* notion as introduced in (Sultanik et al., 2007).

4.3.1 The Channel Allocation Problem

WiFi access points are almost everywhere these days, and unfortunately most use the same channel to broadcast and receive information. This leads to interference, and thus a smaller transmission speed and lower reliability of the network. If different access points would use different channels, the overall throughput of information can be much higher. Traditionally, such a channel allocation problem has been modeled as

a graph coloring problem (Balasundaram and Butenko, 2006). Access points that cause interference to one another are considered neighbors in the graph, and the different channels are the different colors. This is a simplification of the problem that abstracts away many important aspects. For example, the interference between different nodes depends not only on the channels that are chosen, but also on the power with which both transmit, and the distance between them. Since most allocation problems are over constraint, any solution must allow a certain amount of interference. In such cases, modeling the strength of the interference is important in obtaining a good solution.

The capacity of a channel is given by the Shannon-Hartley theorem, that provides a relationship between the power of a signal, and the signal noise. The relationship has the following form

$$C = W \log_2(1 + \frac{S}{N}) \tag{4.22}$$

where C is the capacity of a channel, W is the bandwidth of the channel, S the signal strength and N the total noise or interference power. A channel causes interference to another channel when the power of its signal is stronger than that of the background noise. The current WiFi standard allows for 13 channels. Two channels are overlapping, i.e. interfering, when the distance between them is not more than three channels. There are thus only 3 non-overlapping channels available. The allocation problem can be modeled as follows.

Let $x_i \in D_i = \{1, \ldots, n\}$ denote the channel allocated to access point i, with n the number of available channels, Δ the maximal channel distance for which interference can occur, P_i the signal strength of i at the source, and $d_{i,j}$ the distance between access points i and j. Let $\text{Ap}_i = \{x_j | \frac{P_j}{d_{i,j}^2} > N_b\}$ be the set variables representing access points that can cause interference to the signal of i, with N_b the background noise. The function $I(x_i, x_j)$ returns 1 when x_j chose an overlapping channel, and 0 otherwise

$$I_i(x_i, x_j) = \begin{cases} 1 & \text{if } |x_i - x_j| \leq \gamma, \\ 0 & \text{otherwise} \end{cases} \tag{4.23}$$

Then, given a variable assignment $\mathbf{x} \in \prod_{x_j \in Ap_i} D_j$, the capacity of access point i is modeled as

$$f_i(x_i, \mathbf{x}) = W \log_2(1 + \frac{P_i}{\sum_{x_j \in \text{Ap}_i} I(x_i, x_j) \frac{P_j}{d_{i,j}^2}}) \tag{4.24}$$

The goal is now to find an assignment $\mathbf{x} \in \prod_i D_i$ such that

$$\mathbf{x} = \arg\max_{\mathbf{y}} \sum_i f_i([\mathbf{y}]_{f_i}) \tag{4.25}$$

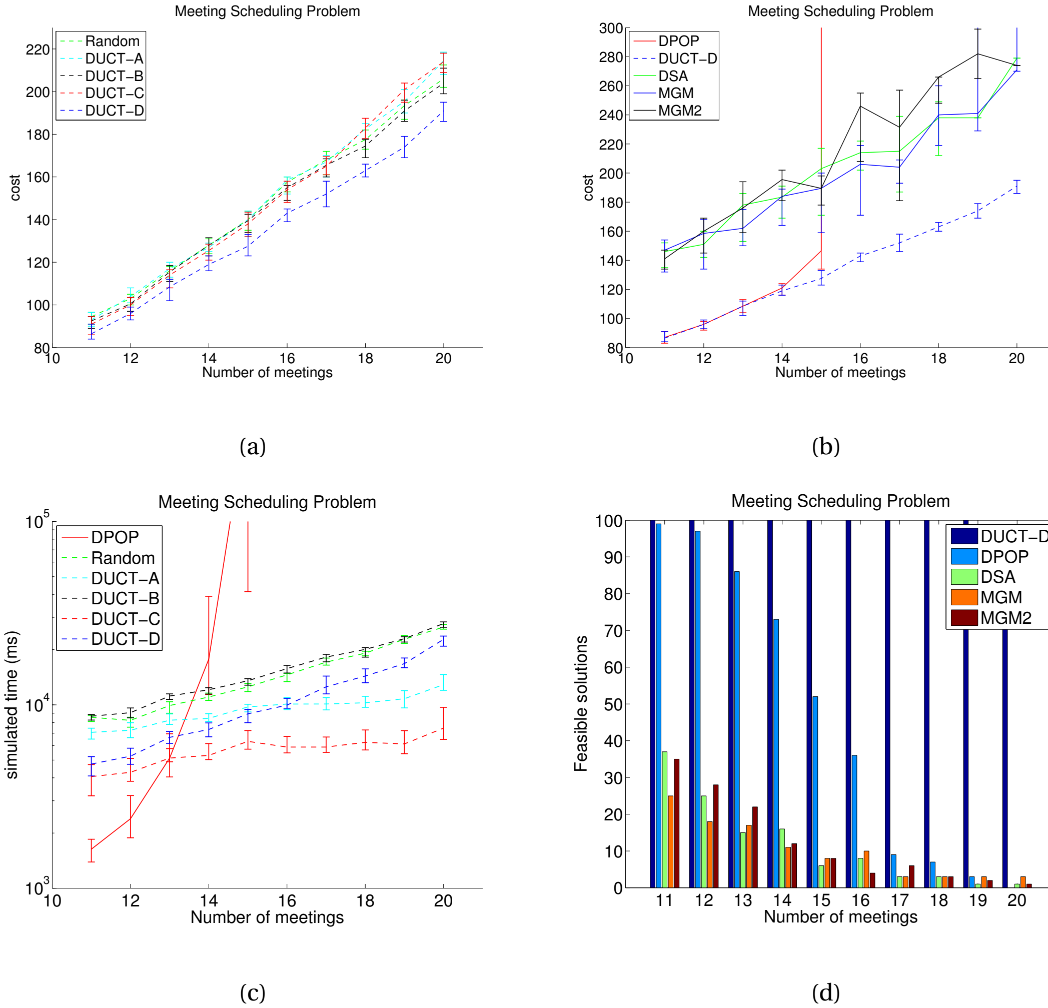


Figure 4.3: Meeting Scheduling Results

4.3.2 Experimental Setup

All evaluated algorithms are implemented in the FRODO platform (Léauté et al., 2009). For *meeting scheduling*, we used a pool of 30 agents, with 3 agents per meeting and between 11 and 20 meetings per instance. The cost for each meeting is randomly taken from $[0, 10]$. For the *graph coloring* problems, graphs with a density of 0.4 were randomly generated, the number of available colors was 3 and between 10 and 30 nodes per instance. Problems were modeled as Min-CSPs. For the channel allocation problems, the access points were randomly placed on a map of size 100x100, with a minimal distance of 4 units. The power of an access point ranges between 490 and 510, $W = 20$, the number of channels is set to 6 and $N_b = 1$. For each set of parameters, 99 problem instances were generated using the generators provided by FRODO, with a time out set at 15 minutes per problem. The values of both δ and ϵ were set at 0.05. The

