2015

An equitable approach to solving distributed constraint optimization problems

Graham Billiau
University of Wollongong

An Equitable Approach to Solving Distributed Constraint Optimization Problems

A thesis submitted in fulfillment of the
requirements for the award of the degree

Doctor of Philosophy

from

UNIVERSITY OF WOLLONGONG

by

Graham Billiau

Computer Science Department
November 2015

Dedicated to

my parents

Declaration

This is to certify that the work reported in this thesis was done by the author, unless specified otherwise, and that no part of it has been submitted in a thesis to any other university or similar institution.

Graham Billiau
November 26, 2015

Abstract

We propose a new algorithm, Support Based Distributed Optimization (SBDO). SBDO is designed specifically to address the challenges unique to distributed problems. These challenges are autonomy, equality and fault tolerance. SBDO uses an argumentation based approach to solve DCOP problems. Each agent constructs a partial solution to the problem, and attempts to convince the other agents to either extend this partial solution or modify their own partial solution so that it is consistent with this one. The resulting algorithm has the following properties: Anytime, Asynchronous, Dynamic, Fault tolerant and Sound. It can solve DCOP problems which meet the following conditions:

- The domain of each variable must be finite.
- There must be a total order over solutions.
- The quality of a solution must be monotonically non-decreasing as the solution is extended.

Next we present a general model for DCOPs, called Semiring DCOP (SDCOP). SDCOP utilizes semirings to represent utility values and read/write permissions on variables to represent the distributed nature of the problem. The resulting model is capable of representing all of the accepted problems in the constraint programming domain and several problems which have not been explored by the community.

Finally we begin extending SBDO to solve problems represented using SDCOP. We start by removing the restriction that there must be a total order over the solutions. This is achieved by allowing each agent to maintain many different partial solutions simultaneously.

Future work includes extending SBDO to support problems where variables may have bounded infinite domains and problems where several agents have write access to the same variable.

Acknowledgements

I would like to thank my parents for providing me with the opportunity to study a PhD.

I would also like to thank my supervisors, Prof. Aditya Ghose, Dr Chee Fon Chang and Prof. John Fulcher, for guiding me through my PhD with their wisdom and experience.

Finally would like to thank the various members of the Decision Systems Lab, University of Wollongong, for for many insightful discussions over the years.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The Constraint Satisfaction/Optimization Problem (CSP/COP) paradigm has proven to be a very effective way of describing many real life problems. The approach is to reduce a problem to a set of decision variables and the constraints on those decisions. A popular example is the Sudoku puzzle. In Sudoku, each cell in the grid is a decision variable and there are a set of constraints which dictate what can be placed in each cell. These constraints are: only the symbols 1 to 9 may be used and all the symbols in each row, column and group must be different. Other examples include configuring a personal computer or scheduling classes at a university.

In an optimization problem, it is not enough to find any consistent solution, the best consistent solution is desired. Which solution is the best is defined by an *objective function*. In general, the form of an objective function is not well defined. In the simplest case, it takes two solutions as input and returns the better solution. The objective function itself is often confused with the value (usually an integer) returned by weighted constraints. While the objective function often uses the total value of a solution as its decision criteria, the value and the objective function are separate things.

In the simplest case, a distributed COP (DCOP) [70] is a COP where the COP must be solved while the required information is split between multiple agents[1]. By introducing agents, a lot of extra properties can be applied to the problem. These include privacy, fault tolerance, agent responsibilities, agent autonomy and equality between agents. We believe that just three of these are important for DCOPs; autonomy, equality and fault tolerance. As such we include these three properties when comparing different algorithms. The solving protocol should not impose any artificial hierarchy on the agents in the problem. Such a hierarchy gives the agents at the top more power over the final solution and requires the agents at the bottom to do more work. Each agent should also be able to act independently of the other agents. While finding a good solution requires co-ordination between the agents, an agents local environment can change at any time, possibly requiring the agent to react immediately. Finally, as the protocol is distributed over many agents, there is a much higher risk that problems will occur. The protocol must be designed in such a way that it can recover from such problems quickly.

For an example, consider a supply chain optimisation problem. There are three companies Acme, Crawen and TriOptimum are co-operating to build a faster than light spaceship. While they are working towards the same goal, and must co-operate to achive their goal, none of the companies actually trust the others. Due to the lack of trust, none of the companies will reveal any more information to the others than is required for co-operation. Acme is prepared to reveal the specifications of a part used in the spaceship and the rate at which they can be produced, but are not prepared to reveal

[1] Any software which is rational, proactive and social.

how they are produced or what other demand for this part exists. Building this spacehip is only a part of each companies business. Each company requires that they can still change the details of their own operations without consulting with their partners, particularly if such changes don't impact the current solution. The lack of trust also means that no company will accept a solution which gives another company significantly more power than it has. If one company has significantly more power than the others, it can force a solution which benefits itself over the other companies. The other companies obviously won't accept a procedure which allows one company to have that much power. ADOPT and DPOP both exhibit this power imbalance potential, as the root agent can choose the order the search space is explored in (ADOPT) or explictly gets to choose the final solution (DPOP).

The primary reason for distributing a COP is to maintain the autonomy of the agents in the problem. By solving the problem in a distributed manner, each agent can decide how it will model its sub-problem, which (centralized) algorithm to use to solve its sub-problem and which information to share with the other agents. Another benefit of the agent retaining control of its sub-problem is it also retains control of any insights that are gained regarding its sub-problem.

The concept of equality includes many facets. Here we discuss power, resource usage, knowledge and opportunity, though they are not the only ones. Power refers to how much control each agent has over the final solution. This control can come indirectly, by being able to influence the actions of other agents (often represented by the agents place in the overall ordering). Control can also come directly, by being able to specify parts of the solution (often also linked to the agent's place in the ordering). An agent may still have more power than another by virtue of controlling more variables in the problem. Resource usage refers to the resources that the agent must use while solving the problem. The most obvious examples are processing cycles, memory and communication bandwidth. This is particularly important for battery powered agents, as a higher demand on one agent means that agent will fail before the others. If resource usage is the only reason for using a DCOP, it is more efficient to parallelize a centralized algorithm over the different agents in the problem, rather than using a truly distributed system. Knowledge refers to what the agents know about the problem itself and the other agents. Much effort has been invested in maintaining privacy, that is, minimizing the knowledge each agent gains in the process of solving the DCOP. There are many reasons why maintaining privacy is desirable. An obvious case is when the COP encodes commercially sensitive information. Also, if an agent knows significantly more than the other agents, it has more incentive and opportunity to 'cheat' by not following the DCOP protocol faithfully. Many people believe that privacy is the sole reason for using a DCOP approach. If this is the only reason, then cryptographic approaches can provide better privacy guarantees. Finally, opportunity refers to when an agent can act or not act, and what actions it can take. While the requirements of the DCOP protocol will greatly limit the actions are available to an agent, each agent should not be unduly constrained by the actions of other agents. This principle is often described as asynchronicity, and the constraints often manifest as one agent waiting for a message from another agent before it can act.

Many causes, both deliberate and accidental, can cause the DCOP procedure to not be executed correctly. Fault tolerance refers to how quickly the procedure 'breaks' when these faults occur. It is not expected that the procedure is always executed correctly when faults occur, but it should not be the case that the entire procedure can not terminate because a message is not delivered. Possible faults include an agent failing, messages not being delivered, messages being corrupted and additional messages being delivered.

Due to the versatility and power of the CSP paradigm, it is often used for solving real world problems. One of the challenges with real world problems is they are often not static. Whatever causes the change to the real world, the COP must change to reflect the new reality. Often these changes are small incremental changes, so it is a waste to throw away all the effort used to solve the original problem. Dynamic COPs (DynCOPs) and Dynamic DCOPs (DynDCOPs) [66, 67] were created to model this change over time. The corresponding algorithms follow one of two basic paradigms, reactive and proactive. Reactive algorithms wait until the problem changes, then attempt to repair the previous solution to get a solution to the new problem. Proactive algorithms attempt to guess what the next problem will be and find a solution ahead of time. If a proactive algorithm is tuned correctly, it may be able to find a solution which will be correct for several changes to the problem and be able to deploy a new solution very quickly. On the other hand, if it guesses incorrectly, it is reduced to solution repair like reactive algorithms.

This thesis makes the following contributions to the state of the art:

1. We have developed a novel algorithm for solving Distributed Dynamic Constraint Optimisation Problems, Support Based Distributed Optimisation (SBDO). SBDO is the first algorithm suitable for use in unreliable environments. SBDO maintains soundness when agents are unreliable. We show how SBDO can be modified to solve multi-objective problems.

2. We describe a unification framework which can describe all of the different types of Constraint Optimisation Problems. This includes explicit support for problem classes which the constraint programming community has not considered yet.

This thesis starts with an overview of the other influential algorithms for solving Distributed Constraint Optimisation Problems. Chapter 3 describes the main contribution of this thesis, the Support Based Distributed Optimisation (SBDO) algorithm. This algorithm has been developed to solve Dynamic Distributed Constraint Optimisation Problems while satisfying the properties identified here. Next, chapter 4 shows the performance of the SBDO algorithm compared against the other influential algorithms. After the evaluation, we describe some initial work deploying SBDO to solve some real world problems in chapter 5.

The second part of this thesis covers some ways SBDO can be extended to solve a greater range of problems. Chapter 6 formally describes a new framework, SDCOP, to characterise different classes of DCOP problems. Then chapter 7 extends the SBDO algorithm to be able to solve most problem classes which can be described in SDCOP.

In the next chapter, we will briefly introduce the current research in this area. We start with various different approaches to DCOPs, and then cover some of the influential algorithms for solving them.

Chapter 2

Literature Review

In this chapter we briefly introduce the relevant existing work in the area of constraint optimization. We start with a formal definition of a CSP and it's common variations, COP, DCOP and DynDCOP. We then go into further detail on some of the common constraint types, in particular c-semiring constraints, which generalize most of the constraint types. Finally we briefly describe the competing algorithms. There have been many COP algorithms proposed, so we limit our discussion to the most influential DCOP algorithm based on each approach. These are: Asynchronous Distributed Optimization (best/depth first search), Distributed Gibbs (sampling), Distributed Pseudotree Optimization Procedure (dynamic programming), Dynamic Constraint Optimization Ant Algorithm (ant colony optimization), max-sum (generalized distributive law) and Maximum Gain Message (local search). We also describe the Support Based Distributed Search algorithm, as it is the existing algorithm that best satisfies our desired properties (autonomy, equality and fault tolerance). It has also strongly influenced the development of SBDO.

2.1 Background

2.1.1 Problem Types

Definition 1: *A Constraint Satisfaction Problem (CSP) [30, 41] is a tuple* $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ *where* $\mathcal{X}$ *is a set* $\{x_1, \ldots, x_n\}$ *of variables,* $\mathcal{D}$ *is a set* $\{d_1, \ldots, d_n\}$ *of sets of values,* $\mathcal{C}$ *is a set* $\{c_1, \ldots, c_m\}$ *of constraints defined over* $\mathcal{X}$*.*

A **constraint** *is a function* $c : (d_i \times \cdots \times d_j) \longrightarrow \{True, False\}$ *where* $\{d_i, \ldots d_j\} \subseteq \mathcal{D}$*. True if the combination of values is allowed and False otherwise.*

Given a CSP $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$, let $\sigma : \mathcal{C} \longrightarrow 2^{\mathcal{X}}$ return the signature of a constraint, that is the input variables of the constraint. A solution to a CSP is a tuple consisting of an assignment to every variable such that none of the constraints are violated.

For an example of a CSP, we consider an instance of a graph colouring problem. A graph colouring problem is to assign each node in a graph a colour such that neighbouring nodes do not have the same colour.

$$\mathcal{X} = \{\alpha, \beta, \gamma, \delta, \epsilon\}$$

$$\forall i \in \{1, \ldots, |\mathcal{X}|\}, d_i = \{\mathrm{R}, \mathrm{G}, \mathrm{B}\}$$

$$\mathcal{C} = \{\alpha \neq \beta, \beta \neq \gamma, \gamma \neq \alpha, \alpha \neq \delta, \delta \neq \epsilon\}$$

And one of the solutions to this problem is:

$$(\alpha = \mathrm{R}, \beta = \mathrm{B}, \gamma = \mathrm{G}, \delta = \mathrm{G}, \epsilon = \mathrm{B})$$

Another common problem class is meeting scheduling, in which each variable represents the interval of time at which a person will attend a meeting. All variables associated with one person must not overlap, as a person can not be in two meetings at once. All variables associated with a meeting must have the same value, as everyone must be there at the same time.

The CSP formulation is not expressive enough to represent many of the problems encountered in the real world. Problems which it can not represent include problems with a relaxed definition of constraint satisfaction or problems with non-functional requirements. In order to represent these problems constraints are modified to weighted constraints and an objective function is added to a CSP to form a COP. A weighted constraint is a generalization of a constraint which returns a real number rather than boolean value. The objective function is used to order the solutions of the problem, typically minimize or maximize the sum of the weighted constraints.

Definition 2: *A* **Constraint Optimization Problem** *(COP) is a tuple* $\langle \mathcal{X}, \mathcal{D}, \mathcal{R} \rangle$ *where* $\mathcal{X}$ *is a set* $\{x_1, \ldots, x_n\}$ *of variables,* $\mathcal{D}$ *is a set* $\{d_1, \ldots, d_n\}$ *of sets of values, and* $\mathcal{R}$ *is a set* $\{r_1, \ldots, r_o\}$ *of weighted constraints defined over* $\mathcal{X}$*.*

A **weighted constraint** *is a function* $o : (d_i \times \cdots \times d_j) \longrightarrow \Re$. Given a COP $\langle \mathcal{X}, \mathcal{D}, \mathcal{R} \rangle$, let $\sigma : \mathcal{R} \longrightarrow 2^{\mathcal{X}}$ return the signature of a weighted constraint, that is the input variables of the weighted constraint.

A solution to a COP is a tuple consisting of an assignment to every variable such that the specified objective function is satisficed. Typically minimize or maximize the sum of all the constraints.

CSPs have also been modified to support distributed problems [70]. In this case the information required to solve the CSP is distributed between many agents and it is undesirable to centralize the information. There are many reasons why it may be undesirable to centralize the information. As discussed in the introduction, the most important is the autonomy of the agents, though equality and fault tolerance are also important.

Definition 3: *[Distributed Constraint Optimisation Problem] A* **Distributed Constraint Optimization Problem** *(DCOP) is a tuple* $\langle \mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{R} \rangle$ *where* $\mathcal{A}$ *is a set* $\{a_1, \ldots, a_p\}$ *of agents, each of which controls a distinct subset of the variables.* $\mathcal{X}$ *is a set* $\{x_1, \ldots, x_n\}$ *of variables,* $\mathcal{D}$ *is a set* $\{d_1, \ldots, d_n\}$ *of sets of values, and* $\mathcal{R}$ *is a set* $\{r_1, \ldots, r_o\}$ *of weighted constraints defined over* $\mathcal{X}$*.*

A solution to a DCOP is a tuple consisting of an assignment to every variable such that the specified objective function is satisficed. Typically minimize or maximize the sum of all the constraints. If an agent controls a variable, then the agent has both the permission and responsibility to assign a value to that variable.

The last extension of CSPs that is directly relevant to this thesis is the idea of dynamic CSPs [66, 67]. In this extension researchers are attempting to solve CSP problems where the problem changes over time. Obviously, each time the problem changes a new problem can be constructed and solved. Doing so is highly inefficient, as all the accumulated knowledge about the problem is lost. It is also not sufficient if the problem changes faster than it can be solved. Current work on this problem focuses on repairing the current solution when the problem changes. This allows intermediate knowledge generated by the solver to be reused.