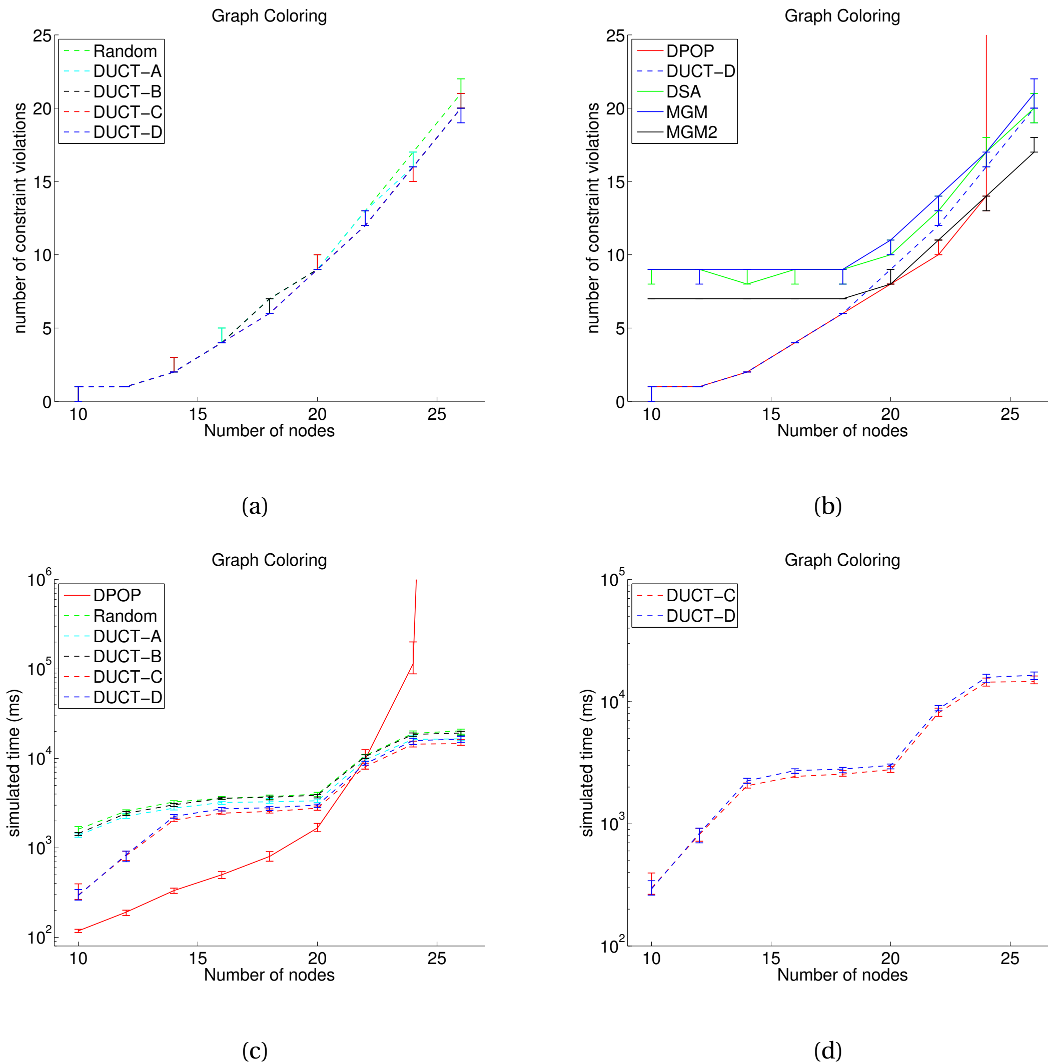


Figure 4.4: Graph coloring

experiments are run on a 64 core linux machine, with 2Gb per run.

DSA, MGM and MGM2 do not have any termination condition, and, to be fair, were given as much time as DUCT-D (which results show is the best performing DUCT variant).

The median is reported, together with 95% confidence intervals. In some instances, the optimal algorithms time out and consequently, their median can be higher than the median for some DUCT variants.

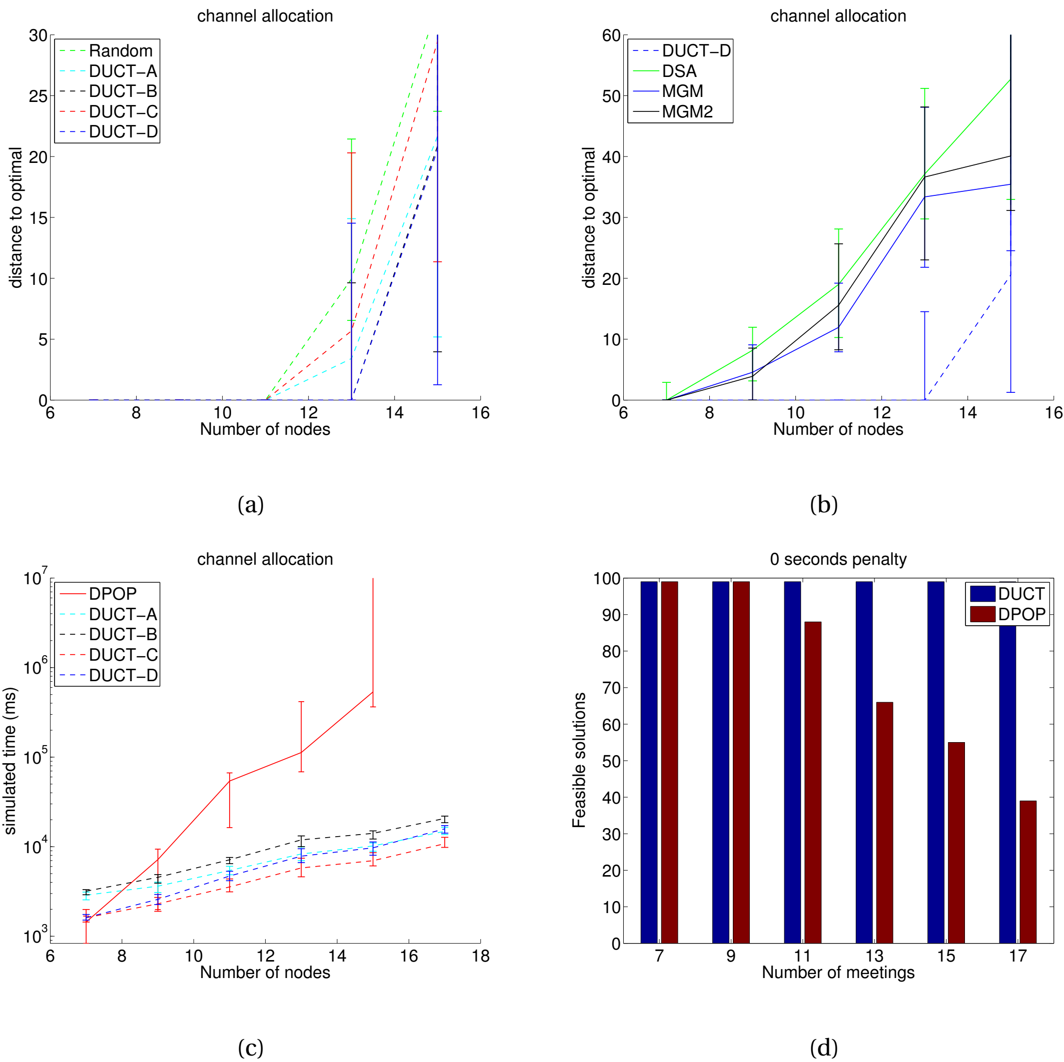
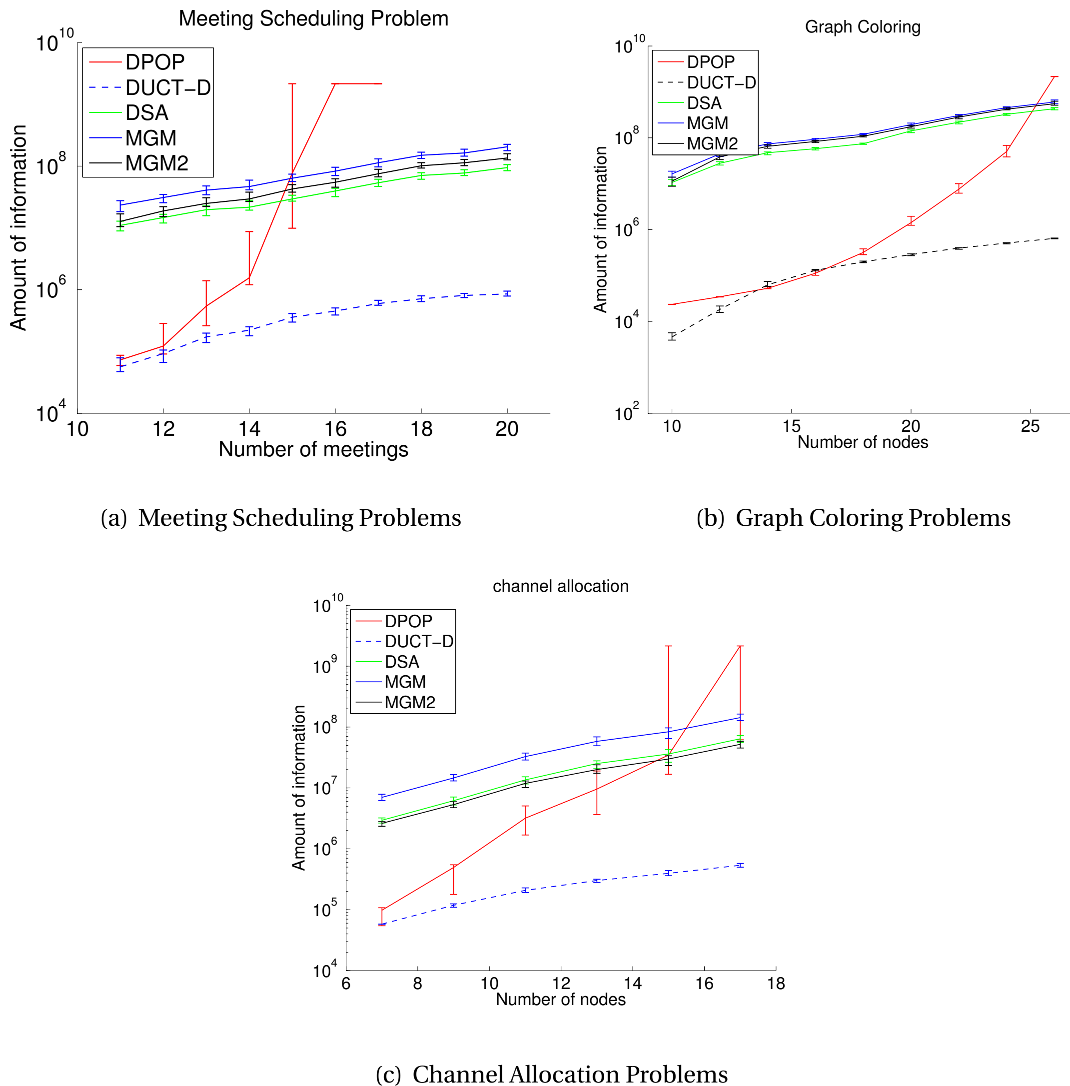


Figure 4.5: Channel allocation

4.3.3 Experimental Results

Figure 4.3(a) shows the performance of the different DUCT variants on the meeting scheduling problem. It is clear that DUCT-D finds the best solution, although Figure 4.3(c) shows that in terms of runtime DUCT-D is outperformed by both DUCT-A and DUCT-C. In general, the duct versions with $\lambda_a = 1$ terminate faster than the DUCT versions with $\lambda_a =$ path length, for the simple reason that the latter versions need more samples to make the confidence bounds small enough. That increasing the amount of time spent sampling does not lead to better solutions can be seen from the fact that although RANDOM takes the most time, it doesn't find the best solution. It is the steering of the search using the UCB bounds that leads to a better solution. When compared with the other algorithms (Figure 4.3(b)), one can see that DUCT-D finds



(a) Meeting Scheduling Problems

(b) Graph Coloring Problems

(c) Channel Allocation Problems

Figure 4.6: Amount of Information Exchanged

the optimal solution most of the time. In fact, 93% of the solutions found fall within the 5% error range set as the target for the DUCT algorithms. This means that the assumptions underlying the bound used characterizes the search space well. Where the DUCT variants are able to find a solution to most scheduling problems, DPOP is only able to solve problems with up to 14 meetings, after which it times out. The local-search algorithms, given as much time as DUCT-D, failed to find a feasible solution in more than 50% of the instances while DUCT-D always found a feasible schedule (See Figure 4.3(d)). To still give an idea of the performance of local-search algorithms, Figure 4.3(b) contains only the solved solutions.

Figure 4.4 shows the solution quality and runtime for the graph coloring problem. In

terms of the solution quality found, all DUCT variants find equally good solutions (Figure 4.4(a)). The performance of the local-search algorithms is rather poor for the smaller problems, but for the bigger problems they are on par with DUCT-D (Figure 4.4(b)), with MGM2 finding better solutions than the DUCT algorithms. When looking at runtime (Figure 4.4(c)), it is clear that the DUCT variants scale much better then DPOP, which fails to solve problems bigger than 24 nodes in the allotted time. DUCT-C and DUCT-D consistently run faster than DUCT-A, DUCT-B and RANDOM, with DUCT-C being slightly better than DUCT-D (Figure 4.4(d)). In the graph coloring setting, the influence of the value of λ_a is thus of less importance, while considering the child bounds brings a significant speed up[1].

For the channel allocation problem (Figure 4.5(a), 4.5(b)), note that we plot the distance to the optimal solution. As for the meeting scheduling problem, DUCT-D finds the best solution, and hence we only display its solution quality. One can see that for the small problems, DUCT-D is able to find the optimal solution, while only for the bigger problems its performance starts to deteriorate a little. As for the local-search algorithms, their performance deteriorates as the problem instances grow. When comparing the runtime of the DUCT variants to that of DPOP, it is clear that all DUCT variants scale much better. In fact, as can be seen from Figure 4.5(d), while the number of problems that DPOP solves drops rapidly, the DUCT variants are able to solve all instances.

Figure 4.6 shows the amount of information transmitted by the different algorithms. An interesting fact is that DUCT-D consistently sends less information than any of the other algorithms. Given the same amount of time, the local-search algorithms exchange up to 2 orders of magnitude more information. Furthermore, the amount of information sent scales well to bigger problems.

4.4 Conclusions

We introduced a sampling-based approach for solving Distributed Constraint Optimization Problems, inspired by Monte-Carlo tree search and the application of UCB to tree-structured problems. The result, a Distributed UCT algorithm (DUCT), takes advantage of the distributed and deterministic nature of the problem, as well as the hard constraints, in order to efficiently search the solution space.

Theoretically, we show that even though DCOP is not a smooth domain, we can still obtain meaningful bounds on the regret. However, the gap between the upper and lower bounds indicates that even better results may be possible. It is our view that a

[1]Since the graph coloring problem is in fact a constraint minimization problem, showing the number of *solved* solutions per algorithm does not provide any information. Hence we leave out the bar plot for this benchmark

more intricate analysis requires additional assumptions, which we shall investigate in further work.

The experiments show that DUCT not only can obtain good solutions within a reasonable amount of time, but also that for most problems it performs significantly better than local-search within the same time constraints. From the four variants of DUCT evaluated, DUCT-D shows the most consistent performance, both in runtime and in the quality of the solution found. Although adjusting the number of samples taken based on the position of the agent in the tree, and taking the child bounds into account do not provide a big improvement in terms of solution quality, combing the two allows for better solutions. The experiments also show that DUCT can handle much bigger problems than optimal algorithms, and performs nearly as well for smaller domains, while sending a much smaller amount of information, making it a very suitable algorithm for practical use in distributed environments. In all, DUCT is thus very well suited for solving DCOPs

5 FRODO 2

All algorithms described in this thesis have been implemented in the *FRODO 2* platform. FRODO 2 is a java-based platform for DisCSP, DCOP and StochDCOP, and is a complete re-design of the original FRODO platform developed by (Petcu, 2006). The platform has been developed in collaboration with Thomas Léauté and Radoslaw Szymanek, and has been implemented by me, Thomas Léauté and several students.

The main reasons for re-designing FRODO was that we wanted a platform that is both reliable, easily extendible and easily maintainable. Despite the existence of frameworks, such as FRODO, DisChoco (Ezzahir et al., 2007) or DCOPolis (Sultanik et al., 2007), none of them exhibited all the properties that we were looking for.

We used a modular design to obtain both the wanted extendibility and maintainability. On top of that, we forced ourselves to properly document all pieces of code. To make sure that FRODO 2 is reliable, we wrote *JUnit* tests for every new piece of code, and tested it on randomly generated problems.

Furthermore, in order to make use of a rich language to describe problem instances, we decided to make use of the XCSP format. The XCSP format has been developed for centralized CSP problems, and we had to adjust it to make it suitable for distributed optimization problems. After the release of DisCHOCO 2, which uses a slightly different adaptation of the XCSP format, we adjusted the FRODO format to make it compatible with theirs, and thus allow the use of their problem generators for generating benchmark problems.

5.1 FRODO Architecture

This section describes the multi-layer, modular architecture chosen for FRODO (Figure 5.1). We describe each layer in more detail in the following subsections.

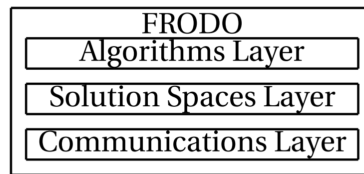


Figure 5.1: General FRODO software architecture.

5.1.1 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. Queues can send messages to each other, in the form of `Message` objects. Such messages can be send either through the use of shared memory (when the queues run in the same JVM), or via TCP. Classes implementing the `IncomingMsgPolicyInterface` can process messages received by a queue, by registering to this queue. Such classes are called modules. When a module registers to a queue, it can specify which types of messages it wishes to be notified of.

Queues can be instantiated with their own thread, making FRODO a multi-threaded framework. In order to allow timing measurements on a single machine, however, all queues can run in a single thread that assigns messages to queues one-by-one.

5.1.2 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. A solution space support different types of operations that can be used to reason over the spaces. Examples of operations are

- *join* merges two or more solution spaces into one;
- *project* removes one or more variables from the space by optimizing over their values in order to maximize utility or minimize cost;
- *bestFirstIterator* returns an iterator over all solutions in the space that iterates over solutions in a best first order.

Several implementations of solution spaces are provided by FRODO for different types of problems. Different parsers exist for different types of problem descriptions. The `XCSPparser` generates *hypercubes*, while the `XCSPparserjaCoP` outputs *JaCoP spaces* that serve as a bridge between FRODO and the centralized CSP solver JaCoP (JaCoP, 2011).

The solution spaces are optimized for both speed and space use. An hypercube, for example, is a table of values with entries for each possible variable assignment. When joining two hypercubes, the explicit join is only calculated when the space is used to exchange information, as in the UTIL messages send by DPOP. FRODO supports the following types of solution spaces:

- *Hypercubes*: an extensional representation of a space in which each variable assignment is associated with a utility value;
- *VRP Spaces*: a VRP space is a special space used in the Vehicle Routing Problem benchmark. It is an implementation of a specialized intensional space representing a single VRP constraint;
- *Meeting Spaces*: a meeting space is an intensional space representing the preferences an agent has over different meeting schedules 3.2.2. In addition, it implements a best-first order generator based on the procedure discussed in 3.2.2;
- *TTC Spaces*: a TTC space is an intensional space used to obtain utility information for the TTC problem 3.2.1, that implements a best-first order generator as specified in 3.2.1
- *Allocation Space*: an allocation space is an intensional space used in the Channel Allocation benchmark.
- *JaCoP Space*: a JaCoP space is a special space whose constraints are instantiated in an instance of the JaCoP solver.