Definition 4: *[Dynamic Distributed Constraint Optimisation Problem] A* **Dynamic Distributed Constraint Optimization Problem** *(DynDCOP) is a sequence* $\langle DCOP_1, \ldots, DCOP_n \rangle$

A solution to a DynDCOP is a sequence $\langle s_1, \ldots, s_n \rangle$ where s_n is a solution to $DCOP_n$. In order for the problem solver to reuse information from a previous state of the problem when solving the current state, at least some of the problem must remain unchanged. If there are no similarities between one state of the problem and the next state then there is no advantage to using a solver designed for dynamic problems.

2.1.2 Constraint Types

The previous section describes the various forms of a CSP which are defined by changes to the core CSP definition. In this section we describe some of the different types of problems which can be represented by changing the form of the constraints.

Set based constraints [55] change the requirement that a single value is selected for each variable. Instead the goal is to find the smallest subset of each variables domain that satisfies the constraints. In these problems each variable is assigned a set of values which are all consistent. A common example is temporal CSPs, which are concerned with finding the temporal relations that hold between events {before, after, during, ...} [3]. Temporal CSPs can also be used for scheduling problems, where the temporal relations between tasks are known and a specific order to perform the tasks in is needed.

In some problems, the entire CSP can not be specified ahead of time. Uncertain CSPs describe the case where the constraints are not fully known. A probability is associated with each constraint in the problem, representing the chance that the constraint is part of the actual problem. The goal is to find a solution to the uncertain CSP that is most likely to be a solution to the actual CSP. Another example is open CSPs, where the domain of each variable is not known when the problem is defined. Instead the domain of each variable is elicited as the problem is solved.

In asymmetrical COPs [21] the reward for each agent involved in the constraint is different. This does not matter for utilitarian settings, as only the total reward is important. In other settings, such as egalitarian or competitive settings, the reward each agent receives is important. Asymmetrical rewards are modelled by having constraints return a tuple of values, one for each agent involved in the constraint.

C-semiring constraints [9] generalize most of the other types of constraints, so we cover them in greater detail. A c-semiring is a mathematical structure which combines a set of values with an aggregation operator and a comparison operator. The advantage of using a framework such as c-semirings is that an algorithm defined with respect to c-semirings can be instantiated to solve any of the classes of problems which can be described by c-semirings. This means that one algorithm can solve problems using classical constraints, weighted constraints or set based constraints.

Definition 5: *[9] A c-semiring is a tuple* $\mathcal{V} = \langle V, \oplus, \otimes, \bot, \top \rangle$ *satisfying (for all* $\alpha \in V$*):*

- V *is a set of abstract values with* $\bot, \top \in V$*.*
- $\oplus$ *is defined over possibly infinite sets as follows:*
  - $\forall v \in \mathcal{V}, \oplus(\{v\}) = v$
  - $\oplus(\emptyset) = \bot and \oplus(\mathcal{V}) = \top$
  - $\oplus(\bigcup v_i, i \in S) = \oplus(\{\oplus(v_i), i \in S\})$ *for all sets of indices* S

- $\otimes$ *is a commutative*[1]*, associative*[2] *and closed*[3] *binary operator on* V *with* $\top$ *as unit element*[4] *and* $\bot$ *as absorbing element*[5]*.*
- $\otimes$ *distributes over* $\oplus$[6]*.*

In a c-semiring, V is any set of symbols, the relationship between them is defined by the $\oplus$ and $\otimes$ operators. Depending on the usage, the symbols may have a meaning outside of the c-semiring. $\oplus$ is the comparison operator, given any set of values it returns the best one. When there are multiple values which are equal or not comparable, which one is returned is not specified. $\otimes$ is the aggregation operator, within c-semirings this returns a value which is equal to or worse than both of its operands. $\top$ is the 'best' value in V and $\bot$ is the 'worst' value.

Using the c-semiring framework leads to several useful properties. The $\oplus$ operator defines a partial order over V. This ordering is if $\alpha \oplus \beta = \alpha$ then $\alpha \prec \beta$. The $\oplus$ and $\otimes$ operators are monotone. In the case of $\oplus$, the result will always be the same or better as more values are compared. Similarly, the value returned by $\otimes$ will always be the same or worse as more values are aggregated. The combination of the $\oplus$ and $\otimes$ operators form a complete lattice [12] over V. Though for this discussion, the most useful property is that it generalizes many different constraint types. Specifically: classical (boolean), fuzzy, probabilistic, weighted and set constraints.

While the c-semiring framework generalizes most of the constraint types, it does not support problems with a maximization objective. Specifically, problems with weighted constraints where each constraint returns a positive value and the objective is to maximize the sum of the constraints. For these problems, as more values are aggregated, the result improves with respect to the ordering defined by $\oplus$. While c-semirings require that the opposite occurs.

2.2 Recent Work

To provide context for the Support Based Distributed Optimization (SBDO) algorithm presented here, we briefly describe some of the other DCOP algorithms which have been published. As SBDO is a general purpose DCOP algorithm we deliberately limit our discussion to other general purpose DCOP algorithms. There are many other specialized DCOP algorithms and general purpose COP algorithms worthy of note, but this discussion will not consider them. In addition there are some DCSP algorithms which also consider the problem of agent autonomy [72, 74, 73].

All of these algorithms are executed by a set of agents. For our purposes, an agent is a piece of software that is pro-active, social and rational. Pro-active means that the agent has goals and is capable of taking actions to achieve its goals. Social means that it is capable of communicating with other agents using a shared language (communication protocol). Rational means that the agent can reason about the expected consequences of its actions and will choose the actions with the most desirable outcome. Agents may have additional capabilities, for instance they may be situated in an environment and have sensors and actuators to perceive and change the environment. The agents do not need to be homogeneous, so long as they share a common language (communication protocol)

[1] $\alpha \otimes \beta = \beta \otimes \alpha$
[2] $\alpha \otimes (\beta \otimes \gamma) = (\alpha \otimes \beta) \otimes \gamma$
[3] $(\alpha \otimes \beta) \in V$
[4] $\alpha \otimes \top = \alpha$
[5] $\alpha \otimes \bot = \bot$
[6] $\alpha \otimes (\beta \oplus \gamma) = (\alpha \otimes \beta) \oplus (\alpha \otimes \gamma)$

they can communicate, even if they are implemented differently. Strictly speaking, agents may even be human, though for this work we assume all agents are software agents.

2.2.1 Asynchronous Distributed Optimization

Asynchronous Distributed Optimization (ADOPT) [39] is the first complete algorithm for solving DCOP problems. It requires linear memory on each agent and requires an exponential number of messages. ADOPT is a straightforward adaptation of a best first search algorithm to a distributed setting with additional work to increase the efficiency of recomputing previously explored nodes in the search space.

There have been many modifications and improvements to the ADOPT algorithm, these include precomputing tighter bounds for the agents, adding support for nogoods and using a depth first instead of best first search strategy [2, 22, 58, 59, 60, 61, 69].

The ADOPT algorithm relies on the agents being arranged in a depth first search (DFS) tree [27, 33]. Due to the algorithm using a best first search strategy, agents higher in the tree must assign a value to their variable before agents lower in the tree. This leads to agents higher in the tree having more authority than agents lower in the tree, which is necessary for the ADOPT algorithm but leads to problems in some settings. A side effect of this is the higher authority agents change their value less often than the lower authority agents. In an environment such as meeting scheduling where no agent has any previous authority over any other, the authority imposed by the DFS tree is often undesirable. To create a DFS tree, each agent is mapped to a node in the tree and each constraint is a potential edge (n-ary constraints are first decomposed to binary constraints). Then edges are selected such that the graph is connected, does not contain loops, and constraints only exist between an agent and its ancestors or descendants. This ensures that the problems described by different subtrees of the DFS tree are independent problems, hence can be solved independently.

ADOPT uses a modified branch and bound strategy that is based on the lower bound instead of the upper bound. It only requires local knowledge to calculate the lower bound, compared to global knowledge to calculate the upper bound. This allows agents to operate asynchronously. The search strategy is optimistic, in that an agent will change its value whenever there is a possibility that its new value will lead to a better solution. Another effect of the branch and bound strategy is that the algorithm can only function as a minimization algorithm. This is not a serious problem as any maximization problem can be transformed into a minimization problem by multiplying all of the utility values by -1. ADOPT also inherits several other properties from general branch and bound strategies, the operator used to aggregate utilities must be associative, commutative and monotonic. Monotonicity means that the total utility can only increase as more utilities are aggregated.

This approach means that an agent may abandon the current assignment to its own variable to explore a more promising assignment. If its new value does not lead to a better solution it must then rebuild the previous solution. A backtrack threshold is used to make this process more efficient. It is the best known lower bound for the agent and its sub-tree. When reconstructing a previous solution an agent will not change its value so long as the current cost is less than its backtrack threshold. This stops agents from exploring partial solutions that are known to be sub-optimal.

Due to ADOPT being a branch and bound algorithm, each agent can easily detect when the optimal solution for its subtree has been found by watching the bound interval (the difference between the

upper and lower bounds). When the interval reaches zero the optimal solution has been found and the agent can safely pause. Only the root node can detect when the optimal solution has been found as calculating the upper bound requires global knowledge. Once it has found that the optimal solution has been achieved it informs its children. The message then cascades down the tree and all the agents terminate.

The termination mechanism leads to a method for bounded error termination. Instead of only terminating when the bound interval equals zero, the algorithm terminates when the bound interval is less than some user specified value. This allows the algorithm to find a solution that is guaranteed to be within a certain distance of the optimum. Finding such a solution is often significantly faster than finding the optimal solution.

ADOPT uses the following messages in the process of finding the solution

- VALUE messages are sent from an agent to its descendants which share a constraint with this agent. They inform the children what value the agent has assigned to its variable.
- COST messages are sent from an agent to its parent. They inform the parent of the upper and lower bounds that the agent has calculated, as well as the context in which they were calculated. The context is the assignments to all of the agent's ancestors.
- THRESHOLD messages are sent from an agent to its children. They contain the child's backtrack threshold that has been allocated by the agent.
- TERMINATE messages are sent from an agent to its children. They inform the child that an acceptable solution has been found and so it may terminate.

Figure 2.1 are the details of a sample execution running ADOPT on the second example problem. This is only one of the many ways that the execution could have occurred. It also omits some messages that are not essential to the example.

1. All agents randomly take on the value α. They then send VALUE messages to their neighbours lower in the tree.
2. Upon receiving VALUE messages each agent can now calculate its costs. None of the agents have yet received cost messages so the lower bounds are assumed to be 0 and the upper bounds are assumed to be infinity. Agent 2 calculates $LB(\alpha) = \delta(\alpha) + lb(\alpha, a3) = 1 + 0 = 1$, $LB(\beta) = \delta(\beta) + lb(\beta, a3) = 5 + 0 = 5$ and $LB(\gamma) = \delta(\gamma) + lb(\gamma, a3) = 7 + 0 = 7$. From this it is clear that the lowest utility comes from selecting α, so $LB = LB(\alpha)$. Similarly it calculates its upper bound $UB(\alpha) = \delta(\alpha) + ub(\alpha, a3) = 1 + \infty = \infty$, $UB(\beta) = \delta(\beta) + ub(\beta, a3) = 5 + \infty = \infty$ and $UB(\gamma) = \delta(\gamma) + ub(\gamma, a3) = 7 + \infty = \infty$. So agent 2 sends a cost message $[lb = 1, ub = \infty, context = (a1 = \alpha)]$.

   In the same manner agent 3 calculates and sends the COST message $[lb = 2, ub = \infty, context = (a1 = \alpha, a2 = \alpha)]$ to agent 2. Agent 4 calculates and sends the COST message $[lb = 1, ub = \infty, context = (a1 = \alpha)]$ to agent 1. Agent 5 calculates and sends the COST message $[lb = 1, ub = \infty, context = (a4 = \alpha)]$ to agent 4.
3. After agent 1 has received either of the COST messages sent to it, it recalculates its lower bound and finds that it is greater than its threshold (which is still 0). This causes it to attempts to

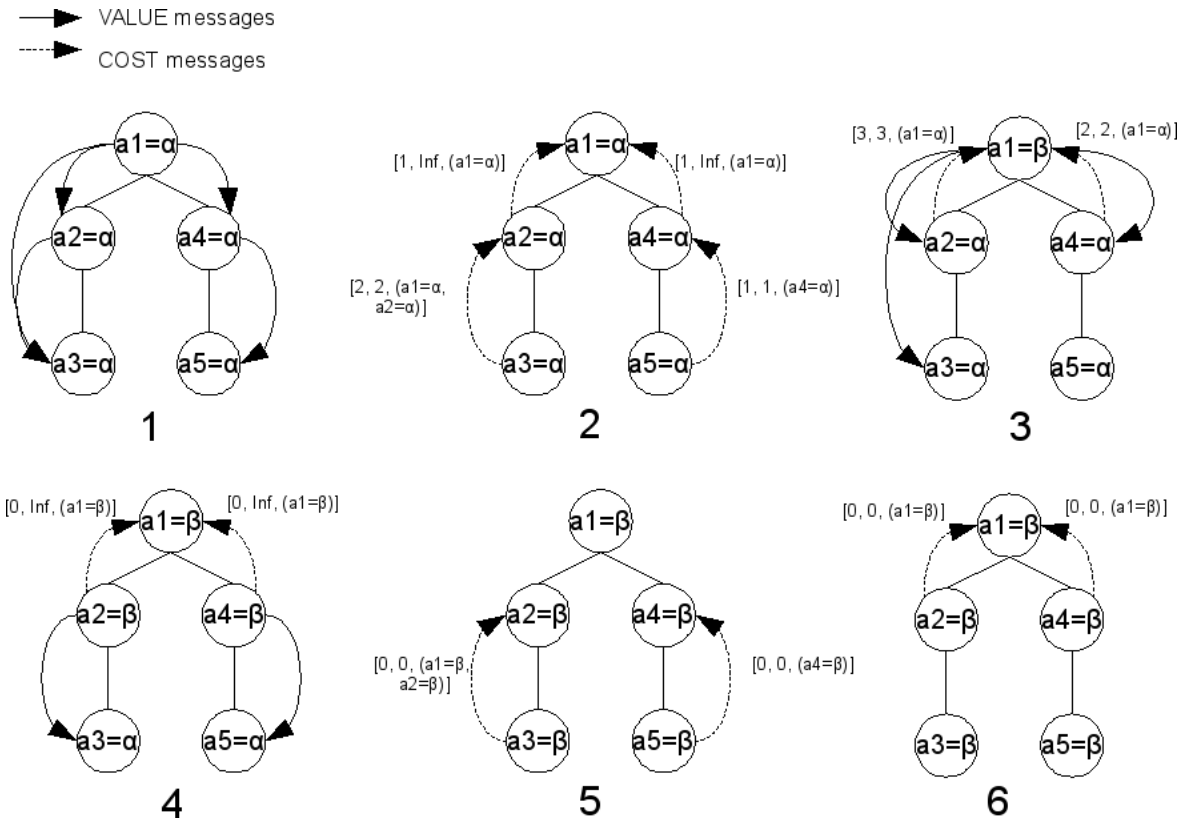

Figure 2.1: Messages passed in each step of the ADOPT algorithm

change its own value. Following the above procedure it finds that β is the best value for itself. It then sends VALUE messages advertising its change.

Agents 2 and 4 update their own cost estimates according to the COST messages sent by agents 3 and 5. Their new lower bounds are also greater than their threshold, so they attempt to find a better value for themselves. Neither of them has a better value, so they keep their value and then send THRESHOLD messages to their children. They also send the updated cost estimates to agent 1.

4. Agent 1 receives the updated cost estimates but discards them as their context is not compatible with its current value.

   Agents 2 and 4 receive the new VALUE message from agent 1 and recalculate the best value for themselves, which is β. They then send out COST and VALUE messages to communicate this change. They also reset the thresholds allocated to their children to zero and send THRESHOLD messages communicating that change.

5. Agents 3 and 5 receive the VALUE messages and also recalculate the best value for themselves, which is again β. They then send COST messages with both the lower and upper bounds equal to 0.

6. Agents 2 and 4 receive the updated COST messages, recalculate their lower and upper bounds and send the updated values to agent 1.

7. Agent 1 receives the updated COST messages and finds that $LB = UB = threshold$ and so the optimal solution has been found. It then sends TERMINATE messages.

8. The TERMINATE messages propagate down the tree until all agents have stopped.

In the worst case ADOPT has an exponential time complexity in the order of the number of variables in the problem. Also, in the worst case, space complexity is polynomial in regard to the number of variables in the problem. The space complexity can be reduced to linear at the cost of more processing at each step.

The specific DFS tree picked can have a large impact on the performance of the algorithm. The worst arrangement is a chain where each agent has at most one child. This forces the algorithm to treat it as one problem and does not allow sub-problems to be solved independently. It also means that agents high in the tree spend a lot of time idle. The ideal tree is a DFS tree with the largest possible branching factor. This decomposes the problem into lots of sub-problems which can be solved independently.

Some preprocessing techniques have been applied to significantly reduce the time required to find the solution [2]. These techniques do not reduce the search space, instead they attempt to focus the search by computing approximate utility bounds for each assignment. The more accurate bounds reduce the number of times ADOPT explores a non-optimal solution, and hence the number of times it has to recompute a previously explored solution.

2.2.2 Distributed Gibbs

The Distributed Gibbs (DGibbs) algorithm [43] is an adaption of the Gibbs sampling algorithm [20] to solve DCOPs. The Gibbs sampling algorithm is designed to solve Maximum Likelihood Estimation (MLE) problems in Markov random fields. The maximum likelihood estimation problem is equivalent to solving a COP problem with a maximization objective. Specifically, valued constraints in a COP are equivalent to probability functions in MLE problems. The DGibbs algorithm requires that the agents are first organized in a pseudo-tree, this is a Depth First Search tree where edges (constraints) which are not part of the DFS tree are retained as pseudo-edges.

In each iteration of the algorithm, a value is selected for each of the variables in the problem and the utility is compared with the previous best solution. Processing starts at the root agent, which selects (using the Gibbs sampling strategy) a value for its own variables. The root agent then informs all of its neighbours which assignments it has made and the marginal utility due to those assignments using a VALUE message. Once an agent has received VALUE messages from all of its (pseudo-)parents, it selects values for all its own variables (using the Gibbs sampling strategy). The agent then informs all of its neighbours which assignments it has made and the marginal utility gained due to those assignments using a VALUE message. If the total marginal utility gained is better than the previous best, the agent assumes that this set of assignments is part of a better solution than the previous solution. In this case it then remembers its current assignments as the best so far.

Once a leaf agent has assigned a value to it's variables it sends a BACKTRACK message to its parents. The BACKTRACK message includes the agents marginal utility information, so that if the children of an agent have discovered a better solution that information is passed up the tree. Once an agent has received BACKTRACK messages from all its children, it sends a BACKTRACK message

to its parents. Finally, once the root node has received a BACKTRACK message from all its children, one iteration of the algorithm has completed.

This is not sufficient to ensure that what each agent believes is the best solution so far is correct, as it will not have been informed about a good partial solution found in a different subtree. To communicate this information, VALUE messages indicate which iteration the best solution found so far was found in. If the best solution was found in the previous iteration, then it must have been found in a different subtree. The agent then knows to mark its current value (before selecting new values for this iteration) as part of the best solution found so far.

The Gibbs sampling strategy is guaranteed to converge to the true values. Given the true probability values, the MLE technique is sound, as such DGibbs is sound and complete. The algorithm is anytime, as it stores the best solution found so far.

2.2.3 Distributed Pseudotree Optimization Procedure

Distributed Pseudotree-Optimization Procedure (DPOP) [47] is based on a dynamic programming approach. Dynamic programming exploits two properties found in some problems to produce more efficient algorithms. These properties are optimal substructure and overlapping subproblems. For a problem to have the optimal substructure property it must be possible to decompose the problem into subproblems, solve them optimally, then combine the solutions to get the optimal solution. COPs do not have this property, however in DPOP they emulate this property by using a different definition of a subproblem. In their definition the subproblem includes fixed assignments for the variables outside the subproblem which have constraints with variables inside the subproblem. For a problem to have the overlapping subproblems property the same subproblem must be used many times to solve a larger problem. COPs do have this property. Dynamic programming increases efficiency by saving and reusing the solution to each subproblem, rather than calculating it every time it is needed.

DPOP also relies on ordering the agents into a modified DFS tree [27, 33], the definition is extended to include pseudo-edges and pseudo-parents. A pseudo-edge is any constraint that is not an edge in the DFS tree. A pseudo-parent is any agent higher in the tree that the agent is connected to via a pseudo-edge. This decomposes the problem into a series of subproblems that can be solved in a bottom up approach.

DPOP has a very simple structure due to its dynamic programming approach. It assumes that every agent knows the domain of its neighbours. Processing starts at the leaf agents. Each agent constructs an n-dimensional matrix, with one dimension for each parent and pseudo-parent of the agent. Each cell in the matrix stores the optimal utility for the agent and its children, given the assignments to this agents parents. It then sends the matrix to its parent. It does not need to consider any constraints between it and its children as they will have been included in the messages it receives from its children.

All other agents wait until they have received a matrix from all of their children. They then combine the utilities given by their children with their own utilities to create their own matrix and their optimal assignments.

When the root node has received a matrix from all of its children, it can combine them to get the globally optimal utilities for each value it might assign its own variables. Using this knowledge, it picks the optimal value for itself. It then communicates this value to each of its children.

Once an agent has received a value message from all its parents it can look up its previously calculated table to find the optimal assignment for its own variables. It then assigns those values to its variables and communicates this to all its children. That agent can then terminate.

As explained above DPOP uses two messages

- UTIL messages contain the matrix showing the utilities for an agent. They are sent from an agent to its parent.
- VALUE messages contain the assignment an agent has chosen for its variable as well the assignments to other variables that were in the UTIL message received from that child. They are sent from an agent to its children and not its pseudo-children.

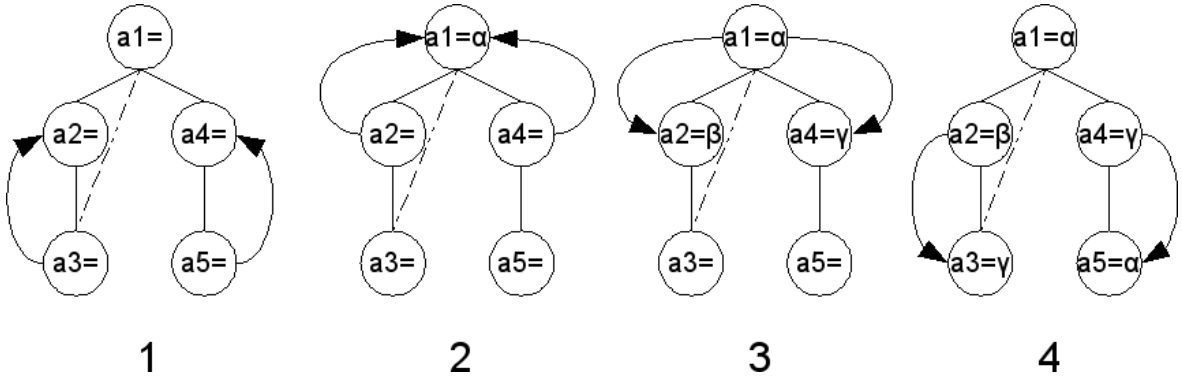

Figure 2.2: Messages passed in each step of the DPOP algorithm

Figure 2.2 shows the passing of messages and the tree structure when running DPOP on the second example problem. In the first two steps the UTIL messages propagate up the tree, the agent 1 can select its value and the VALUE messages propagate down the tree in the remaining two steps.

1. Agent 3 and agent 5 are the leaf nodes, so they send the first messages. As agent 5 has no pseudo-parents, the UTIL message it sends has only one dimension while agent 3 has one pseudo-parent so its UTIL message has 2 dimensions. These messages are depicted in Table 2.1 .

(a) Message sent from agent 5 to agent 4

agent 4's value	α	β	γ
utility	7	6	7

(b) Message sent from agent 3 to agent 2

	α	β	γ
α	14	13	11
β	13	12	12
γ	11	12	14

Table 2.1: The messages sent in the first step of the DPOP algorithm

2. When agent 2 and agent 4 receive these messages they can calculate the utility for themselves and then send their own UTIL messages. They are depicted in Table 2.2 .
3. Agent 1 then combines the UTIL messages it has received and discovers that assigning a value of α or γ to its variable leads to the optimal utility of 32. It then arbitrarily decides to take the value α and sends VALUE messages to its children.

(a) Message sent from agent 4 to agent 1

agent 1's value	α	β	γ
utility	14	13	14

(b) Message sent from agent 2 to agent 1

agent 1's value	α	β	γ
utility	18	18	18

Table 2.2: The messages sent in the second step of the DPOP algorithm

4. When agent 2 receives the VALUE message it looks up its internal table to find what the optimal assignment to its value is, which is β. Similarly agent 4 takes on the value γ. They then send VALUE messages to their children. The message sent by agent 2 to agent 3 also includes the value of agent 1, as it was referenced in the UTIL message agent 2 received from agent 3.

5. Finally agent 3 and agent 5 receive the VALUE messages and look up the optimal values for themselves, which are γ and α respectively.

All of the complexity of this algorithm lies in the size of the messages it must communicate. The number of messages sent is linear, as it only requires $2n$ messages where n is the number of edges in the DFS tree. However the size of the messages is exponential, with the largest message size being dom^w where dom is the size of the domain and w is the maximum number of parents and pseudo-parents of any agent.

The size of the messages sent between agents is not constant. Messages sent by agents with the most pseudo-parents are the largest. In an attempt to keep the size of messages to a manageable level an extension to this algorithm has been developed that identifies the areas which require message sizes that are too large at run time. It then uses a local search algorithm to find a solution for these areas while still using DPOP for the rest of the problem [49].

This is process is controlled by a parameter called $maxDims$ which represents the maximum number of dimensions allowed in a matrix. When an agent would normally be required to send a matrix which is larger than this limit it removes the highest agents from the matrix until it is equal to $maxDims$ and records that these agents have been removed from the matrix. When an agent receives a matrix from one of its children which does not include a dimension for its own variables, it knows that it must use local search for that part of the problem. It may still receive complete matrices from other children, in which case it knows that subtree has been solved optimally just as in normal DPOP.

If an agent has to use local search it will not send a UTIL message to it's parent until it has explored the assignments to itself through local search. This ensures that higher agents do not calculate their own utilities using incorrect information.

Because no local search algorithm can guarantee that it will find the optimal solution this extension no longer guarantees that DPOP will find the optimal solution. In practise this is not a serious problem. It allows bigger problems to be solved and still produces very good quality solutions, as most of the problem is solved optimally.

2.2.4 Dynamic Constraint Optimization Ant Algorithm

The Dynamic Constraint Optimization Ant Algorithm (DynCOAA) [35, 36, 37, 38] uses Ant Colony Optimization (ACO) [15] to solve DCOP problems. In order to solve DCOP problems, they must first be transformed into the type of problems which ACO can solve, namely graph search problems. For

ease of description we describe the transformation into a set of graphs, though the solver treats them as a single graph.

The first graph, referred to as the agent ordering graph, determines the agent ordering. It is a fully connected graph where each node represents an agent in the problem. This graph connects the sub-problems controlled by each agent and is used by the ants to decide on the best agent ordering. Each node may be visited by each ant only once, and once the ant visits the node it must then traverse the agent's variable assignment graph.

The variable assignment graph is a labeled directed multi-graph with two types of nodes, variable nodes and transition nodes. For each variable controlled by the agent there is one variable node and for each pair of variables there are two transition nodes, one in each direction. There is one edge from a variable node to each of the transition nodes leaving that variable. There are many edges from the transition node to the destination variable node, each one labeled with a different value from the source variable's domain.

As per standard ACO techniques, a swarm of ants are spawned at one node in the graph (specifically in the agent ordering graph) and must find a route to the destination (also in the agent ordering graph). When an ant reaches a node in the agent ordering graph it must traverse that agent's variable assignment graph. By selecting an edge from a variable node (the source variable) to a transition node, the ant is deciding on a variable ordering. By selecting an edge from a transition node to a variable node (the destination variable), the ant is assigning a value to the source variable. The agent picks which edge to traverse next stochasticly. It combines the weight of the edge and the utility (if any) gained by selecting the edge and uses the combined value for a weighted random selection. An ant will not pick a variable assignment that violates any of the constraints, similarly it will not visit a variable or an agent it has visited before. If the ant can not pick a valid value for a variable, then it dies.

Once all the ants have reached the destination, or died, there is a pheromone update step. First the weights on all edges are reduced logarithmicly, this reduces the chance that the edges will be selected in the future. Afterward, the weights on the edges traversed by the best ant in the swarm and the best ant overall are increased by a constant value, so they are more likely to be chosen in the future. This step requires global communication between all the nodes, so will not scale well.

The graphs can easily be extended or contracted when the problem changes, with any new edges given an average weight. Better results can be achieved by normalizing the weights of all edges after a change to the problem. This allows the algorithm can easily adapt to changes in the problem. Soundness is ensured by killing ants when they are unable to find a constant assignment to any variable.

2.2.5 Max-Sum

The max-sum algorithm [63] is a distributed implementation of the Generalized Distributive Law [1]. This approach exploits the distributive law to reduce the total number of operations required to compute the solution. In general, max-sum requires a linear amount of memory at each agent and an exponential amount of messages.

In order to use the max-sum algorithm to solve a DCOP, it must be transformed into a factor graph. Nodes in a factor graph are either functions (constraints) or variables. Edges may only exist

between a function node and a variable node. An edge means that the variable is an input to the function.

Exactly one agent in the problem is given responsibility for each node in the graph (both functions and variables). One agent often controls many nodes.

Solving proceeds simultaneously on all agents. For variable nodes the agent iteratively calculates the following equation and sends the result to the nodes neighbours.

$$Q_{n\to m}(x_n) = \alpha_{nm} + \sum_{m' \in M(n)m} Rm' \to n(x_n) \tag{2.1}$$

Where α_{nm} is chosen such that

$$\sum_{x_n} Q_{n\to m}(x_n) = 0 \tag{2.2}$$

For function nodes the agent iteratively calculates the following equation.

$$R_{m\to n}(x_n) = \max_{x_m\, n} \left(U_m(x_n) + \sum_{n' \in N(m)n} Q_{n'\to m}(x_{n'}) \right) \tag{2.3}$$

In general the graph may contain cycles, so there is a normalizing component (αnm) to the equations. Another effect of the cycles in the graph is the system may is not guaranteed to converge to a solution. This is unusual, as generally max-sum still finds a good solution to the problem.

The max-sum algorithm is complete when applied to acyclic graphs. To take advantage of this, later work does not solve the original problem directly, instead a minimal change to the problem is computed to create a non-cyclic graph. This is done by giving a weight to each edge in the graph and then computing a minimal spanning tree [16, 53]. By solving the problem given by the minimal spanning tree it guarantees that the algorithm will converge and that it will be optimal with respect to the modified problem. It is also possible to calculate the bounds on the solution quality by comparing the original problem with the modified problem.