5.1.3 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. For example the DPOP algorithm consists of the `DFSgenerationParallel`, `UTILpropagation` and `VALUEpropagation` modules.

The advantages of such a modular design are manyfold. For instance, as several algorithms operate on *DFS pseudo-trees*, the `DFSgenerationParallel` module 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. Combining various modules to produce algorithms is performed by simply declaring the modules and their parameter values in an XML agent definition file.

FRODO currently supports the following algorithms: SynchBB (Hirayama and Yokoo, 2003), MGM and MGM-2 (Maheswaran et al., 2004), ADOPT (Modi et al., 2005), DSA

(Zhang et al., 2005), DPOP (Petcu and Faltings, 2005a), S-DPOP (Petcu and Faltings, 2005b), MPC-Dis(W)CSP4 (Silaghi and Mitra, 2004; Silaghi, 2005), O-DPOP (Petcu and Faltings, 2006), AFB (Gershman et al., 2006a), MB-DPOP (Petcu, 2007), Max-Sum (Teacy et al., 2006), ASO-DPOP (Ottens and Faltings, 2008), P-DPOP (Faltings et al., 2008), P^2-DPOP (Léauté and Faltings, 2009b), $\mathbb{E}$[DPOP] (Léauté and Faltings, 2009a, 2011b), Param-DPOP, and $P^{3/2}$-DPOP (Léauté and Faltings, 2011a). 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 `algorithms.dpop.count`.

5.2 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, the number of Non-Concurrent Constraint Checks (NCCCs) (Gershman et al., 2006b) and the simulated time (Sultanik et al., 2007). 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.

5.3 O-DPOP implementation

The description of the O-DPOP algorithm as given in (Petcu and Faltings, 2006) does not define how received goods should be handled, and how the upper bounds are to be handled. For this thesis we implemented a data structure related to the HRJN data structure (Ilyas et al., 2003) used in databases to answer top-k queries. This section gives some more details on the implementation.

O-DPOP belongs to the inference-based family of DCOP algorithms. It is based on the concept of Open Constraint Programming (Faltings and Macho-Gonzalez, 2005), where a problem can be solved without having full information on the problem. The basic idea is that by eliciting the problem in a best first manner, a solution can be found before the full problem is elicited.

In inference-based algorithms, every variable must collect enough information to be able to make the optimal decision on its value given the values of ancestor variables. Given a variable x_i and an assignment a to the variables in its separator sep_i, let $f^i(a,d)$ be the utility of the subtree rooted at x_i when $x_i = d$. f^i consists of the local utility of variable x_i combined with the utilities of the subtrees rooted at the children of x_i. If $children_i i$ is the set of children of x_i, then $f^i(a,d)$ can be defined as

$$f^i(a,d) \quad = \quad \ell(a,d) + \sum_{j \in children_i} f^j(a,d) \tag{5.1}$$

The values for $f^j(a,d)$ are reported by the children of x_i. Note that by definition the separator sep_j should be a subset of sep_i. Hence, child j reports not on (a,d), but on some assignment *compatible* with it, where compatible is defined as

Definition 13 (Compatible)**.** *Two assignments a and a' are compatible if they agree on the values of the shared values.*

In O-DPOP, children report assignments one by one, in a best first order. This best first order allows one to determine bounds on the utilities of unseen assignments. Let f^j_{last} be the last utility reported by child j, and (a,d) an assignment for which $f^j(a,d)$ is not yet known, then child j will report at most f^j_{last} for this assignment. Hence, an upper bound on the utility of a can be defined as

$$UB^j(a,d) = \begin{cases} f^j(a,d) & \text{if child } j \text{ has reported a value} \\ & \text{compatible with } a, \\ f^j_{\text{last}} & \text{otherwise} \end{cases} \tag{5.2}$$

$$UB_i(a,d) = \ell(a,d) + \sum_{j \in children_i} UB^j(a,d) \tag{5.3}$$

Furthermore, for every assignment (a,d) on which child j has not yet reported a compatible assignment, $f^j(a,d) = -\infty$ when maximizing (and ∞ when minimizing). The upper bound on unseen assignments can be used to determine whether a variable is *valuation sufficient*, i.e. when it has enough information to determine the next best assignment.

Definition 14 (Valuation sufficient). *An agent i is* valuation sufficient *when there is an assignment (a,d) such that $f^i(a,d) \geq max_{a',d'} UB_i(a',d')$.*

The algorithm as given in (Petcu and Faltings, 2006) can be found in Algorithm 3. The basic idea is that the root agent initiates the search by sending an ASK message to all its children. When a variable receives an ASK message, it will start looking for its *next best assignment*, by sending ASK message to its children, that reply with assignments in a best first order. As soon as the variable is valuation sufficient, it replies to its parent with a GOOD message containing its next best assignment and the corresponding utility. Do note that the variable does not report its own value in the GOOD message.

5.3.1 Processing received goods

In the original publication, it is not defined how received GOOD messages should be processed. However, the efficient processing of such messages is paramount to quickly finding the next best solution. Two things need to be done when processing a GOOD message. First, the received partial assignment should be joined with all compatible assignments. Second, the upper bound should be updated.

A variable has two types of *sources* from which it gets utility information; from its own local problem and from its children. For the purpose of O-DPOP, all sources should be able to provide assignments in a best-first order. In order to properly calculate the upper bound the variable also needs to know the *best* assignment (a,d) for which no children have reported a value.

Given two *sorted* sources, a simple data structure presents itself. Let L be the left source, R the right source, list$_L$ the *sorted* list of assignments received from L and list$_R$

Algorithm 9: SimpleJ.next()

```
1  ass = checkQ();
2  if ass ≠ null then
3  |  return ass;
4  while at least one of the sources still has unreported assignment information do
5  |  S = chooseSource();
6  |  if S is finished then
7  |  |  min_S = −∞
8  |  else
9  |  |  a = S.next();
10 |  |  if a is the first utility received from S then
11 |  |  |  max_S = a.utility;
12 |  |  list1 and list2 are respectively the list belonging to S and the list belonging to
   |  |  the other source;
13 |  |  insert(a, list1, list2);
14 |  𝕋 = max(L⁺ + R₋, R⁺ + L₋);
15 ass = checkQ();
16 if ass ≠ null then
17 |  return ass;
```

Algorithm 10: checkQ()

```
1  if Q.isempty() then
2  |  ass = ℚ.peek();
3  |  if ass.utility > 𝕋 then
4  |  |  remove ass from ℚ;
5  |  |  if project = true then
6  |  |  |  project out and return ass;
7  |  |  else
8  |  |  |  return ass;
9  return null;
```

the list for R. Without loss of generality, assume that a new assignment x_l is received from source L. Then traverse through list_R to find all compatible assignments. Join x_l with every compatible assignment $x_r \in \text{list}_R$ and store them in $\mathbb{Q}$, which is a sorted list of assignments.

Algorithm 11: chooseSource()

```
1  if 𝕋 = R⁺ + L₋ then
2  |  return L;
3  else
4  |  return R;
```

Algorithm 12: SimpleJ.insert(a, list1, list2)

```
1  if a == null ∧ ¬list1.hasNext() then
2  |   min_list1 = −∞;
3  |   UB_list1 = −∞;
4  else
5  |   add a to list1;
6  |   min_list1 = a.utility;
7  |   for next in list2 do
8  |   |   join = a.join(next);
9  |   |   if project = true and then
10 |   |   |   if join is compatible with an already reported assignment then
11 |   |   |   |   continue;
12 |   |   ℚ.add(join)
```

In order to determine the next best assignment, an upper bound on unseen assignments can be calculated as follows. Let L^+ and L_- be the highest, respectively the lowest utility present in list$_L$, and R^+ and R_- be the same for list$_R$. Then the threshold $\mathbb{T}$ is defined as

$$\mathbb{T} = \text{max}(L^+ + R_-, R^+ + L_-) \tag{5.4}$$

All elements in $\mathbb{Q}$ with a utility higher than $\mathbb{T}$ are now guaranteed to be in best-first order. The data structure described above is called a Simple Join (SimpleJ) and can be found in Algorithm 9. Notice that next() alternates between L and R by querying the source that provides the min part of the current threshold. Only a change in this source's minimal value can result in a change to the threshold.

A SimpleJ also reports assignments in a best first order, and can thus be seen as a source. Hence, a variable that gets information from more than 2 sources can combine this information by *pipelining* multiple SimpleJs. For example, if a variable has a local problem and 2 children, the data received from the 2 children can be combined using a SimpleJ. The output of this SimpleJ can then be combined with the output of the local problem using another SimpleJ.

6 Conclusions

This thesis has dealt with the topic of Distributed Constraint Reasoning (DCR). DCR is a paradigm that can be used to model constraint optimization problems that are inherently distributed. The work in the DCR community can be subdivided into work on satisfaction problems (DCSP), and work on optimization problems (DCOP). In the former, it is enough to find a solution that satisfies all constraint, while in the latter one is interested in finding the best solution. This thesis focussed solely on optimization problems.