From this we can see that the max-sum algorithm does have the desired properties. There is no artificial hierarchy imposed on the agents, so the equality of the agents is maintained. This also makes it easy to update the factor graph when the problem changes. The ability to handle dynamic problems is hindered by using the minimal spanning tree rather than the original factor graph. Finally, Max-Sum does not assume that messages will always arrive. The authors have shown that the algorithm is still able to converge to a solution even when 90% of messages are not delivered [63]. No results have been presented regarding other failure types.

Further work on this algorithm has covered both performance improvements and applying it to real world domains [18, 51].

2.2.6 Maximum Gain Message

Maximum Gain Message (MGM) [31] is a simple distributed hill climbing algorithm. It is an extension to the Distributed Breakout Algorithm (DBA) to support problems with weighted constraints. MGM is a hill climbing algorithm, so it is incomplete, however it is anytime and supports dynamic problems.

At the start of the algorithm, each agent randomly selects a value for its own variables. The algorithm then proceeds in a series of rounds, each round consists of two phases. In the first phase, agents inform their neighbours what their current assignments are using a VALUE message (assuming

they have changed). Each agent can then calculate the optimum change to its own assignments. In the second phase, agents inform their neighbours how much utility they gain by changing their assignments using a GAIN message. If the utility gain of an agent is larger than all of its neighbours (ties are broken using an arbitrary tie breaking scheme), the agent changes its assignments. The algorithm terminates when no agent can improve its own utility.

The synchronization via GAIN messages ensure that two neighbouring agents do not change value simultaneously, but does still allows parallelism by allowing unconnected agents to change value simultaneously. Due to this synchronization and the hill climbing approach, the solution quality is monotonically increasing. None of the agents store any history, so the algorithm easily supports dynamic problems.

As a local search algorithm, MGM does not have any overarching hierarchy, so it maintains the equality property we desire. The strict two phase structure of the algorithm reduces the autonomy of the agents, Depending on the specific implementation, each agent may have to send a value message every round. Unless a timeout scheme is implemented, then the agents can not know if their neighbour intends to take part in the current round or not. This can lead to a deadlock while agents wait for a failed agent. Due to this we do not consider this algorithm autonomous or fault tolerant.

2.2.7 Support Based Distributed Satisfaction

Support based distributed search (SBDS) [24, 25] is a local search algorithm inspired by argumentation. Unlike the other algorithms presented in this section, SBDS is a DCSP algorithm. It is included because of the strong influence it has had on the SBDO algorithm presented in this thesis.

In SBDS all of the agents have equal standing. They attempt to find a solution by arguing with each other. This involves all agents trying to get other agents to change their value by sending successively stronger and stronger arguments. Each agent continues sending arguments until either that neighbour changes it's value to be consistent, or this agent is convinced to change its own value. Two messages are used in the arguing process.

- isgood messages tell other agents what value this agent has taken on and why.
- nogood messages tell other agents that a partial solution can not be part of a complete solution.

To arrive at a solution all agents first pick an arbitrary value for themselves. They then send isgoods to their neighbours to tell them what value they have taken on. Next each agent checks if its value is consistent with the values it has received from all of its neighbours. If it isn't, then it picks the neighbour with the strongest argument as its support. It then chooses a new value for itself that is consistent with its support and sends out new, stronger isgoods to its neighbours. All agents repeat this step until there are no inconsistent assignments. If at any time an agent finds that there is no consistent assignment for itself it sends a nogood.

Cycles of oscillating values can occur because all agents act asynchronously. This occurs when a group of variables oscillate between one set of values and another without settling on one. To prevent this, SBDS it imposes a total order on the domain of possible isgoods. The highest ranking isgood is always chosen as the new support. This, along with the constantly increasing size of isgoods ensures that any cycle will eventually settle into one state.

From	To	message	
α	β, γ, δ	$\langle(\alpha, \mathrm{R})\rangle$	Agents α and ϵ announce their values, other agents could also announce their values.
ϵ	δ	$\langle(\epsilon, \mathrm{G})\rangle$	
β	α, γ	$\langle(\beta, \mathrm{B})\rangle$	Agents β and γ pick α as their support and agent δ picks ϵ as it's support. They then announce the values they have chosen.
γ	α, β	$\langle(\gamma, \mathrm{G})\rangle$	
δ	α, ϵ	$\langle(\delta, \mathrm{R})\rangle$	
δ	ϵ	$\langle(\epsilon, \mathrm{G}), (\delta, \mathrm{R})\rangle$	As agent α is inconsistent with agent δ, δ sends a stronger isgood to α.
α	β, γ, δ	$\langle(\delta, \mathrm{R}), (\alpha, \mathrm{G})\rangle$	Agent α is then convinced to change its value and so selects δ as support.
γ	α, β	$\langle(\alpha, \mathrm{G}), (\gamma, \mathrm{B})\rangle$	Agent γ is then inconsistent and so changes its value.
β	α, γ	$\langle(\alpha, \mathrm{G}), (\beta, \mathrm{B})\rangle$	
γ	α, β	$\langle(\delta, \mathrm{R}), (\alpha, \mathrm{G}), (\gamma, \mathrm{B})\rangle$	The values of agents β and γ are then inconsistent so they attempt to convince the other to change.
β	α, γ	$\langle(\delta, \mathrm{R}), (\alpha, \mathrm{G}), (\beta, \mathrm{B})\rangle$	
γ	α, β	$\langle(\delta, \mathrm{R}), (\alpha, \mathrm{G}), (\beta, \mathrm{B}), (\gamma, \mathrm{R})\rangle$	Finally agent β convinces γ to change its value and the solution is consistent.

Table 2.3: Messages sent in a sample run of SBDS

Algorithm	Time Complexity	Space Complexity	Properties
ADOPT	Exponential	Polynomial	Bounded error, Complete, Sound
DGibbs	Exponential	Linear	Anytime, Complete, Sound
DPOP	Polynomial	Exponential	Complete, Dynamic, Sound
DynCOAA	Exponential	Linear	Anytime, Dynamic, Global communication, Sound
Max-Sum	Exponential	Linear	Anytime, Autonomy, Bounded error, Dynamic, Equality, Fault tolerant, Multi-objective
MGM	Exponential	Linear	Anytime, Dynamic, Equality
SBDS	Exponential	Exponential	Anytime, Autonomy, Equality, Sound

Table 2.4: Summary of existing algorithms

This results in an iterative improvement algorithm that begins by exploring many partial solutions and slowly expanding these solutions until one complete solution is found.

2.2.8 Comparison

In this section we present an overview of the previously described approaches used to solve DCOPs. The differences between the algorithms are summarized in Table 2.4.When reading the table, it is important to note that the complexity represents the worst case. The average case is much more representative of the algorithm's performance, but in general is only shown through empirical results.

The complexity of each algorithm is only part of the story, the properties of the algorithm are also important. In some cases a complete algorithm is required, though in general proving completeness is very expensive. An algorithm which can be stopped when the solution quality is good enough can be sufficient. Similarly if the problem is dynamic or has multiple objectives, then an algorithm with those properties is required.

We chose to base SBDO on SBDS because SBDS already has the autonomy and equality properties which we are looking for, even though SBDS is only a satisfaction algorithm. Further, modifying SBDS

to add other desired properties is easier than modifying other algorithms to add autonomy or equality, as both of those properties reflect the core of the algorithm.

2.3 Conclusion

In this chapter we have provided a brief overview of the constraint optimization domain, focusing on DCOPs. First we presented formal definitions of a CSP and related problems, including distributed and dynamic variations. Next we covered the different forms of constraints that can be used with those problems. We focused on the c-semiring framework, as it provides an overarching framework to capture most of the different types of constraints used in constraint optimization.

We then showed a representative sample of the algorithms which have been developed to solve DCOPs. Only DCOP algorithms are considered as they are directly relevant to this thesis. From our analysis it is clear that none of the previous algorithms satisfy the properties we have identified as being important in a DCOP system (autonomy, equality and fault tolerance). The algorithm which best satisfies these properties is SBDS, a DCSP algorithm.

The c-semiring framework is insufficient to describe all the constraint types used in recent COPs, so before we describe the SBDO algorithm we first describe a more general framework, the Semiring Distributed Constraint Optimization Problem (SDCOP) to use as the theoretical basis for SBDO.

Chapter 3

Support Based Distributed Optimization

3.1 Introduction

As described in the introduction to this thesis, we consider the important properties of a DCOP solving algorithm to be autonomy, equality and fault tolerance. These are in addition to the standard properties of termination, soundness and completeness. Of the algorithms mentioned in the literature review, Max-Sum and DynCOAA come closest to meeting these ideals. Both Max-Sum and DynCOAA give autonomy and equality to the agents, in addition, Max-Sum is fault tolerant. Specifically, Max-Sum retains the termination and soundness properties when messages may not be delivered. However, neither of them are complete, not being able to return all optimal solutions at completion. On the other hand, Adopt and DPOP are complete, but they do not have equality or fault tolerance.

In order to attempt to meet these ideals, the Support Based Distributed Optimization (SBDO) algorithm presented in this chapter is based on negotiation/argumentation between the agents, rather than a power structure as commonly used. Because of this, the agents have equal power over the other agents and the final solution to the problem. In the process of solving the problem, the agents dynamically form coalitions and use their combined power to influence the other agents. These coalitions are determined based on the structure of the problem and random chance, rather than any bias in the algorithm. This equality means that the algorithm is not unduly affected should an agent fail, and can continue attempting to solve the problem without the failed agent. The redundancy built into the message passing means that an agent which fails can be recovered quickly. The communication protocol used is also not particularly sensitive to messages arriving in the wrong order, which makes the algorithm tolerant of a variety of different faults.

The version of SBDO described in this chapter can solve DCOP problems which meet the following conditions:

- The domain of variables ($\mathcal{D}$) must be finite.
- There must be a total order over solutions, i.e. $\forall a, b \in V, a \prec b \vee b \prec a$.
- The quality of a solution must be monotonically non-decreasing as the solution is extended, i.e. $\forall a, b \in V, a \otimes b \preceq a$.

In addition, the algorithm can only solve dynamic problems in a reactive manner. SSBDO, described in chapter 7, removes the requirement for a total order over the solutions.

The rest of this chapter is structured as follows: In section 3 we explain the core ideas behind SBDO and show how they work together. Next, sections 3.3.1 to 3.3.3 describe the algorithm. Then

in section 3.4 we explain the properties of the algorithm and present the relevant proofs. Finally section 3.6 sums up our results and outlines future research directions. The empirical evaluation of this algorithm is presented in chapter 4.

Parts of this chapter have previously been published in conference proceedings [8, 6].

3.2 Support Based Distributed Optimization

SBDO is a DynDCOP algorithm which uses a novel approach inspired by argumentation, it is complete with respect to hard constraints though not complete with respect to objectives. Each agent sends 'arguments' to its neighbours in an attempt to convince the other agents to adopt its proposed partial solution of the problem. When an agent accepts the solution proposed by another agent it uses that solution as the basis for its own partial solution. This adds strength to the agents argument, making other agents more likely to accept its proposed partial solution. Accepting another agent's solution also makes the other agent the parent of this agent in the agent ordering. This leads to a dynamic agent ordering and completely asynchronous processing, which makes SBDO a very efficient solver.

Before describing the SBDO algorithm itself we first define the concepts used. This algorithm can only solve a limited subset of the problems which can be represented by the SDCOP framework introduced in chapter 6. The constraints are divided into hard (classic) constraints ($\mathcal{C}$) and soft (weighted) constraints ($\mathcal{R}$) which we refer to as objectives. The definition of a DCOP used in this chapter is defined as follows:

Definition 6: *A* **Distributed Constraint Optimization Problem** *(DCOP) is a tuple* $\langle \mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{R} \rangle$ *where* $\mathcal{A}$ *is a set* $\{a_1, \ldots, a_p\}$ *of agents, each of which controls a subset of the variables.* $\mathcal{X}$ *is a set* $\{x_1, \ldots, x_n\}$ *of variables,* $\mathcal{D}$ *is a set* $\{d_1, \ldots, d_n\}$ *of sets of values,* $\mathcal{C}$ *is a set* $\{c_1, \ldots, c_m\}$ *of constraints defined over* $\mathcal{X}$*, and* $\mathcal{R}$ *is a set* $\{r_1, \ldots, r_o\}$ *of weighted constraints defined over* $\mathcal{X}$*.*

Note that the set of variables each agent controls does not have to be distinct. Thus SBDO supports some of the read write privileges defined in SDCOP.

The neighbourhood graph (Figure 3.1) represents the communication channels between agents. Each node in the graph represents an agent and there is an edge between two nodes if the respective agents must communicate. Two agents must communicate if they each control a variable which contributes to the same constraint or objective. To minimize communication bandwidth and privacy loss, the communication channels between agents exist only in situations where the agents share variables. More formally,

Definition 7: *Given a SDCOP* $= \langle \mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{V}, \mathcal{C} \rangle$*, a* **neighbourhood graph** *is a directed graph* $\langle N, E \rangle$ *where* $N = \mathcal{A}$ *and* $E \subseteq \mathcal{A} \times \mathcal{A}$ *such that* $\forall \alpha, \alpha' \in \mathcal{A}, \langle \alpha, \alpha' \rangle \in E$ *iff* $\alpha \neq \alpha' \wedge \exists x \in \mathcal{X}$ *such that* $x \in W_\alpha \wedge x \in R_{\alpha'}$*.*

If an agent α has write privileges to a variable x and another agent α' has read privileges to x then they must be able to communicate, as α must inform α' of any changes it makes to x. If two agents must communicate then they are in the same neighbourhood.

For example, consider a DCOP with five agents A, B, C, D and E, each of which controls one variable a, b, c, d and e respectively. The constraints are $AllDiff(a, b, c), a > b, c \neq d$ and the objectives are $min(a + e), min(c - d)$. This will produce the neighbourhood graph shown in Figure 3.2.

During each SBDO agent's processing cycle, the agent can communicate with any of its neighbours.

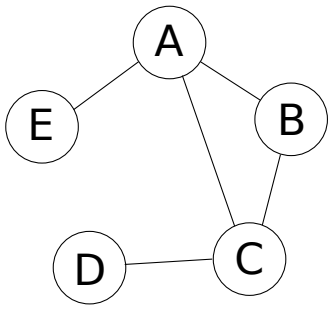

Figure 3.1: Example neighbourhood graph.

This is unlike other algorithms which define a hierarchy over the agents. In those algorithms it may be the case that the agent must wait for communication from one of its children, or alternatively the agent may not be able to communicate directly with an agent it shares a constraint with. By allowing free communication between agents the 'reaction speed' of the algorithm can be improved and the entire algorithm does not deadlock when one agent fails.

Each processing cycle proceeds as follows: The agent first processes all messages it has received from other agents and updates its local knowledge appropriately. Then the agent computes a new, locally optimal, solution to the part of the problem it can see. This solution is created by extending one of the partial solutions it has received from its neighbours with assignments to its own variables. Finally the agent tells all of its neighbours what its new proposal is, in an attempt to convince them to chose a value that is consistent with this new proposal.