The DCOP framework can be used to model a wide range of distributed optimization problems. Initially the field focussed on synthetic problems, like the n-queens problem or the graph coloring problem. Later work has attempted to model more realistic problems. Examples are the use of DCOPs to solve sensor network problems, meeting scheduling problems or distributed vehicle routing problems.

This thesis consists of two parts. In the first part, we have looked at coordination problems with preferences, while in the second part of this thesis we introduce a new sampling-based algorithm for solving DCOPs.

6.1 Coordination Under Complex Local Preferences

Coordination among agents is an important, and interesting problem that has applications in many different problems and fields. In this thesis, we have argued that certain types of coordination problems are naturally modeled using the Distributed Constraint Optimization framework. Although early on it has been recognized that agents can have complex local problems, the role of preferences and how they are obtained has largely been ignored in the present DCOP literature.

We have introduced a general DCOP model for coordination problems with preferences, where we make an explicit distinction between the coordination constraints, and the

preference constraints of the agents. Based on this general model, we introduced two benchmark problems. The first, the Truck Task Coordination (TTC) problem, is a logistics problem where packets must be assigned to trucks. Each truck is assigned a region in which it can accept packets, while each packet can only be offered to trucks within a certain range. The local problem of an agent consists of a set of Vehicle Routing Problems (VRP), one for each possible packet assignment.

The second benchmark is an adaptation of the well known meeting scheduling problem. While the meeting scheduling problems presented in the literature do deal with preferences, these preferences are generally defined over single meetings. We introduce a variant where agents have preferences over their entire schedule.

Many existing DCOP algorithms assume that the preferences of the agents are known a-priori. DPOP, for example, requires an agent to submit preferences on all possible variable assignments, while ADOPT can request the preference on a single assignment multiple times. O-DPOP assumes that the best-first order over assignments is readily available. When the local problems defining the agents' preferences are non-trivial to solve, these assumptions do not hold anymore. We have evaluated the performance of DCOP algorithms under the presence of complex, non-trivial local problems. The algorithms have been evaluated on the TTC problem (a problem with complex, non-trivial local problems), the meeting scheduling problem (a problem with complex local problems) and the graph coloring problem. The results show that in the presence of preferences O-DPOP outperforms other complete DCOP methods in terms of runtime, queries to the local solver and the information transmitted.

Furthermore, the presence of non-trivial local problems implies that the best-first order is not a-priori available. For the TTC problem, we have used a local solver that was not guaranteed to provide a correct best-first order. Despite this, O-DPOP still found solutions that were close to the optimal solution, and better or on par with the solutions found by local-search algorithms. When dealing with coordination problems with preferences, using a best-first preference elicitation is thus an efficient strategy when finding a coordinated solution, even when the best-first order is only approximate.

6.2 Solving DCOPs Using Sampling

Methods for solving DCOPs can be divided into search-based, inference-based and local-search-based methods. The first two are complete methods, i.e. they are guaranteed to find the optimal solution. Their complexity, however is exponential in the size of the problem. The latter family of algorithms has a much smaller complexity, but does not provide any guarantees in terms of finding the optimal solutions. A notable exception is the Max-Sum algorithm, that is optimal in the absence of cycles in the

factor graph.

Solving a DCOP basically involves finding the maximal value of a function, ranging over multiple variables. In a centralized optimization setting, the UCT (Upper Confidence bounds on Trees) algorithm has shown to perform well. UCT is based on the Upper Confidence Bound (UCB) approach to solving the multi-armed bandit problem. In a multi-armed bandit problem, an agent must find a strategy for choosing gambling machines in such a way as to minimize its regret. The same strategy can be used to choose values for variables, where the reward is given by the function that is to be optimized.

In this thesis we have introduced a new UCB inspired algorithm, DUCT, for solving DCOPs. DUCT is a sampling-based algorithm that uses confidence bounds to steer the search into promising parts of the search space. Although the UCB algorithm, and its generalizations to tree structured problems, have been designed for problems where the function is unknown and stochastic, we have shown that similar confidence bounds are applicable to the deterministic DCOP setting as well. We have analyzed several different bounds, and evaluated them on a range of different problems. In particular, we have introduced a new benchmark problem based on the channel allocation problem. In a channel allocation problem, wireless access points must be assigned channels over which to transmit and receive data. Channels that are close to each other can cause interference. When two channels interfere with each other, the capacities of their channels are reduced. The goal is to assign channels to access points, so as to minimize the number of overlapping channels. Traditionally, channel allocation problems have been modeled as graph coloring problems. Since many such problems are over-constraint, i.e. interference cannot be prevented, it is important to incorporate the interference in the model such that the final solution maximizes the capacity of the selected channels.

Given a weak assumption on the structure of the problem, we give bounds on both the quality of the solution found, as on the number of times DUCT samples sub-optimal parts of the search space. Furthermore, our experiments show that the DUCT algorithm consistently performs well on different types of problems, while scaling better than complete DCOP approaches. It sends less information than both the complete algorithms. Given the same amount of time, local-search methods find worse solutions, while also exchanging more information than the DUCT algorithm.

6.3 Future Work

Our work on coordination problems with preferences has shown that a systematic approach, combined with incremental preference elicitation results in an efficient method to find an optimal solution. Calculating a best first order, however, can be a

difficult task. A similar problem can by found in the database literature. Much research in the database community deals with how to efficiently perform joins on databases. A particular type of join is the top-k join. In a top-k join, the aim is to produce the k best elements in a join, based on some ordering of the tuples. Some of the methods used to efficiently solve top-k joins can also be used to improve the performance of the O-DPOP algorithm. In particular, mixing random access queries with a best-first order has shown significant improvements in solving top-k joins. How to allow for random access queries in a DCOP setting is not obvious, since the information must be collected and combined from different hierarchically ordered sources.

The bounds given for the DUCT algorithm are based on a weak assumption on the structure of the problem to be solved. An important property needed for the centralized UCB algorithm to function is the smoothness of the function. The assumption used in this thesis is similar to a smoothness property, but there might exist other types of assumptions that result in better bounds. Performing a more thorough analysis of the problem structure of different types of DCOP problems could give a deeper insight into the performance of the DUCT algorithm on those problems. Furthermore, the analysis could shed light on the question whether better bounds exist for the deterministic DCOP setting.

Bibliography

(1993). *Proceedings of the 11th Conference on Artificial Intelligence (AAAI'93)*, Washington, DC, USA.

(2003). *Proceedings of the 2nd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'03)*, Melbourne, Australia.

(2003). *Proceedings of the 9th Internation Conference on Principles and Practice of Constraint Programming - CP 2003*, Kinsale, Ireland.

(2005). *Proceedings of the 23th National Conference on Artificial Intelligence (AAAI'08)*, Chicago, USA.

(2006). *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'08)*, Estoril, Portugal.

(2009). *Proceedings of the Distributed Constraint Reasoning Workshop (DCR'09)*.

(2011). *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI'11)*, Barcelona, Spain. AAAI Press.

Abdennadher, S. and Schlenker, H. (1999). Nurse Scheduling using Constraint Logic Programming. In *Proc. of the 20th National Conference on Artificial Intelligence (AAAI'05)*, pages 838–843, Orlando, USA.

Aji, S. and McEliece, R. (2000). The Generalized Distributive Law. *Information Theory, IEEE Transactions on*, 46(2):325 –343.

Armstrong, A. and Durfee, E. (1997). Dynamic Prioritization of Complex Agents in Distributed Constraint Satisfaction Problems. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI'97)*, pages 620–625, Nagoya, Japan.

Auer, P., Cesa-Bianchi, N., and Fischer, P. (2002). Finite Time Analysis of the Multiarmed Bandit Problem. *Machine Learning*, 47(2/3):235–256.

Bakker, R. R., Dikker, F., Tempelman, F., and Wogmim, P. M. (1993). Diagnosing and Solving Over-determined Constraint Satisfaction Problems. In *Proceedings of*

the 13th International Joint Conference on Artificial Intelligence (IJCAI'93), pages 276–281, Chambéry, France.

Balasundaram, B. and Butenko, S. (2006). Graph Domination, Coloring and Cliques in Telecommunications. In *Handbook of Optimization in Telecommunications*, pages 865–890.

Béjar, R., Domshlak, C., Fernándex, C., Gomes, C., Krishnamachari, B., Selman, B., and Valls, M. (2005). Sensor Networks and Distributed CSP: Communication, Computation and Complexity. *Artificial Intelligence*, 161:117–147.

Bessière, C., Maestre, A., Brito, I., and Meseguer, P. (2005). Asynchronous Backtracking Without Adding Links: a New Member in the ABT Family. *Artificial Intelligence*, 161:7–24.

Bessière, C., Maestre, A., and Meseguer, P. (2003). Distributed Dynamic Backtracking. In CP0 (2003).

Boutilier, C., Brafman, R., Geib, C., and Poole, D. (1997). A Constraint-Based Approach to Preference Elicitation and Decision Making. In *Proceedings of the AAAI Spring Symposium on Qualitative Decision Theory*.