Definition 8: *Given a DCOP* $\langle \mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{R} \rangle$*, an* **assignment** *is a triple* $\langle a, V, u \rangle$ *where* $a \in \mathcal{A}$*,* V *is a set of variable-value pairs and* u *is the utility of this assignment. The variable-value pairs indicate the values currently assigned to a subset of the variables which are controlled by* a*. The utility of this assignment is the sum of the value returned by all of* a*'s local objective functions, given the current assignments to* a*'s variables.*

The function $f_u : assignment \longrightarrow \mathbb{R}$ *returns the utility of the assignment.*

The choice of which variable assignments to reveal to the other agents in the problem is a trade off between the privacy concerns of each agent, the correctness of the algorithm and the redundancy in the algorithm. For the algorithm to be correct, the assignments to variables which are used in shared constraints or shared objectives must be revealed to all of the agents neighbours. As these variables are part of the co-ordination between agents they will have to have to be revealed to some other agents anyway. Due to this revealing them to all neighbours (from which they might propagate to all agents) is a small breach of privacy. All other variables can be kept private, however if they are not revealed to other agents, they do not benefit from the redundancy within SBDO. This means that if the agent fails it will have to recompute the values for those variables, while the values assigned to revealed variables can be recovered from the agents neighbours.

Definition 9: *Given a DCOP* $\langle \mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{R} \rangle$*, an* **evaluation** *is a tuple* $\langle r, V, u \rangle$ *where* $r \in \mathcal{R}$*,* $V = \{\langle x_1, d_1 \rangle, \ldots, \langle x_n, d_n \rangle\}$ *is a set of variable-value pairs such that* $\{x_1, \ldots x_n\} = \sigma(r)$ *and* u *is the utility returned by the objective function* r *given the assignments* v*.*

The function $f_u : evaluation \longrightarrow \mathbb{R}$ *returns the utility of the evaluation.*

It is assumed that in practice the actual objective function is not shared (due to privacy concerns),

only the unique identifier for the objective function is shared. This makes it necessary to also include the output of the objective function.

Definition 10: *Given a DCOP* $= \langle \mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{R} \rangle$ *and its associated neighbourhood graph* N*, a* **proposal** *is a pair* $\langle VA, SOA \rangle$*, where* VA *(variable assignments) is a sequence* $\langle assignment_1, \ldots, assignment_n \rangle$ *of assignments such that the sequence of agents is a simple path through* N*.* SOA *(shared objective assignments) is a set of evaluations.*

The sequence of assignments is the order in which this proposal has been constructed, as each agent appends their own assignment to the proposal. The utility returned by shared objectives must be stored separately to each agents local utility to prevent double counting.

For example, consider the two agents A and B, who share an objective r, as well as having their own (unspecified) local objectives. When A first creates a proposal it simply contains an assignment to A, $\langle \langle \langle A, \{\langle a, 1\rangle\}, 3\rangle\rangle, \{\}\rangle$. Later when B extends the proposal, B then has enough information to evaluate the shared objective, producing $\langle \langle \langle A, \{\langle a, 1\rangle\}, 3\rangle, \langle B, \{\langle b, 2\rangle\}, 2\rangle\rangle, \{\langle r, 10\rangle\}\rangle$. A then receives the extended proposal from B and chooses to change its value, in the process updating the shared objective and producing $\langle \langle \langle B, \{\langle b, 2\rangle\}, 2\rangle, \langle A, \{\langle a, 3\rangle\}, 2\rangle\rangle, \{\langle r, 15\rangle\}\rangle$. Note that if the shared objective was not stored separately it would be double counted, once by B and once by A.

For our purposes, a nogood with justification (originally defined in [56]) is treated as follows: **Definition 11:** *Given a DCOP* $\langle \mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{R} \rangle$*, a* **nogood** *is a pair* $\langle V, C \rangle$ *where* V *is a set of variable-value pairs forming a partial assignment and* $C \subseteq \mathcal{C}$ *is the set of constraints that provides the* **justification** *for the nogood, such that* V *violates at least one constraint in* C*. Each variable can appear at most once in* V*. A nogood* $\langle V, C \rangle$ *is a* **minimal nogood** *iff there does not exist a a set* $V' \subset V$ *such that* $\langle V', C \rangle$ *is a valid nogood and there does not exist a set* $C' \subset C$ *such that* $\langle V, C' \rangle$ *is a valid nogood.*

For example, given a problem with three variables, a, b and c, all of which have a domain of {0, 1} and constraints $a = b, b \neq c$, the nogood $\langle \{\langle a, 0\rangle, \langle c, 0\rangle\}, \{a = b, b \neq c\}\rangle$ is a minimal nogood. $\langle \{\langle a, 1\rangle, \langle b, 1\rangle, \langle c, 1\rangle\}, \{a = b, b \neq c\}\rangle$ is also a valid nogood, but is not minimal. If the constraint $b \neq c$ is replaced with $b < c$, then $\langle \{\langle c, 0\rangle\}, \{b < c\}\rangle$ is not a valid nogood. It is only inconsistent with reference to b's domain, so it must include the implicit constraints on b's domain. The minimal nogood is $\langle \{\langle c, 0\rangle\}, \{b < c, b \geq 0\}\rangle$.

Definition 12: *Given a proposal* $\langle VA, SOA \rangle$*, the* **total utility** *is* $\sum_{a \in VA} f_u(a) + \sum_{e \in SOA} f_u(e)$

The total utility of a proposal can be considered as the overall quality of the solution that the proposal represents. Now that the utility of a proposal is defined, we can define an ordering over the proposals. Whenever we refer to one proposal being better than another in this paper it is with respect to this ordering. We assume that all agents have the same deterministic function to implement consistent random choice. This will ensure that the ordering is the same for all agents.

Definition 13: *The ordering over proposals* ($\prec$) *is defined by a lexicographical order on the following criteria:*

1. *Highest total utility.*
2. *Highest number of assignments.*
3. *Consistent random choice with respect to the partial solution the proposal represents*[1]*. That is, if an agent randomly chooses the partial solution A over the partial solution B, it will choose A*

[1] *Hash functions can provide a suitable comparison.*

over B in all future comparisons, even if the order of the assignments has changed. Further, the choice must be consistent among all agents in the problem.

Consider four different proposals:

1. $\langle\langle\langle A, \{\langle a, 0\rangle\}, 10\rangle\rangle, \{\}\rangle$
2. $\langle\langle\langle A, \{\langle a, 1\rangle\}, 8\rangle\rangle, \{\}\rangle$
3. $\langle\langle\langle A, \{\langle a, 2\rangle\}, 5\rangle, \langle B, \{\langle b, 0\rangle\}, 3\rangle\rangle, \{\}\rangle$
4. $\langle\langle\langle A, \{\langle a, 3\rangle\}, 4\rangle, \langle B, \{\langle b, 1\rangle\}, 4\rangle\rangle, \{\}\rangle$

Proposal 1 is preferred over all the others because it has the greatest total utility. Proposals 3 and 4 are preferred over proposal 2 because they have more assignments and the total utility is the same. As proposals 3 and 4 have the same total utility and the same number of assignments one is picked arbitrarily.

The ordering over proposals can be changed to suit the specific problem being solved, such as modifying it to be able to solve minimization problems. The ordering must be defined such that it is a total ordering and as more assignments are added to a proposal the preference of the proposal is monotonically increasing. If the preference is not monotonically increasing, agents will not be able to convince their neighbours to accept them as their support. When the preference of the proposal decreases then other agents will prefer to do nothing, rather than extend the proposal by accepting the agent as its support. It must be a total order otherwise the algorithm may get stuck in a cycle of oscillating values. Both of these properties are required for the algorithm to terminate.

Each agent contains the following data structures:

- support. The agent that this agent is using as the basis for almost all decisions it makes. The support's beliefs about the world (its view) are considered to be facts.
- view. This is a proposal consisting of the proposal received from support with an assignment to this agents variables appended. This represents the agent's current beliefs about the world, or its world view.
- recv. This is a mapping from an agent A to the most recent proposal received from that agent.
- nogoods. This is a store of all current nogoods received. It contains pairs $\langle \text{sender}, \text{nogood} \rangle$.
- sent(A). This is a mapping from an agent A to the most recent proposal sent to that agent.
- sent-nogoods. This is a multi-set of all nogoods sent by this agent. It contains pairs of $\langle \text{destination}, \text{nogood} \rangle$.
- objectives. A set of all objectives this agent knows. It must include all of an agent's local objectives and all objectives this agent shares with other agents.
- constraints. A set of all constraints this agent knows. It must include all of an agent's local constraints and all constraints this agent shares with other agents.

The following messages are required for the basic functionality of the algorithm:

- *proposal-update* $\langle I \rangle$ where I is a proposal. This message is sent from one agent to another. It contains a proposal which represents (part of) an agents current view of the problem, it is always sent by one agent to another agent. An agent that receives this message updates its record of the latest proposal received from the sending agent in recv.

- *new-nogood* $\langle n \rangle$ where n is a nogood. This message is sent from one agent to another. It contains a nogood which indicates that a partial solution is unacceptable. An agent that receives this message adds the nogood to its nogoods.

The following messages are required for dynamic problems.

- *remove-constraint* $\langle c \rangle$ where c is a constraint. This message is sent from the environment to all agents that know the constraint which should be removed. It contains the constraint that has been removed. An agent that receives this message removes the constraint from constraints and checks to see if any nogoods it has sent are now obsolete.

- *remove-nogood* $\langle c, \{n_1, \ldots, n_i\} \rangle$ where c is a constraint and $n_1, \ldots, n_i$ are nogoods. This message is sent from an agent to another agent. It contains a constraint that has been removed and a set of nogoods that are obsolete because this constraint has been removed. An agent that receives this message removes the obsolete nogoods from nogoods and checks to see if it any nogoods it has sent are now obsolete.

- *remove-objective* $\langle r \rangle$ where r is an objective. This message is sent from the environment to all agents that know the objective which should be removed. It contains the objective that has been removed. An agent that receives this message removes the objective from objectives.

- *add-constraint* $\langle c \rangle$ where c is a constraint. This message is sent from the environment to all agents involved in the new constraint. It contains the constraint that has been added to the problem and the agent identifier and variable name for each variable in the signature of the constraint. An agent that receives this message adds the constraint to constraints.

- *add-objective* $\langle r \rangle$ where r is an objective. This message is sent from the environment to all agents involved in the new objective. It contains the objective that has been added to the problem and the agent identifier and variable name for each variable in the signature of the objective. An agent that receives this message adds the objective to objectives.

Adding new agents is handled implicitly by the add-constraint and add-objective messages. The messages contain the details for all relevant agents so if the new constraint or objective involves an agent who is not already a neighbour of this agent, the new agent must be added to this agent's neighbours. Similarly removing agents is handled implicitly by the remove-constraint and remove-objective messages. When none of an agent's constraints or objectives reference an agent who is a neighbour of this agent, the agent is removed from this agent's neighbours.

The following messages are required to cope with agent failure:

- *resend-state* $\langle \rangle$. This message is sent from one agent to another agent. It requests that an agent send all its relevant public information to the sender agent. An agent that receives this message responds by sending a proposal-update message with the agents latest proposal and a series

nogood messages containing all nogoods which were previously sent to the sender. This has the effect of restoring all the knowledge that the failed agent had before it failed.

We use the notation $I \sqsubseteq I'$ to say that I is a sub-proposal of I'.$|I|$ to denote the number of assignments in I. $scope(I)$ is the set of variables that are assigned in I. $trim(I, A)$ is the suffix of proposal I including the assignment to agent A through to the end of the proposal. If there is no assignment to A then $trim(I, A) = I$.

3.3 Algorithm

The SBDO algorithm is divided into three major sections, processing received messages, computing a new solution and sending updates to other agents. Each agent loops over these sections until it receives no new messages.

3.3.1 Processing Received Messages

The physical communication channels that agents must use to communicate are never perfect, so it is desirable for distributed algorithms to be able to tolerate messages being corrupted, arriving in random order or even never arriving. SBDO is generally robust against messages arriving in random order but not robust against message loss or message corruption. To ensure correctness when agents fail, it must be possible for an agent to know if a message from one agent was sent before or after another message from the same agent.

The messages that an agent has received are processed in a particular order. This ordering is not required for the correctness of the algorithm, it is only done for efficiency.

1. new-nogood and remove-nogood messages. Nogoods are processed first as a remove-constraint message might make them obsolete, if that happens they should be marked as obsolete immediately.

2. add-constraint, remove-constraint, add-objective and remove-objective messages. These messages, which are generally sent by the environment, are processed next. It is important that they are processed before any proposal-update messages as they may affect the consistency of the new proposals. If the proposal-update messages are processed first the algorithm might generate a nogood which is immediately rendered obsolete, which requires at least two extra messages. One to inform the other agent about the nogood and one immediately after to delete the extra nogood, which might also disrupt the other agents processing.

3. proposal-update messages. Now that the agents knowledge about its environment is up to date it can consider the new proposals it has received. Processing the new proposals last minimizes the chance that a proposal will conflict with the known state of the environment when it doesn't conflict with the actual state of the environment.

4. resend-state messages. Only once all other messages are processed are resend-state messages considered. This is to ensure that the information provided to the newly restarted agent is as up to date as possible.

```
begin
    while Not Terminated do
        for All new-nogood messages N do
            if The nogood n in N is not obsolete then
                Add n to nogoods
            end
        end
        for All remove-nogood messages R do
            for All nogoods n in R do
                if N in nogoods then
                    Delete n from R
                    Delete n from nogoods
                end
            end
            if The set of nogoods in R is not empty then
                Add R to removed-constraints
            end
        end
        for All messages received from the environment do
            Process message
        end
        for All received proposal messages I do
            Set A to the agent who sent I
            set recv(A) to I
        end
        for I ∈ recv do
            if there is no valid assignment to self wrt I then
                send_nogood(A)
            end
        end
        update_view()
        for All neighbours A do
            if self and A are consecutive assignments in view then
                if view ≺ sent(A) or sent(A) is not consistent then
                    Send view to A
                    Set sent(A) to view
                end
            else
                Set L to the minimum proposal length that satisfies the requirements
                Set P to a proposal such that |P| = L and P ⊑ view
                if P ≠ sent(A) then
                    Send P to A
                    Set sent(A) to P
                end
            end
        end
        Wait until at least one message has been received
    end
end
```

Algorithm 1: main()

```
begin
    Set C to the removed constraint
    for Each neighbour A do
        for Each nogood N sent to A do
            Set obsolete = {}
            if N contains C as part of its justification then
                Add N to obsolete
                Delete N from sent-nogoods
            end
        end
        if |obsolete| > 0 then
            Set M to a new remove-constraint message ⟨C, obsolete⟩
            Send M to A
        end
    end
    for Each received nogood N do
        if N contains C as part of its justification then
            Mark N as obsolete
        end
    end
end
```

Algorithm 2: process remove-constraint message

```
begin
    Set N to a nogood derived from recv(A)
    Set recv(A) to null
    if N not in sent-nogoods then
        Send N to A
    end
    if support = A then
        Set support to self
    end
end
```

Algorithm 3: send_nogood(A)

A nogood becomes obsolete if any of the constraints in its justification are removed from the problem. If the nogood is not minimal it is possible that it is still a valid nogood after the constraint is removed. In this case (false positive) the nogood will be reinstated by an agent in the problem, ensuring correctness. However if the nogood is invalid and not removed (false negative) a set of solutions to the new problem will not be allowed due to the nogood, which violates the correctness of the algorithm.

Adding new agents to the problem is handled implicitly by the add-objective and add-constraint messages. When one of those messages is received that creates a constraint or objective with a previously unconnected agent, the agents it is now connected to add the new agent to their set of neighbours, and vice versa. This means that the new agent has been added to the overall problem. Similarly when a remove-constraint or remove-objective message is received that removes the last link between one agent and the other agents, the other agents remove it from their set of neighbours. This means that the agent is now removed from the problem.