Boutilier, C., Brafman, R. I., Domshlak, C., Hoos, H. H., and Poole, D. (2004). CP-nets: A Tool for Representing and Reasoning with Conditional Ceteris Paribus Preference Statements. *Journal of Artificial Intelligence Research (JAIR)*, 21:135–191.

Brafman, R., Rossi, F., Salvagnin, D., Venable, B., and Walsh, T. (2010). Finding the Next Solution in Constraint- and Preference-Based Knowledge Representation Formalisms. In *Proceedings of the 12th International Conference on Principles of Knowledge Representation and Reasoning (KR 2010)*, pages 425–433, Toronto, Canada.

Brito, I. and Meseguer, P. (2003). Distributed Forward Checking. In CP0 (2003), pages 801–806.

Bubeck, S., Munos, R., Stoltz, G., and Szepesvári, C. (2011). X-Armed Bandits. *Journal of Machine Learning Research*, 12:1655–1695.

Burke, D. A. and Brown, K. N. (2006). A Comparison of Approaches to Handling Complex Local Problems in DCOP. In *Proceedings of the ECAI'06 Workshop on Distributed Constraint Satisfaction (DCSP06)*.

Chandy, K. M. and Lamport, L. (1985). Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Trans. Comput. Syst.*, 3:63–75.

Chen, L. and Pu, P. (2004). Survey of Preference Elicitation Methods. Technical Report IC/2004/67, Ecole Politechnique Federale de Lausanne.

Conen, W. and Sandholm, T. (2001). Preference Elicitation in Combinatorial Auctions (Extended Abstract). In *Proceedings of the ACM conference on electronic commerce (ACM-EC01)*, pages 256–259. MIT Press.

Coquelin, P.-A. and Munos, R. (2007). Bandit Algorithms for Tree Search. In Halpern (2004).

Dechter, R. (2003). *Constraint processing*. Elsevier Morgan Kaufmann.

Dechter, R. and Mateescu, R. (2004). Mixtures of Deterministic-Probabilistic Networks and their AND/OR Search Space. In Halpern (2004), pages 120–129.

Decker, K. and Lesser, V. (1995). Designing a Family of Coordination Algorithms. *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95)*, pages 73–80.

Durfee, E., Lesser, V., and Corkill, D. (1989). Trends in Cooperative Distributed Problem Solving. *IEEE Transactions on Knowledge and Data Engineering*, KDE-1(1):63–83.

Ezzahir, R., Bessiere, C., Belaissaoui, M., and Bouyakhf, E. H. (2007). DisChoco: A Platform for Distributed Constraint Programming. In *Proceedings of the Eighth International Workshop on Distributed Constraint Reasoning (IJCAI-DCR'07)*, pages 16–27, Hyderabad, India.

Faltings, B., Léauté, T., and Petcu, A. (2008). 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.

Faltings, B. and Macho-Gonzalez, S. (2005). Open Constraint Programming. *Artificial Intelligence*, 161(1-2):181–208.

Fitzpatrick, S. and Meertens, L. G. L. T. (2001). An Experimental Assessment of a Stochastic, Anyti Decentralized, Soft Colourer for Sparse Graphs. In *Proceedings of the International Symposium on Stochastic Algorithms: Foundations and Applications*, SAGA '01, pages 49–64, London, UK, UK. Springer-Verlag.

Franzia, M. S., Freuder, E. C., Rossi, F., and Wallace, R. (2002). Multi-agent meeting scheduling with preferences: Efficiency, privacy loss, and solution quality. *Computational Intelligence*, 20:2004.

Garrido, L. and Sycara, K. (1995). Multi-agent Meeting Scheduling: Preliminary Experimental Results. In *Proceedings of the First International Conference on MultiAgent Systems ICMAS95*.

Gelain, M., Pini, M. S., Rossi, F., Venable, K. B., and Walsh, T. (2010). Elicitation Strategies for Soft Constraint Problems with Missing Preferences: Properties, Algorithms and Experimental Studies. *Artificial Intelligence*, 174:270–294.

Gelly, S. and Silver, D. (2005). Achieving Master Level Play in 9 x 9 Computer Go. In AAA (2005), pages 1537–1540.

Gershman, A., Meisels, A., and Zivan, R. (2006a). Asynchronous Forward-Bounding for Distributed Constraints Optimization. In *Proceedings of the 17th Eureopean Conference on Artificial Intelligence (ECAI'06),*, pages 103–107.

Gershman, A., Zivan, R., Grinshpoun, T., Grubshtein, A., and Meisels, A. (2006b). Measuring Distributed Constraint Optimization Algorithms. In *Proceedings of the AAMAS'06 Distributed Constraint Reasoning Workshop (DCR'06)*, pages 17–24.

Gogate, V. and Dechter, R. (2003). A New Algorithm for Sampling CSP Solutions Uniformly at Random. In *Proceedings of the 12th Internation Conference on Principles and Practice of Constraint Programming - CP 2006*, pages 711–715. Nantes, France.

Grinshpoun, T. and Meisels, A. (2008). Completeness and Performance of the APO Algorithm. *Journal of Artificial Intelligence Research (JAIR)*, 33:223–258.

Halpern, J., editor (2004). *Proceedings of the 20th Conference Conference on Uncertainty in Artificial Intelligence (2004)*, Banff, Canada.

Hamadi, Y. (1998). Backtracking in Distributed Constraint Networks. In *International Journal on Artificial Intelligence Tools*, pages 219–223.

Haralick, R. M. and Elliott, G. L. (1979). Increasing Tree Search Efficiency for Constraint Satisfaction Problems. In *Proceedings of the 6th International Joint Conference on Artificial Intelligence*, IJCAI'79, pages 356–364, Tokyo, Japan.

Hassine, A. B., Defago, X., and Ho, T. B. (2005). Agent-Based Approach to Dynamic Meeting Scheduling Problems. In *Proceedings of the 4th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'05)*, pages 1132–1139, Utrecht, Netherlands.

Hassine, A. B. and Ghédira, K. (2002). How to Establish Arc-Consistency by Reactive Agents. In *ECAI*, pages 156–160, Lyon, France.

Hassine, A. B., Ito, T., and Ho, T. B. (2004). Scheduling Meetings with Distributed Local Consistency Reinforcement. In *Proceedings of the 17th international conference on Innovations in applied artificial intelligence*, IEA/AIE'2004, pages 679–688.

Hirayama, K. and Toyoda, J. (1995). Forming Coalitions for Breaking Deadlocks. In *Proceedings of the 1st International Conference on Multiagent Systems*, ICMAS'95, pages 155–162.

Hirayama, K. and Yokoo, M. (2003). Distributed partial constraint satisfaction problem. In CP0 (2003), pages 222–236.

Hoeffding, W. (1963). Probability Inequalities for Sums of Bounded Random Variables. *Journal of the American Statistical Association*, 58(301):13–30.

Ilyas, I. F., Aref, W. G., and Elmagarmid, A. K. (2003). Supporting Top-k Join Queries in Relational Databases. In *In Proceedings of the 29th International Conference on Very Large Data Bases*, pages 754–765, Berlin, Germany.

JaCoP (2011). JaCoP java constraint programming solver. http://jacop.osolpro.com/.

Jain, M., Taylor, M. E., Yokoo, M., and Tambe, M. (2009). DCOPs meet the real world: Exploring unknown reward matrices with applications to mobile sensor networks. In *Proceedings of the 21th International Joint Conference on Artificial Intelligence (IJCAI'09)*, Pasadena, USA.

Keeney, R. L., Raiffa, H., and Rajala, D. W. (1979). Decisions with Multiple Objectives: Preferences and Value Trade-Offs. *Systems, Man and Cybernetics, IEEE Transactions on*, 9(7):403.

Kleinberg, R. (2005). Nearly Tight Bounds for the Continuum-armed Bandit Problem. In *Advances in Neural Information Processing Systems 17*, pages 697–704. MIT Press.

Kocsis, L. and Szepesvári, C. (2006). Bandit based Monte-Carlo Planning. In *Proceedings of the 17th European Conference on Machine Learning, (ECML06)*.

Lamma, E., Mello, P., Milano, M., Cucchiara, R., Gavanelli, M., and Piccardi, M. (1999). Constraint Propagation and Value Acquisition: Why we should do it Interactively. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI'99)*, pages 468–477, Stockholm, Sweden.

Léauté, T. (2011). *Distributed Constraint Optimization: Privacy Guarantees and Stochastic Uncertainty*. PhD thesis, Ecole Polytechnique Fédérale de Lausanne (EPFL), Lausanne, Switzerland.

Léauté, T. and Faltings, B. (2009a). E[DPOP]: Distributed Constraint Optimization under Stochastic Uncertainty using Collaborative Sampling. In DCR (2009), pages 87–101.

Léauté, T. and Faltings, B. (2009b). 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.

Léauté, T. and Faltings, B. (2011a). Coordinating Logistics Operations with Privacy Guarantees. In IJC (2011), pages 2482–2487.

Léauté, T. and Faltings, B. (2011b). Distributed Constraint Optimization under Stochastic Uncertainty. In *Proc. of the 25th National Conference on Artificial Intelligence (AAAI'11)*, pages 68–73, San Francisco, USA.

Léauté, T., Ottens, B., and Faltings, B. (2010). 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)*.

Léauté, T., Ottens, B., and Szymanek, R. (2009). FRODO 2.0: An Open-Source Framework for Distributed Constraint Optimization. In DCR (2009), pages 160–164.

Lesser, V. (1990). An Overview of DAI: Viewing Distributed AI as Distributed Search. *Journal of Japanese Society for Artificial Intelligence-Special Issue on Distributed Artificial Intelligence*, 5(4):392–400.

Lesser, V. R. (1998). Reflections on the Nature of Multi-Agent Coordination and Its Implications for an Agent Architecture. *Journal of Autonomous Agents and Multi-Agent Systems*, 1:89–111.

Liu, J. and Sycara, K. P. (1995). Exploiting Problem Structure for Distributed Constraint Optimization. In *International Conference on Multiagent Systems*. MIT Press.

Mackworth, A. K. (1987). Constraint Satisfaction. In Shapiro, S., editor, *Encyclopedia of Artificial Intelligence*, pages 205–211. J. Wiley and Sons, NY.

Maheswaran, R. T., Pearce, J. P., and Tambe, M. (2004). Distributed Algorithms for DCOP: A Graphical-Game-Based Approach. In *Proceedings of ISCA PDCS'04*, pages 432–439.

Maheswaran, R. T., Tambe, M., Bowring, E., Pearce, J. P., and Varakantham, P. (2003). Taking DCOP to the Real World: Efficient Complete Solutions for Distributed Multi-Event Scheduling. In AAM (2003), pages 310–317.

Mailler, R. and Lesser, V. (2003). A Mediation Based Protocol for Distributed Constraint Satisfaction. In *Proceedings of the IJCAI'03 Distributed Constraint Reasoning Workshop (DCR'03)*, pages 49–58, Acapulco, Mexico.

Marx, D. (2004). Graph Coloring Problems and Their Applications in Scheduling. In *Proceedings John von Neumann PhD Students Conference*, pages 1–2.

Meisels, A. and Razgon, I. (2001). Distributed Forward Checking with Dynamic Ordering. In *Proceedings of the IJCAI'01 Distributed Constraint Reasoning Workshop (DCR'01)*, pages 21–27.

Meisels, A. and Zivan, R. (2007). Asynchronous Forward-checking for DisCSPs. *Constraints*, 12:131–150.

Meseguer, P., Rossi, F., and Schiex, T. (2006). *Soft Constraints*, volume 2 of *Foundations of Artificial Intelligence*, chapter 9, pages 281 – 328. Elsevier.

Modi, P. J., Jung, H., Tambe, M., min Shen, W., and Kulkarni, S. (2001). A Dynamic Distributed Constraint Satisfaction Approach to Resource Allocation. In *Proceedings of the 7th Internation Conference on Principles and Practice of Constraint Programming - CP 2001*, pages 685–700, Paphos, Cyprus.

Modi, P. J., Shen, W.-M., Tambe, M., and Yokoo, M. (2003). An Asynchronous Complete Method for Distributed Constraint Optimization. In AAM (2003), pages 161–168.

Modi, P. J., Shen, W.-M., Tambe, M., and Yokoo, M. (2005). ADOPT: Asynchronous Distributed Constraint Optimization with Quality Guarantees. *Artificial Intelligence*, 161:149–180.

Morris, P. (1993). The Breakout Method for Escaping from Local Minima. In AAA (1993), pages 40–45.

Netzer, A., Meisels, A., and Grubshtein, A. (2010). Concurrent Forward Bounding for DCOPs. In *Proceedings of the AAMAS'10 Distributed Constraint Reasoning Workshop (DCR'10)*.

Nguyen, V., Sam-haroud, D., and Faltings, B. (2004). Dynamic Distributed BackJumping. In *Proceedings of the Joint Annual Workshop of ERCIM/CoLogNet on Constraint Solving and Constraint Logic Programming*, pages 71–85.

Nisan, N. (2006). *Bidding Languages for Combinatorial Auctions*, chapter 9. MIT Press.

Olteanu, Á., Léauté, T., and Faltings, B. (2011). Asynchronous Forward Bounding (AFB): Implementation and Performance Experiments. Semester project rep, Swiss Federal Institute of Technology (EPFL), Lausanne (Switzerland).

Ottens, B. and Faltings, B. (2008). Coordinating Agent Plans Through Distributed Constraint Optimization. In *Proceedings of the ICAPS'08 Multiagent Planning Workshop (MASPLAN'08)*.

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

Petcu, A. (2007). *A Class of Algorithms for Distributed Constraint Optimization*. Phd. thesis no. 3942, Swiss Federal Institute of Technology (EPFL), Lausanne (Switzerland).

Petcu, A. and Faltings, B. (2005a). DPOP: A Scalable Method for Multiagent Constraint Optimization. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI'05)*, pages 266–271.

Petcu, A. and Faltings, B. (2005b). S-DPOP: Superstabilizing, Fault-containing Multiagent Combinatorial Optimization. In AAA (2005), pages 449–454.

Petcu, A. and Faltings, B. (2006). O-DPOP: An algorithm for Open/Distributed Constraint Optimization. In *Proceedings of the 21th National Conference on Artificial Intelligence (AAAI'06)*, pages 703–708, Bosten, U.S.A.

Sandholm, T. (1993). An Implementation of the Contract Net Protocol Based on Marginal Cost Calculations. In AAA (1993), pages 256–262.

Satomi, B., Joe, Y., Iwasaki, A., and Yokoo, M. (2011). Real-Time Solving of Quantified CSPs Based on Monte-Carlo Game Tree Search. In IJC (2011), pages 655–661.

Sen, S. (1997). Developing an Automated Distributed Meeting Scheduler. *IEEE Expert: Intelligent Systems and Their Applications*, 12:41–45.

Shoham, Y. and Tennenholtz, M. (1992). On the Synthesis of Useful Social Laws for Artificial Agent Societies. In *Proceedings of the tenth national conference on Artificial intelligence*, AAAI'92, pages 276–281. AAAI Press.

Silaghi, M.-C. (2005). 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. AAAI Press.

Silaghi, M.-C. and Mitra, D. (2004). Distributed Constraint Satisfaction and Optimization with Privacy Enforcement. In *Proceedings of IAT'04*, pages 531–535.

Silaghi, M.-C., Sam-Haroud, D., and Faltings, B. (2000a). Asynchronous Search with Aggregations. In *Proceedings of the 17th National Conference on Artificial Intelligence and 12th Conference on Innovative Applications of Artificial Intelligence*, pages 917–922. AAAI Press.

Silaghi, M.-C., Sam-haroud, D., and Faltings, B. V. (2000b). Maintaining Hierarchical Distributed Consistency. In *In CP2000 DCS Workshop*.

Silaghi, M. C. and Yokoo, M. (2006). Nogood Based Asynchronous Distributed Optimization (ADOPT ng). In *Proceedings of the 5th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'06)*, pages 1389–1396, Hakodate, Japan.

Smith, R. G. (1988). The Contract Net Protocol: High-level Communication and Control in a Distributed Problem Solver. In Bond, A. H. and Gasser, L., editors, *Distributed Artificial Intelligence*, pages 357–366. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Solotorevsky, G. and Gudes, E. (1996). Algorithms for Solving Distributed Constraint Satisfaction Problems (DCSPs). In *International Conference on Automated Planning and Scheduling/Artificial Intelligence Planning Systems*, pages 191–198.

Stone, P. and Veloso, M. (2000). Multiagent Systems: A Survey from a Machine Learning Perspective. *Autonomous Robots*, 8:345–383.

Stranders, R., Tran-Tranh, L., Fave, F., Rogers, A., and Jennings, N. (2012). DCOPs and Bandits: Exploration and Exploitation in Decentralised Coordination. In *Proceedings of the 11th International Conference on Autonomous Agents and Multi Agent Systems*, Valencia, Spain.

Sultanik, E. A., Lass, R. N., and Regli, W. C. (2007). DCOPolis: A Framework for Simulating and Deploying Distributed Constraint Optimization Algorithms. In *Proceedings of the 9th Intl Workshop on Distributed Constraint Reasoning (CP-DCR'07)*.

Taylor, M. E., Jain, M., Jin, Y., Yooko, M., and Tambe, M. (2010). When should there be a "me" in "team"? distributed multi-agent optimization under uncertainty. In *Proceedings of the 9th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'10)*, Toronto, Canada.

Teacy, W. T. L., Farinelli, A., Grabham, N. J., Padhy, P., Rogers, A., and Jennings, N. J. (2006). Max-Sum Decentralised Coordination for Sensor Systems. In AAM (2006), pages 1697–1698.

Toth, P. and Vigo, D., editors (2001). *The Vehicle Routing Problem*. SIAM.