While the processing of most environment messages is straightforward, the procedure for removing

```
begin
    Let C be the constraint referenced
    for Each received nogood N do
        if N is in the remove-nogood message then
            Delete N from nogoods
            delete N from the remove-nogood message
        end
    end
    if counter ≠ 0 then
        Add the remove-nogood message to removed-constraints
    end
    pre-remove constraint(C)
end
```

Algorithm 4: process remove-nogood message

```
begin
    Set G to a valid assignment to all local variables, chosen greedily
    Choose support and G such that all the following hold:

      • view is recv(support) extended by G

      • for All received proposals I do
        view ≺ I or I is consistent with view
          end

end
```

Algorithm 5: update_view()

constraints is quite complicated (alg. 4, alg. 2 and alg. 1 lines 6-12). We must guarantee that all nogoods that might be obsolete because of the removal of this constraint are deleted, while allowing for messages to arrive any order. To make the same guarantees of correctness when messages can arrive in any order and also accepting that agents can fail requires another step, which we do not include in this algorithm due to the extra complexity. This extra step is is explained in the fault tolerance section (3.4.2).

When an agent receives a remove-constraint message it searches sent-nogoods (which contains the nogoods and who they were sent to) for any nogoods which have the removed constraint as part of their justification, these nogoods are obsolete (alg. 4 lines 3-8). The agent then sends a separate remove-nogood message to each of its neighbours, containing the constraint that has been removed and the nogoods sent from this agent to this neighbour that are obsolete (alg. 4 lines 9-11). After sending the remove-nogood messages this agent deletes the obsolete nogoods from sent-nogoods. This agent only marks its own nogoods as obsolete at this time, it does not delete them from nogoods (alg. 4 lines 12-14).

This complexity is required because we allow messages to arrive in any order. To illustrate this, consider the following:

1. Agent A sends a nogood N to agent B.
2. A receives a remove-constraint message which makes N obsolete.
3. A sends a remove-nogood message to B.

4. B receives the remove-nogood message.

5. B receives N.

Without some means of knowing that N is still in transit, N would not be deleted, which breaks the correctness of the algorithm.

Similarly (though less likely) this scenario can occur:

1. Agent A sends a nogood N to agent B.

2. B receives N.

3. A receives a remove-constraint message that makes N obsolete.

4. A sends a remove-nogood message to B.

5. A receives an add-constraint message that reinstates the previously removed constraint.

6. A re-sends the nogood N to agent B.

7. B receives N (again).

8. B receives the remove-nogood message.

In this scenario, without a limit on the number of nogoods deleted, both copies of N will be deleted, when only one should be deleted. While both copies of the nogood are valid, so can be kept, it is important to keep the knowledge of each agent synchronized. If both copies are deleted then A will resend the nogood, but this also leads to the nogoods A remembers sending to B and the nogoods B remembers receiving from A being different. If this information is not synchronized, then the future removal of nogoods will not be performed correctly.

When an agent receives a remove-nogood message it attempts to delete exactly the nogoods mentioned in the message from its store of known nogoods (alg. 1 lines 6-12). If there are obsolete nogoods in nogoods that are not mentioned in the remove-nogood message it does not delete them yet. Instead it waits for another remove-nogood message, as deleting them now would also make the agents knowledge of sent and received proposals out of sync with the other agents. If there are less obsolete nogoods than mentioned, they must still be in transit. In this case the remove nogood message is retained, then when the obsolete nogoods do arrive they can be deleted (ignored). Once all the nogoods mentioned in the remove-nogood message have been deleted the remove-nogood message itself can be deleted.

The agent must also check its own store of sent nogoods to see if any of its neighbours must be notified of the change. This is exactly as if it had just received a remove-constraint message.

We acknowledge that this approach to removing constraints may delete nogoods that are not obsolete, as the nogood may not be minimal, so is still valid after this constraint has been removed. The cost for a false negative is that potential solutions to the problem are erroneously rejected (breaking the correctness of the algorithm). The cost for a false positive is two extra messages for the nogood to be reinstated. Due to this we choose to eliminate the false negatives and accept the false positives.

Finally, the received proposal messages are processed. Each received proposal message replaces the proposal previously received from that agent in recv. After all new proposals have been stored, *all* currently known proposals must be checked to see if they are correct wrt the known state of the

environment. This is to ensure that both the newly received proposals and the old proposals are correct and do not violate any of the nogoods. If a proposal is not correct a nogood is created and sent to the agent that sent this proposal. This will force the sender to change their value in the next iteration. When determining the set of constraints violated the set of implicit constraints that represent the domain of each variable must also be considered. It does not have to be a minimal nogood, though if it is it greatly increases the usefulness of the nogood. If a partial solution violates more than one set of constraints two (or more) separate nogoods may be created, each with a different justification.

3.3.2 Find Solution

Now that the agent has the most recent information about its environment, it can continue with the second phase of processing. This involves choosing which of its neighbours to be its support, then using the assignments in the latest proposal received from its support to compute the best value for its own variables. This will normally require a centralized COP solver inside each agent.

The selection of a support for each agent is similar to defining an ordering over the agents. When an agent A selects another agent B as its support, A accepts B as its 'parent'. This ordering is not a total ordering, and it often contains cycles. The existence of any ordering over the variables/agents may appear to go against our desire to avoid any ordering, but some ordering is required to solve the problem. Also, as it is a dynamic ordering it does not cause the problems identified with a static ordering.

The agents new support A must match the following criteria (alg. 5):

- A = support or view $\prec$ recv(support).
- there exists an assignment v to this agent such that for all received proposals I, v is consistent with I or $trim(\mathsf{recv}) + v \prec$ I.

The criteria have deliberately been kept as loose as possible to allow for problem specific heuristics to be used. Sometimes it is not possible to satisfy the second criterion, in which case the agent should select the best assignment for itself.

3.3.3 Send Updates

Finally each agent A must communicate any changes to its neighbours (alg 1, lines 22 - 32). It goes through the following steps for each of its neighbours B: First it checks to see if A and B are part of a cycle (alg 1, line 23). If there is an assignment to A immediately followed by an assignment to B in the proposal received from support, then there is a cycle in the variable ordering. When A is in a cycle with B it skips the usual proposal creation steps, instead it checks to see if it should postpone sending a message to B (alg 1, line 24). If the proposal previously sent to B is preferred over A's view, then sending a proposal would cause the cycle to continue. Because of this A does not send a proposal to B this cycle. Otherwise A sends its entire view to B (alg 1, line 25).

As an example, suppose agents A, B and C are part of a cycle. Agent A has received the proposal $\langle\langle\langle A, \{\langle, 1\rangle\}, 4\rangle, \langle B, \{\langle b, 4\rangle\}, 3\rangle, \langle C, \{\langle c, 1\}, 6\rangle\rangle, \{\}\rangle$ from C. When A sends an proposal message to C it follows the normal construction procedure, as while C is involved in the cycle, it is not the next agent in the cycle. However when A sends an proposal message to B it must use the cycle prevention logic.

Given that A's view is now $\langle\langle\langle B, \{\langle b, 4\rangle\}, 3\rangle, \langle C, \{\langle c, 1\}, 6\rangle, \langle A, \{\langle a, 2\rangle\}, 3\rangle\rangle, \{\}\rangle$ (total utility of 12). If the previous proposal A sent to B was $\langle\langle\langle B, \{\langle b, 2\rangle\}, 5\rangle, \langle C, \{\langle c, 5\}, 1\rangle, \langle A, \{\langle a, 3\rangle\}, 3\rangle\rangle, \{\}\rangle$ (total utility of 9) then A will send its entire view in an proposal message to B. On the other hand, if the previous proposal A sent to B was $\langle\langle\langle B, \{\langle b, 1\rangle\}, 4\rangle, \langle C, \{\langle c, 1\}, 6\rangle, \langle A, \{\langle a, 1\rangle\}, 4\rangle\rangle, \{\}\rangle$ (total utility of 14) then A will not send an proposal message to B.

If A and B are not part of a cycle then processing proceeds as follows: First the length of the proposal this agent should send is worked out (alg. 1, line 28). The proposal must be long enough to meet the following criteria:

1. At least as long as the proposal previously sent to the destination agent.
2. It must contain enough assignments to evaluate shared objectives/constraints. Specifically, if there exists a constraint/objective involving at least the destination agent B and an agent C in this agents' view (which might be A), then the proposal must contain the assignment to C.
3. If the assignment to A is not consistent with the proposal received from B, then A should send a counter-proposal that is more preferred than B's. Because of the way A picks its view (should chose a view such that it is consistent with or stronger than all received proposals) A should be able to construct a preferred proposal. In the case that A can not it sends the strongest proposal possible (A's entire view).
4. If the proposal previously sent to B is not a sub-proposal of A's view, then the new proposal must be longer than the previous proposal.

Once the length of the proposal has been worked out, a proposal is constructed by taking the most recent N assignments from A's view and all relevant shared assignments. Obviously, if the length of A's view is less than the required length, the new proposal will be shorter than required. Finally, if the new proposal is different to the proposal previously sent to B, it is sent to B. If the proposal has not changed, no proposal is sent this processing cycle (alg. 1, line 30).

3.4 Discussion

If the algorithm is deployed in a static environment detecting that the network has reached a quiescent state is sufficient to detect termination. This can be achieved by taking a consistent global snapshot [11]. The algorithm will also terminate if it detects that there is no solution to the problem.

3.4.1 Dynamic Problems

Most of the changes to the problem that can occur in a dynamic system are straightforward to implement, however changes that relax the problem require more work. Several messages are required to communicate any changes to the problem to the agents: add-constraint, remove-constraint, remove-nogood, add-objective, remove-objective, add-domain and remove-domain. These messages all reflect changes to the environment and as such are referred to as environment messages. With the exception of remove-nogood they are assumed to be sent by the environment. Only the agents that control the variables involved in the objective or constraint that is added or removed must be notified.

A change to the agents involved is handled implicitly by the other messages. When an agent no longer has any links to one of its neighbours, that agent is no longer a neighbour. Once an agent has no links to any other agents it is effectively removed from the problem. Agents are added to the problem by creating a link between them and another agent. In the process they are then also a neighbour of the other agent.

Expanding the domain of an agent and removing constraints both relax the problem. Which may cause nogoods to become obsolete. These obsolete nogoods must be identified and removed from the problem, otherwise they will prevent valid solutions to the new problem from being explored.

When a constraint is removed in an update to the underlying COP all of the nogoods that were generated because of the removed constraint must also be removed. They can be identified via the nogoods justification. If the justification contains the deleted constraint then the nogood might be obsolete and must be deleted. This does mean that a nogood which violates two or more constraints, and so is still valid, may be deleted. If this occurs the nogood will be re-posted later. As it is possible for a nogood to arrive after the message that renders it obsolete, `pre-remove constraint(C)` (alg. 2) and `post-remove constraint()` (alg. 4) are required to ensure correctness.

When values are added to the domain of a variable it is more difficult to identify the nogoods that are obsolete. The agent controlling that variable must first work out what implicit constraints have been removed. Once the constraints have been identified, removing nogoods proceeds exactly the same.

In order to remove obsolete nogoods, when an agent receives a remove-constraint message it runs `pre-remove constraint(C)` (alg. 2). First it searches the set of sent nogoods and sends all of its neighbours that are affected a remove-nogood message. This message contains the constraint that has been removed and a set of the of obsolete nogoods that this agent sent to this neighbour (which have been stored in sent-nogoods). Then it deletes the obsolete nogoods from its set of sent nogoods. It is necessary to identify specific nogoods to ensure that all obsolete nogoods are deleted and to ensure newly posted nogoods (if the constraint is reinstated) are not deleted.

When an agent receives a remove-nogood message it runs `post-remove constraint()` (alg. 4). First it attempts to remove all the nogoods that have been marked as obsolete from nogoods. Any which have been removed are also deleted from the set in the remove-nogood message. If nogoods remain, they must refer to nogoods which are still in transit. The remaining nogoods are added to the agents set of known obsolete nogoods, so they can be deleted when they arrive. The agent only deletes nogoods that are mentioned in the set of known obsolete nogoods, even if there are more nogoods that include the removed constraint, and as such are obsolete. Finally it also runs `pre-remove constraint(C)` (alg. 2) in case it has sent any nogoods which were derived because of an obsolete nogood, hence are also obsolete.

To catch any nogoods that arrive after the remove-nogood message that marks them as obsolete, every time a new nogood is received it must be checked against the set of known obsolete nogoods. If the new nogood is obsolete it is ignored (effectively deleting it) and it is removed from the set of known obsolete nogoods.

When deployed in dynamic environments this algorithm will never terminate. In contrast to a static environment, detecting that the network of agents has reached a quiescent state, or that the problem is over-constrained are insufficient as terminating criteria. New inputs from the environment may relax the problem or simply force a new solution to be generated. If desired the algorithm can

be stopped by sending each agent a special terminate message.

3.4.2 Fault Tolerance

Due to the nature of the algorithm, when an agent fails it has a minimal impact on the other agents. Because there is no imposed hierarchy between the agents, each agent can send and receive any message to/from any of its neighbours, at any time. This means that unlike in other algorithms that do have an imposed hierarchy an agent can continue to communicate with its neighbours and attempt to solve the problem. This is different to the situation in which a strict hierarchy is enforced, as an agent may be unable to continue until it receives a message from the failed agent, possibly leaving the entire system in a deadlock. These changes mean that SBDO degrades gracefully as agents fail. Other agents can continue solving using the last known value for the failed agent, rather than waiting for a message that will never arrive.

When an agent fails all its knowledge regarding sent and received proposals is lost. This effectively means that messages have been lost, for which this algorithm can not account. To prevent this, when the failed agent restarts it must request that its neighbours send it all relevant information on the state of the problem. This includes the last proposal and all nogoods that they sent to this agent, as well as the last proposal and all nogoods they have received from this agent. This prevents most knowledge loss and allows the failed agent to resume solving faster. Of course, only an agents public information can be recovered this way, the agent will have to recompute its private information.

If two neighbouring agents both fail at the same time then some information is irretrievably lost. However this does not affect the correctness of the algorithm as the entire transaction between the agents are lost. So all agents knowledge is still consistent.

Unfortunately our assumption that messages may arrive in random order allows for a particular sequence of events which does leave the system in an inconsistent state.

1. Agent A sends nogood N to agent B.
2. A fails and restarts.
3. A requests update from its neighbours.
4. B sends update to A.
5. B receives N.

After this sequence of events B has a nogood N in its store that A has no record of sending. Which means that nogoods will not be correctly deleted when they become obsolete. To avoid this situation it is sufficient to ensure that agent A does not receive any messages from B that were sent before the update collection of messages. One possible solution is to add a time stamp to all sent messages, then whenever an agent receives a message it checks that the time stamp on the new message is greater than the time stamp on the last collection of messages received as a response from an update request sent to that agent. This does not require that the time is synchronized between agents, or even that the time is used, any monotonically increasing counter may be used. We have not incorporated this in the algorithm to minimize the complexity of our description of the algorithm.