Vinyals, M., Rodriguez-Aguilar, J. A., and Cerquides, J. (2010). A Survey on Sensor Networks from a Multiagent Perspective. *The Computer Journal*, 53.

Wellman, M. P. (1996). Market-oriented Programming: Some Early Lessons. In Clearwater, S., editor, *Market-Based Control: A Paradigm for Distributed Resource Allocation*. World Scientific, River Edge, NJ.

Yeoh, W., Felner, A., and Koenig, S. (2006). BnB-ADOPT: An Asynchronous Branch-and-Bound DCOP Algorithm. In AAM (2006), pages 591–598.

Yeoh, W., Felner, A., and Koenig, S. (2010). BnB-ADOPT: An Asynchronous Branch-and-Bound DCOP Algorithm. *Journal of Artificial Intelligence Research (JAIR)*, 38:85–133.

Yokoo, M. (1993). Constraint Relaxation in Distributed Constraint Satisfaction Problems. In *Tools with Artificial Intelligence, 1993. TAI '93. Proceedings., Fifth International Conference on*, pages 56 –63.

Yokoo, M. (1994). Weak-commitment Search for Solving Constraint Satisfaction Problems. In *Proceedings of the 12th Conference on Artificial Intelligence (AAAI'94)*, pages 313–318, Seatle, U.S.A.

Yokoo, M. (1995). Asynchronous Weak-commitment Search for Solving Distributed Constraint Satisfaction Problems. In *Proceedings of the 1st Internation Conference*

on Principles and Practice of Constraint Programming - CP 1995, pages 88–102, Cassis, France.

Yokoo, M. and Hirayama, K. (1996). Distributed Breakout Algorithm for Solving Distributed Constraint Satisfaction Problems. In *Proceedings of the 2nd International Conference on Multiagent Systems*, ICMAS'96, pages 401–408.

Yokoo, M. and Hirayama, K. (1998). Distributed Constraint Satisfaction Algorithm for Complex Local Problems. In *Proceedings of the 4th International Conference on Multiagent Systems*, ICMAS'98, pages 372–381.

Yokoo, M. and Hirayama, K. (2000). The Effect of Nogood Learning in Distributed Constraint Satisfaction. In *International Conference on Distributed Computing Systems*, pages 169–177.

Yokoo, M., Ishida, T., Durfee, E., and Kuwabara, K. (1992). Distributed Constraint Satisfaction for Formalizing Distributed Problem Solving. In *Distributed Computing Systems, 1992., Proceedings of the 12th International Conference on*, pages 614 –621.

Zhang, W., Wang, G., Xing, Z., and Wittenburg, L. (2005). Distributed Stochastic Search and Distributed Breakout: Properties, Comparison and Applications to Constraint Optimization Problems in Sensor Networks. *Artificial Intelligence*, 161(1–2):55–87.

Zivan, R. (2005). Anytime Local Search for Distributed Constraint Optimization. In AAA (2005), pages 393–398.

Zivan, R., Glinton, R., and Sycara, K. P. (2009). Distributed Constraint Optimization for Large Teams of Mobile Sensing Agents. In *Web Intelligence*, pages 347–354.

Zivan, R. and Meisels, A. (2005). *Dynamic Ordering for Asynchronous Backtracking on DisCSPs.*

Zivan, R. and Meisels, A. (2006). Concurrent search for distributed csps. *Artificial Intelligence*, 170:440 – 461.

I am passionate about analyzing and solving real world problems. I love challenges and am not afraid of working hard to meet them.

Brammert OTTENS
Avenue de Béthusy 80
CH-1012 Lausanne
00 41 78 712 27 59
bottens@gmail.com

Working Experience

9/2007 – 8/2012: Research Assistant, AI lab, EPFL, Switzerland

As **a research assistant**, the aim of my research is to both analyze existing approaches to distributed constraint optimization (DCOP), and to come up with new distributed optimization methods. This has resulted in two distinct projects:

- Investigate the behavior of existing DCOP approaches in the presence of complex local problems. I designed a logistics problem, and evaluated a range of existing algorithms. The results showed that algorithms that query agents in a best-first manner perform the most efficient optimization. This work has been published in AAMAS.
- The second part of my research focused on the lack of a scalable, but reliable algorithm in the DCOP community. I developed a new distributed algorithm, based on a sampling approach that outperformed existing DCOP algorithms. This work has been published in AAAI.

In order to be able to perform my research, I co-developed a new platform for distributed algorithms called FRODO. It is a Java-based platform that allows easy and modular implementation of distributed algorithms, and provides a wide range of performance statistics. FRODO is currently used in multiple labs around the world, and is seen as the reference implementation for several state of the art algorithms. I also acted as **a teaching assistant** for a course on intelligent agents and supervised two student projects.

9/2003 – 7/2005: Teaching Assistant, University of Amsterdam, Netherlands

During my studies I worked as a **teaching assistant** for several courses at the University of Amsterdam. For a 3rd year bachelor course I supervised student projects. For a first year introduction to logic I devised the practical exercises, and chaired exercise session where I helped students and answered questions.

Education

10/2007 – 5/2012: PhD in Computer Science

École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland. My thesis is titled « Coordination and Sampling in Distributed Constraint Optimization »

9/2003 - 9/2007: Master of Science in Logic (Cum Laude)

University of Amsterdam, Amsterdam, Netherlands. I obtained a master in Logic, with a specialization in computational logic. My thesis topic was on the winner determination problem for Mixed Multi Unit Combinatorial Auctions (MMUCA). I analyzed the complexity of the problem, and developed a new algorithm for solving such auctions, which I implemented in prolog. The results of my research have been published at AAMAS.

9/2003 - 2/2007: Master of Science in Artificial Intelligence (Cum Laude)

University of Amsterdam, Amsterdam, Netherlands. I obtained a master in Artificial Intelligence with a thesis on the synthesis of computer vision methods and logics. I developed an axiomatization of mathematical morphology in hybrid logic, and extended an existing theorem prover, written in Haskell, to be able to deal with my axiomatization. The results of my research have been published at IJCAI. During my studies, I participated in several projects:

- For a third year project I, together with three other students, designed and implemented an agent that could operate a mobile robot, and virtually jump from one robot to another robot. My task was to write the code that allowed the agent to jump from one robot to another, through an infrared connection.
- In my fourth year, I took part in a robocup project. I was part of a team that analyzed the German team code, which we had to port to the new robot architecture. I was responsible for the vision component of the robot. We successfully ported the code, and the German code was subsequently used by the UvA team as a starting point for their own code.

Skills

Java, C, Python, Prolog, Linux, Unix, HTML, XML, Matlab, Ms Office , Keynote,

Languages

- Dutch: mother tongue
- English: C2
- French: B1
- German: B1

Voluntary Work

I have always been active outside of school. During my final years of high school, I was a member of the board of the Amsterdam chapter of **the dutch youth organization for astronomy (JWG).** I served as its treasurer, gave lectures for children aged 8-18, and helped organize summer camps. Between 2001 and 2004 I helped giving astronomy lectures for the **IMC weekend school**. This is a school for disadvantaged children, that attempts to bring them into contact with many different jobs.
While at university, I have been the treasurer of a student association, and during my PhD I have been active in the **Graduate Student Association** for Computer Science students at EPFL. I acted as president in 2010-2011, and acted as student representative during the search for a new dean of the I&C faculty.

Hobbies

I love everything to do with outdoor sports. I am an avid hiker, and try to make long treks in the mountains with my friends every year. I greatly enjoy skiing and snowboarding. My real passion, however, is long distance running. I've run several half marathons and the Amsterdam marathon.
I'm also a culture consumer. I love to read, listen to music, go to the theater and the cinema. However, I'm not afraid of standing in the spotlights myself, and have participated in several small theatre productions.

Personal Publications

Léauté, T., Ottens, B., and Faltings, B. (2010). 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)*.

Léauté, T., Ottens, B., and Szymanek, R. (2009). 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.

Ottens, B., Dimitrakakis, C., and Faltings, B. (2012). Duct: An upper confidence bound approach to distributed constraint optimization problems. In *Proceedings of the 26th National Conference on Artificial Intelligence*, AAAI'12, Toronto, Canada.

Ottens, B. and Faltings, B. (2008a). ASODPOP: Making Open DPOP Asynchronous. In *Proceedings of the Doctoral Programme of the 14th International Conference on Principles and Practice of Constraint Programming CP07*, Sydney Australia.

Ottens, B. and Faltings, B. (2008b). Asynchronous Open DPOP. In *Proceedings of the Tenth International Workshop on Distributed Constraint Reasoning - AAMAS08*, Estorial Portugal.

Ottens, B. and Faltings, B. (2008c). Coordinating Agent Plans Through Distributed Constraint Optimization. In *Proceedings of the ICAPS'08 Multiagent Planning Workshop (MASPLAN'08)*.

Ottens, B. and Faltings, B. (2012). Global Optimization for Multiple Agents. In *Proceedings of the 11th International Conference on Autonomous Agents and Multi Agent Systems*, AAMAS'12, Valencia, Spain.