The proof of termination for SBDO is based on the proof of termination for SBDS [23]

Lemma 1: *As long as the problem does not change, eventually no new nogoods will be generated.*

Proof: *Each agent keeps all nogoods it ever receives. A nogood is sent when a received proposal is found to be inconsistent, ensuring that the proposal will never be received twice from the same source. As the set of possible proposals must be finite, eventually no new nogoods will be generated.* □

Lemma 2: *If no new nogoods are generated, then eventually the utility of view will become stable for each agent.*

Proof: *Let $W_i \subseteq \mathcal{X}$ be the set of agents whose view has a utility greater than or equal to $i \in \mathcal{Z}$. We will prove that any decrease in $|W_i|$ must be preceded by an increase in $|W_j|$, where $j < i$.*

First, we note that an agent will never willingly reduce the utility of its view, as per the proposal ordering and the requirements of `update_view()`*. So, in the usual case, $|W_i|$ will be monotonically increasing, for all i. However, in limited circumstances an agent may receive a weaker proposal from its support, and so the utility of its view could be forced to decrease. Such events are rare, but they can occur whenever a cycle of supporting agents is formed. Let us assume that some agent v receives a shorter proposal I from its current support, and so v is forced to choose a shorter view. Let i be the length of v's old view, and j be the length of v's new view, respectively. The new, shorter view for v will obviously decrease each $|W_k|$, where $j < k \leq i$.*

However, for v to have received the weaker proposal I, some agent w must have formed a cycle by changing its support. Note that w will only have selected a new support if it could increase the utility of its own view, as per the proposal ordering and the requirements of `update_view()`*. Also note that the newly-formed cycle cannot have a total utility of more than j, else there would have been no reason to reduce the utility of v's view. Therefore, the utility of w's new view must then be less than or equal to j, but is certainly longer than it's old view.*

So, if an agent v is forced to reduce the utility of its view, then there must be some preceding agent w which increased the utility of its view. Further, w's new view is guaranteed to have a utility no greater than v's new view. Therefore, the term $|W_1|.|W_2|.|W_3|\ldots$. must increase lexicographically over time. As the term is bounded above, we can conclude that the utility of view must eventually become stable for each agent. □

Corollary 1: *If no new nogoods are generated, then eventually the support will become fixed for each agent.*

Definition 14: *We will say that two agents A and B are in agreement if their respective views do not contain different assignments.*

Lemma 3: *For each fully connected sub-graph of the neighbourhood graph, which is created because of an n-ary constraint, there will exist a proposal such that the proposal contains an assignment for every agent in the sub-graph.*

Proof: *In order for the lemma to be false, there must be two or more proposals, each covering a part of the sub-graph, which are stable. For the proposals to be stable they must not be propagated to the other agents in the sub-graph, or not be strong enough to defeat the other agent's view.*

Agents only restrain from propagating proposals when the agents involved are part of a cycle or the information previously sent to the agent is still correct. In both of these cases not sending the proposal does not affect the algorithm, as the agent the proposal would be sent to already has the same or better information. So for the proposals to be stable the proposals must not defeat the other agent's view.

In order for one proposal to not defeat another proposal (the other agents view), the two proposals must compare equal or extending the proposal causes the receiving agent to no longer be consistent with its neighbours. As we enforce a total ordering over the proposals they can not compare equal. There

are two situations where a received proposal can cause an agents view to no longer be consistent with its neighbours, either the new proposal contains an assignment to itself or the agent it is in conflict with. In both of these cases the agents involved are already part of the same proposal, so do not impact the stability of the proposals.

Therefore it is not possible to have two or more stable proposals which each cover a part of the sub-graph. □

Lemma 4: *The algorithm will not reach a state of 'rest' until lemma 3 is satisfied.*

Proof: *While agents continue to send messages the algorithm has not terminated. As per lemma 3, if there is more than one proposal which covers part of the sub-graph they are not stable. This means that at least one agent within the sub-graph will change its support in its next iteration. After it has changed its support it must send updates to its neighbours, hence the algorithm has not terminated. Alternatively if the support relation between all the agents in the sub-graph forms a chain, as the fully connected graph is created because of a constraint shared by all agents within the graph, the minimum proposal length specified in* `update_view()` *means that agents will continue to send longer proposals, hence the algorithm has not terminated.* □

Theorem 1: *Support-Based Distributed Optimization is sound.*

Proof: *SBDO has no explicit notion of termination if the problem is satisfiable. However, it is possible for all agents to reach a state of 'rest', which we will treat as equivalent to termination for satisfiable problems, even if there exists no way to verify that all agents are resting. We must now prove that SBDO will not terminate unless the problem is proven unsatisfiable, or all agents are in a state of agreement with no unsatisfied constraints.*

1. *If SBDO states that the problem is unsatisfiable then it must have derived the empty nogood. As nogood derivation is obviously sound, we can conclude that SBDO will only say a problem is unsatisfiable if that is true.*

2. *Assume that agent u is not in agreement with agent v on the value of some variable w (it is possible that $u = w$ or $v = w$ but is not necessary). We know that both u and v obtained their current value of w through chains of agents which intersect at w.*

   If no agent postpones the transmission of a new view then each agent will update neighbours on the current value of w. As they are sending an update, the agents will not be at 'rest' and the algorithm will not have terminated.

   If an agent in the chain postpones the transmission of a new view then it effectively causes a neighbour to be at rest until the conditions causing the postponement are cleared. However, using a total ordering on proposals ensures that not all agents participating in the same cycle will postpone simultaneously. As at least one agent must continue to propagate the 'greatest' view, the algorithm has not terminated.

3. *Assume that agent u is in conflict with agent v, and that the view held by u is stronger than or equal to that held by v. The* `update_view()` *procedure guarantees that* view $\prec recv(v)$*, further, the send proposal logic (alg. 1, lines) guarantees that $sent(v) \prec recv(v)$ and so, from the perspective of v, $recv(u) \prec view$. This condition will force v to either change* support *to u or send a stronger proposal to u, and so the algorithm has not terminated.*

4. *Assume that agent u is in conflict with v due to an n-ary constraint but they can not discover they are in conflict because no agent has enough information to evaluate the n-ary constraint. To ensure a constraint (of any arity) is evaluated it is sufficient to ensure a single proposal which contains assignments for every agent involved in the constraint is generated. As per the definition of the neighbourhood graph, all the agents involved in the constraint form a fully connected sub-graph. Lemma 3 then says that a single proposal containing assignments for all the agents in the constraint will form, and lemma 4 shows that if that is not the case then the algorithm has not terminated.*

From the above, we know that SBDO will not terminate until the problem is proven unsatisfiable, or all agents are in agreement and no set of agents are in conflict. Therefore, SBDO is sound. □

To prove completeness we can use the proof system described by Harvey (Theorem 1 [23]). We must provide a definition for a set of partial assignments I as a function of the current state of the algorithm, satisfying the following properties:

1. A solution to the constraint satisfaction problem is contained in I.
2. For each $s \in I$ where $|\hat{s}| > 1$, there exists $t \in I$ such that $t \subseteq s$ and $|\hat{t}| + 1 = |\hat{s}|$.
3. Testing $s \in I$ takes linear-time with respect to the size of the internal state of SBDO.

Theorem 2: *Support Based Distributed Optimization is complete with respect to hard constraints.*

Proof: *Assume that SBDO is not complete. Let s denote the current complete assignment in an execution of SBDS, where $s(v)$ is that value observed by v itself. Let N be the union of all nogoods held by each agent. Let $I = t : \forall n \in N, n \not\subseteq t$ be a set of assignments, defined by the current set of nogoods. We will prove that $N' \supset N$ following a finite number of iterations. Clearly, if a nogood is generated by any agent in an iteration, then $N' \supset N$. Assume instead that no new nogoods are ever generated. From Corollary 1 we know that the scope of view for each agent will eventually stabilize. As agents may only base their own value selection on their view, and we have assumed that SBDS does not terminate, we know that agents must be involved in a cycle. However, we have shown that the postponement mechanism will eliminate cycles in a finite number of steps! So to maintain the assumption that SBDS does not terminate, we can conclude that a new nogood must eventually be generated and $N' \supset N$. By the definition of I we can see that $I' \subset I$ and so I is convergent to some minimal set.*

As I is determined by the nogoods N, and each nogood is only generated by an inconsistency, we can be sure that I and I' contain all solutions. Further, I clearly contains all partial solutions, and membership is testable in time linear in $|N|$. Therefore I satisfies the conditions, and we have proven that it will converge. From these results and Theorem 1 of [23] we can conclude that SBDO is complete with respect to hard constraints. □

Theorem 3: *Assuming messages are not lost, Support Based Distributed Optimization is complete with respect to hard constraints in dynamic environments.*

Proof: *As per Theorem 2 we know that as long as nogood derivation is correct SBDO is complete with respect to hard constraints. As such it is sufficient to show that nogood derivation is still correct in dynamic environments. From the algorithm it is clear that generating new nogoods is unchanged in*

a dynamic environment. So we need to show that all obsolete nogoods are deleted and that the deletion of obsolete nogoods does not interfere in the event the same constraint is re-posted.

A nogood only becomes obsolete when one of the hard constraints it violates is removed from the problem. As all nogoods contain the set of hard constraints they violate it is possible to identify obsolete nogoods. Further, all agents record all the nogoods they have sent, and so know which of their neighbours have obsolete nogoods.

An agent can only have obsolete nogoods if it received them from an agent that knows the removed constraint or an agent which had received an obsolete nogood earlier. The agents that know the removed constraint must be informed that the constraint has been removed by the environment. They then inform their neighbours who have obsolete nogoods that the constraint has been removed, who then inform their neighbours, etc. Therefore all agents who have obsolete nogoods will be informed that the constraint has been removed. Removing the nogoods from sent-nogoods *ensures that a neighbour will not be informed twice by the same agent, preventing an infinite loop.*

Because each agent keeps a record of all the nogoods it has sent, and which agent it sent them to, for each of its neighbours it knows some of the nogoods it has which are now obsolete. Further, because nogoods can only be generated by an agent's neighbours, between all of an agent's neighbours, they know all of the nogoods this agent has that are now obsolete. Finally as all agents involved will be informed that the constraint is removed, every agent will eventually be informed of all nogoods it has received or are in transit to it that are obsolete. The remove nogood parts of the algorithm then ensure that all obsolete nogoods are removed from an agents knowledge and hence no longer affect the solution.

It is possible that the constraint is reinstated before all the nogoods that were rendered obsolete by its removal have been deleted. It is then possible for a new nogood, which is the same as a previous obsolete nogood to be received by an agent before the previous version. In which case the new proposal will be deleted instead of the old, still in transit proposal. However this does not matter, as the old proposal is no longer obsolete (the constraint has been reinstated), and as they are logically the same nogood, no information is lost.

Therefore SBDO is still complete in dynamic environments. □

Theorem 4: *Given that messages always arrive in the order they are sent, SBDO is correct when agents in the network fail.* **Proof:** *When a single agent A fails all of the information required for correctness is preserved by its neighbours. Each of its neighbours records the set of nogood messages that it sent to A. Similarly they record all the nogood messages that they received from A. When A restarts it requests this information from its neighbours.*

If two or more non-neighbouring agents fail simultaneously it is equivalent to many single agents failing.

When two neighbouring agents A and B fail simultaneously, the messages that A sent to and received from agents other than B will be preserved by those other agents and vice versa. So only the messages exchanged between A and B are lost. Between A and B, A forgets that it sent message M to B and B forgets that it received message M from A. So the entire transaction regarding M has been lost. Because of this each agents set of sent and received messages are still consistent.

If three or more neighbouring agents fail simultaneously it is equivalent to many pairs of agents failing simultaneously.

Therefore even when agents fail the knowledge held by the agents is still consistent. As such the

procedure for removing obsolete nogoods is still correct. □

Theorem 5: *SBDO is an instance of SDCOP with the following limitations:*

- *The domain of variables ($\bigcup \mathcal{D}$) must be finite.*
- *There must be a total pre-order over solutions, i.e. $\forall a, b \in V, a \prec b \vee b \prec a$.*
- *The quality of a solution must be monotonically non-decreasing as the solution is extended, i.e. $\forall a, b \in V, a \otimes b \preceq a$.*
- *Variables can not be shared between agents, i.e. $\forall a, b \in \mathcal{X}, W_a \cap W_b = \emptyset$*

Proof: *We will show that this restricted version of SDCOP is equivalent to the definition of a DCOP used in SBDO.*

SBDO assigns each agent exclusive control over a set of variables. This is equivalent to SDCOP where each agent has write access to disjoint sets of variables, as specified in this restricted definition.

The domain of the variables must be finite for SBDO, which is enforced via one of the restrictions.

SBDO partitions the set of constraints into hard constraints and objectives. To represent this as an SDCOP, it requires a multi-critera idempotent semiring. This semiring $comb(\langle\{True, False\}, \wedge, \vee\rangle, \langle V, \otimes, \oplus\rangle)$ is suitable. Where $\forall a, b \in V, (a \otimes b) \preceq a$ and there exists an element $v \in V$ such that v is the unit element of $\otimes$. Given this semiring, hard constraints can be represented as constraints of the form $\langle def_i, con_i : (\bigcup \mathcal{D})^k \longrightarrow \{True, False\} \times \{v\}$. Objectives can be represented as constraints of the form $\langle def_i, con_i : (\bigcup \mathcal{D})^k \longrightarrow \{True\} \times \{V\}$ □

3.4.3 Example

Example: *Consider the following constraint optimization problem with three variables, δ, θ and γ, each controlled by one agent Δ, Θ and Γ respectively. Their respective domains are $\{0, 1, 2\}, \{-1, 0, 1\}$ and $\{-1, 0, 1\}$. The objectives are $\min(\delta \times \theta), \min(\theta), \min(\gamma)$ and there is one constraint, $\theta < \gamma$.* In this problem agents δ and θ are neighbours as they share an objective, and agents θ and γ are neighbours as they share a constraint.

Initially no agents have any information from their neighbours so in alg. 5 they chose their assignments based on only local information, in this case, $\theta = -1$ and $\gamma = -1$ from their local objectives, while $\delta = 1$ is chosen randomly (as all its options are equally good). All agents then inform their neighbours of their decision by sending proposals. Δ sends the proposal $\langle\langle\langle\Delta, \{\langle\delta, 1\rangle\}, 0\rangle\rangle, \{\}\rangle$ to Θ, Θ sends the proposal $\langle\langle\langle\Theta, \{\langle\theta, -1\rangle\}, 2\rangle\rangle, \{\}\rangle$ to Δ and Γ sends the proposal $\langle\langle\langle\Gamma, \{\langle\gamma, -1\rangle\}, 2\rangle\rangle, \{\}\rangle$ to Θ.

When Θ receives the proposal from Γ, it notices that the proposal is inconsistent with its knowledge, as there is no value in its domain less then -1. This causes Θ to send the nogood $\langle\{\langle\gamma, -1\rangle\}, \{\theta < \gamma, \theta \geq -1\}\rangle$ to Γ. After receiving the proposals all the agents decide which agent to use as their support. Θ has to chose between itself and Δ. Θ can not chose Γ as its support as the proposal Γ sent was inconsistent. The utility of Θ's current view is 2, which is better than or equal to all the others so it keeps itself as its support. Similarly Δ and Γ change their support to Θ. When Δ chooses Θ as its support, its view now includes the assignment to Θ, therefore it now has enough information to evaluate the shared objective and so picks $\delta = 2$. Θ and Γ view's have not changed, so they do not send new proposals, while Δ sends the proposal $\langle\langle\langle\Theta, \{\langle\theta, -1\rangle\}, 2\rangle, \langle\Delta, \{\langle\delta, 2\rangle\}, 0\rangle\rangle, \{\langle\min(\delta \times \theta), 2\rangle\}\rangle$

to Θ. It has to include the assignment to Θ in the proposal it sends, as it is required to detect cycles. If there were any assignments before the assignment to Θ they can not be sent in the proposal to Θ.

Next, Γ receives the nogood from Θ and so is forced to change its assignment to $\gamma = 0$ and sends another proposal to Θ with its new assignment. Simultaneously Θ receives the new proposal from Δ, but does not make any changes because of it, so does not send a new proposal. As no agent disagrees with any other they all stop sending messages. If this is a static problem the algorithm can be safely terminated, however if it is a dynamic problem the agents simply wait for a change in the problem.

To illustrate the dynamic nature of the algorithm, we now remove the constraint $\theta < \gamma$ from the problem. To do this, the environment sends remove-constraint messages to Θ and Γ. Γ has not sent any nogoods so has nothing to do, while Θ has sent a nogood to Γ which is now obsolete, so it sends the remove-nogood message $((\theta < \gamma), \{\langle\{\langle\gamma, -1\rangle\}, \{\theta < \gamma, \theta \geq -1\}\rangle\})$ to Γ. Also as there is no longer a link between Θ and Γ they are no longer neighbours. Meanwhile Δ has not received any messages so is still waiting.

Finally γ receives the constraint removed message, deletes the obsolete nogood and so is again able to adopt the assignment $\gamma = -1$, however it has no neighbours to send an proposal to. No agents have any messages to send, so the network has again reached quiescence.

3.5 Performance Improvements

The performance of SBDO can be improved via two independent changes. First is generalizing the concept of a nogood used in SBDO. Rather than eliminating a single partial solution from the search space, we generalize a nogood to eliminate a volume of the search space (which may be as small as a single solution). This allows the algorithm to eliminate the bounded infinite search space with a finite number of nogoods, ensuring termination. Second, the algorithm is modified to exploit additional information the agents already have by implementing limited forward checking. When an agent learns additional constraints that apply to other agents (generally via nogood messages) it attempts to satisfy those constraints as well as its own. This allows the algorithm to avoid unnecessary messages between agents. Together, these changes also allow the algorithm to solve problems with a bounded infinite domain.

3.5.1 Region Nogoods

SBDO uses nogoods to remember sections of the search space that it has explored and found to not contain a solution. Each nogood describes a point in an n-dimensional space, where n may be less than the number of variables (dimensions) in the entire problem. If a proposed solution includes the point described by a nogood, then the proposed solution is invalid. This is sufficient for problems with finite domains. It does not work for infinite domains, as an infinite number of nogoods may be required to eliminate the entire search space.

To overcome the problems with nogoods in infinite problems, we redefine the concept of a nogood to exclude an arbitrary closed region of the search space rather than a single point in the search space. This region of the search space is described by a set of equalities and inequalities, which describe the boundaries of the region. In the simplest case these will be unary ($x < 5$) though in general they could be of any arity. The higher arity equalities and inequalities cause problems when nogoods are

'relaxed', which is required for SBDO. To relax a nogood, variables which are currently required to test the nogood are removed, making it match more solutions. If only some of the variables which are used in a equality or inequality are removed, the equality or inequality must be retained, otherwise the nogood will be relaxed too much. To ensure the nogood is relaxed correctly, the variables that have been removed are tracked within the nogood. When all of the variables involved in an equality or inequality have been removed from the nogood, the inequality must also be removed.

Definition 15: *A* **region nogood** *is a tuple* $\langle region, justification, deleted \rangle$*. Region is a set of continuous equalities or inequalities that, together with the domain bounds, describe a closed region of the search space. Justification is a set of constraints such that, for every point in the region, the solution or partial solution described by that point violates at least one constraint in justification. Removed is a set of variables which have been projected out of the nogood as part of the relaxation procedure.*

A nogood is an **ideal nogood** *iff there does not exist a larger region which includes the original region and satisfies the definition, and there does not exist a smaller justification that satisfies the definition.*

A simple example of a region nogood, with three inequalities constraining the values of two variables is:

$$\langle \{x > 2, y > 3, x < 10 \times y\}, \{c_1, c_2\}, \emptyset \rangle$$

Definition 16: *Given an* $DCOP = \langle ... \rangle$ *The operator* $\ominus : Nogoods \times 2^{\mathcal{V}} \times 2^{\mathcal{C}} \longrightarrow Nogoods$ *returns a new region nogood which is the result of projecting a region nogood onto a set of variables which does not include the set of input variables.*

Given an original region nogood $N = \langle R, J, D \rangle$*, a set of variables* $V = \{v_0, \ldots, v_n\}$ *and a set of constraints* $C = \{c_0, \ldots, c_m\}$*, the new region nogood* $N' = \langle R', J', D' \rangle$ *must satisfy the following properties.*

- $D' = D \cup V$
- $J' = J \cup C$
- *for all* $j \in J$ *if* $s(j) \subseteq D'$, $j \notin J'$ *else* $j \in J'$

The previous region nogood, after projecting out the variable y may become:

$$\langle \{x > 2, x < 10 \times y\}, \{c_1, c_2, c_3\}, \{y\} \rangle$$

Note that the inequality $x < 10 \times y$ remains, even though one of the variables in it has been removed. Also, the region is still closed, as it is a requirement that the domain of y is bounded.

Lemma 5: *After applying the operator* $\ominus$ *to a valid region nogood* $N = \langle R, J, D \rangle$*, the new region* R' *when combined with the domain bounds is closed.* **Proof:** *When variables are removed from the region nogood, the equations that limit the values of those variables are removed. This effectively replaces the previous limits by the bounds on the domain. By the definition of a nogood, the equations are continuous. Therefore the region remains closed after removing an equation.*

In the case that not all of the variables involved in an equation are removed from the nogood, the equation is retained. Because they are retained, the region does not expand to include any points which do not violate the justification. □

This change will result in less nogoods being required, reducing the memory and communication requirements of the SBDO algorithm. Though it is significantly more complex to generate nogoods and to test if a partial solution violates a nogood, increasing the computational requirements.

3.5.2 Forward Checking

In SBDO, an agent only ensures that its proposal is consistent with its own constraints and relies on its neighbours to check the proposal is consistent with their constraints. Many messages can be saved if an agent ensures its view satisfies the known constraints of its neighbours as well as its own constraints. This is done by also assigning a value to the agent's neighbour's variables (which do not already have a value) as well as assigning a value to the agent's own variables. Note that the values for other agents chosen by this agent are not communicated to its neighbours, as such an agent does not gain any more influence over the other agents. Further, the agent does not have complete knowledge, so it is still possible that it chooses an inconsistent value. No extra communication or knowledge is required, as an agent already learns the important constraints on other agents via the nogood messages it has received.

To illustrate the need for forward checking, we present a worst case example of what happens without forward checking, then with forward checking. Region nogoods are used in both cases. In this example there are three agents, A, B and C, each of which have write privileges to one variable α, β and γ respectively. The domain is (0..1) and the constraints are $\alpha > \beta, \beta > \gamma, \gamma > \alpha$. Without forward checking, the following processing sequence may occur:

1. C receives the proposal $\langle\langle\langle, A, \{\langle\alpha, 1\rangle\}, 1\rangle, \langle B, \{\langle\beta, 0.99\rangle\}, 0.99\rangle\rangle, \{\}\rangle$.
2. C is unable to choose a value for γ that satisfies the constraints, so sends the following nogood to B: $\langle\{\alpha \geq 1, \beta \geq 0.99\}, \{\gamma > \alpha, \beta > \gamma\}\rangle$.
3. B changes the value assigned to β in response the new nogood and sends the following proposal: $\langle\langle\langle, A, \{\langle\alpha, 1\rangle\}, 1\rangle, \langle B, \{\langle\beta, 0.98\rangle\}, 0.98\rangle\rangle, \{\}\rangle$
4. C is unable to choose a value for γ that satisfies the constraints, so sends the following nogood to B: $\langle\{\alpha \geq 1, \beta \geq 0.98\}, \{\gamma > \alpha, \beta > \gamma\}\rangle$.
   $\vdots$
5. Eventually, C sends the following nogood to B: $\langle\{\alpha \geq 1, \beta \geq 0\}, \{\gamma > \alpha, \beta > \gamma\}\rangle$.
6. Now that all of the (infinite) possible values for β have been exhausted, B sends the following nogood to A: $\langle\{\alpha \geq 1\}, \{\gamma > \alpha, \beta > \gamma, \beta \geq 0\}\rangle$.
7. When A receives the nogood, it changes the value assigned to α to 0.99.
8. Soon C receives the proposal $\langle\langle\langle, A, \{\langle\alpha, 0.99\rangle\}, 0.99\rangle, \langle B, \{\langle\beta, 0.98\rangle\}, 0.98\rangle\rangle, \{\}\rangle$ and the cycle continues.
   $\vdots$
9. Eventually, the (infinitely many) possible combinations of values for α and β are exhausted. This causes A to generate the empty nogood $\langle\{\}, \{\gamma > \alpha, \beta > \gamma, \beta \geq 0, \alpha \geq 1\}\rangle$ and the algorithm terminates with no solution.

With forward checking, the same scenario is resolved as follows:

1. C receives the proposal $\langle\langle\langle, A, \{\langle\alpha, 1\rangle\}, 1\rangle, \langle B, \{\langle\beta, 0.99\rangle\}, 0.99\rangle\rangle, \{\}\rangle$.
2. C is unable to choose a value for γ that satisfies the constraints, so sends the following nogood to B: $\langle\{\alpha \geq 1, \beta \geq 0.99\}, \{\gamma > \alpha, \beta > \gamma\}\rangle$.
3. B now knows all the constraints in the problem, so it attempts to choose a value for β such that there is still a consistent value for γ. This is impossible, so it sends the following nogood to A: $\langle\{\alpha \geq 1\}, \{\gamma > \alpha, \beta > \gamma, \alpha > \beta\}\rangle$.
4. A receives the nogood from B and attempts to choose a value for α such that there are still consistent values for β and γ. This is still impossible, so A generates the empty nogood $\langle\{\}, \{\gamma > \alpha, \beta > \gamma, \alpha > \beta\}\rangle$ and the algorithm terminates with no solution.

An agent α only learns the constraints on its neighbours when it receives a nogood. An agent may also learn constraints from agents which are not its neighbours via nogoods. Nogoods are only sent to α when α has chosen a value for its variables which is inconsistent. This means that α will only learn constraints that directly impact it. If privacy is important, an agent need not reveal its constraints immediately. The agent may send just the constraint identifier for a specific constraint in a nogood instead of the actual constraint. To ensure termination, the agent must eventually send the actual constraint. This will ensure only tight constraints are shared with the other agents.

Precomputing the constraints does not change the behaviour of the algorithm or the final solution. Partial solutions which are found to be inconsistent using forward checking would be identified by the recipient. The recipient would then reject the solution, requiring two messages (a proposal and a nogood). The order partial solutions are proposed by an agent depends on the agents objectives, so does not change.

The proofs of termination and completion must be amended to reflect these changes to the algorithm. Specifically, relaxing the assumption that the domain of each variable is finite invalidates the previous proof of soundness (Theorem 1).

Theorem 6: *SBDO with bounded infinite variable domains is sound.*

Proof: *In most situations, Theorem 1 still holds when using generalized nogoods.*

In the case of loops of the form $a_1 < a_2, a_2 < a_3, \ldots, a_n < a_1$ or $a_1 > a_2, a_2 > a_3, \ldots, a_n > a_1$, the previous proof of soundness is not sufficient. Eventually a proposal containing assignments for all of the agents $a_1, \ldots, a_{n-1}$ will be sent to the last agent a_n. As there is no solution, a_n will send a nogood to a_{n-1} containing the constraint $a_n < a_1$ (resp. $a_n > a_1$). Now that a_{n-1} has received the nogood from a_n it knows the constraint $a_n < a_1$ (resp. $a_n > a_1$) and can find that there is no solution. This process repeats until a_1 receives the nogood from a_2 containing all the constraints in the cycle. At this time a_1 determines that there is no solution to the problem and generates the empty nogood. □

Theorem 7: *SBDO with bounded infinite variable domains is complete with respect to hard constraints.* **Proof:** *Per theorem 2, a nogood is only generated when a partial solution is shown to be inconsistent. Per lemma 3.5.1, partial solutions which are consistent are not eliminated in the process of relaxing nogoods.* □

It is worth noting that the combination of region nogoods and forward checking allows sbdo to solve bounded infinite COPs.

3.6 Conclusion

We have presented the Support Based Distributed Optimization algorithm, which solves Dynamic Distributed Constraint Optimization problems by using a novel approach inspired by argumentation. In this approach there is no hierarchy among the different agents, instead each agent is able to send 'proposals', which can be viewed as arguments and represent a partial solution to the problem. A proposal contains the assignment to the variables of an agent as well as the utility of the assignment and the context in which the assignments were made. Each agent can choose one of the other agents as its support and in turn uses that agent's assignments and context as the context for its own decision. By constantly creating and communicating stronger and stronger arguments each agent is able to influence the assignment to other agents. In this way the agents are able to arrive at a good solution using significantly less resources than other DCOP algorithms. As only local communication is used the approach scales well as the size and complexity of the problem increases.

The lack of hierarchy makes this approach very flexible regarding change in the environment, so it is highly suited for solving dynamic problems, as shown in Figure 4.1.1. Further this flexibility, coupled with the large amount of duplication of knowledge makes it fault tolerant. Other agents are able to continue solving unimpeded when one or even many agents fail, as well as allowing an agent that has just restarted to quickly recreate its previous state, as shown in Figure 4.1.2.

The resulting algorithm is completely asynchronous, fault tolerant, and sound. Specifically, the algorithm retains the termination and soundness properties even when agents may fail (assuming they later restart) and messages may arrive in random order. Further it retains the termination property when messages are lost or agents fail and do not restart.

Future work involves a formal study of the privacy loss associated with this algorithm. Finally we would like to further explore how to apply properties that are useful when deploying DCOP technologies in the real world, such as making the algorithm robust against manipulation.

Chapter 4

Evaluation

In this chapter we present the emperical evaluation of SBDO. To evaluate the performance of SBDO on dynamic problems we compare it with DynCOAA [35]. To evaluate SBDO's performance on static problems we compared it with ADOPT [39], DPOP [47], DSA [71] and MGM [31].

We implemented SBDO in C++[1]. Where available, we used the implemention of the other algorithms provided with Frodo [29]. There was not an existing implementation of DynCOAA, so we also implemented it in C++.

All of the algorithms were executed on the same platform. The test platform was an Intel Xeon X3450 processor with 8GB of RAM running Gentoo Linux.

4.1 Support Based Distributed Optimization

To evaluate SBDO we implemented it using C++ and compared it with several different algorithms; ADOPT with preprocessing, DPOP, DSA, MGM and DynCOAA. The implementation of SBDO used here does not have either of the performance enhancements described in section 3.5. We used the implementations of ADOPT (in bounded optimal mode), DPOP, DSA and MGM that are provided with Frodo [29] and we implemented DynCOAA in C++. We used the parameters for DynCOAA that are recommended by its authors [35], with 15 ants in each swarm.

Non-Concurrent Constraint Checks (NCCCs) [34] were used to measure the computational requirements. This computes the largest sequence of constraint checks that could not have been done concurrently. We have measured the number of messages required separately, so do not include a message cost in the NCCC count.

We used three sets of test problems: easy, moderate and hard. The easy set consists of the 120 handcrafted meeting scheduling problems provided in [50]. These problems have between 8 and 12 variables with a constraint density (number of constraints divided by number of variables) of between 1.333 and 1.875. The moderate set consists of 60 randomly generated meeting scheduling problems. These problems have between 9 and 24 variables with a constraint density between 1.000 and 1.860. The hard set consists of 80 randomly generated meeting scheduling problems. These problems have between 12 and 48 variables with a constraint density between 1.750 and 4.000.

To ensure that the results presented here are representative of each algorithm's performance, each algorithm was run five times on every problem. The results have been averaged over all problem instances. This is particularly important for SBDO and DynCOAA as they are highly non-deterministic.

[1] Source code available from http://www.geeksinthegong.net/svn/sbdo/trunk/ .

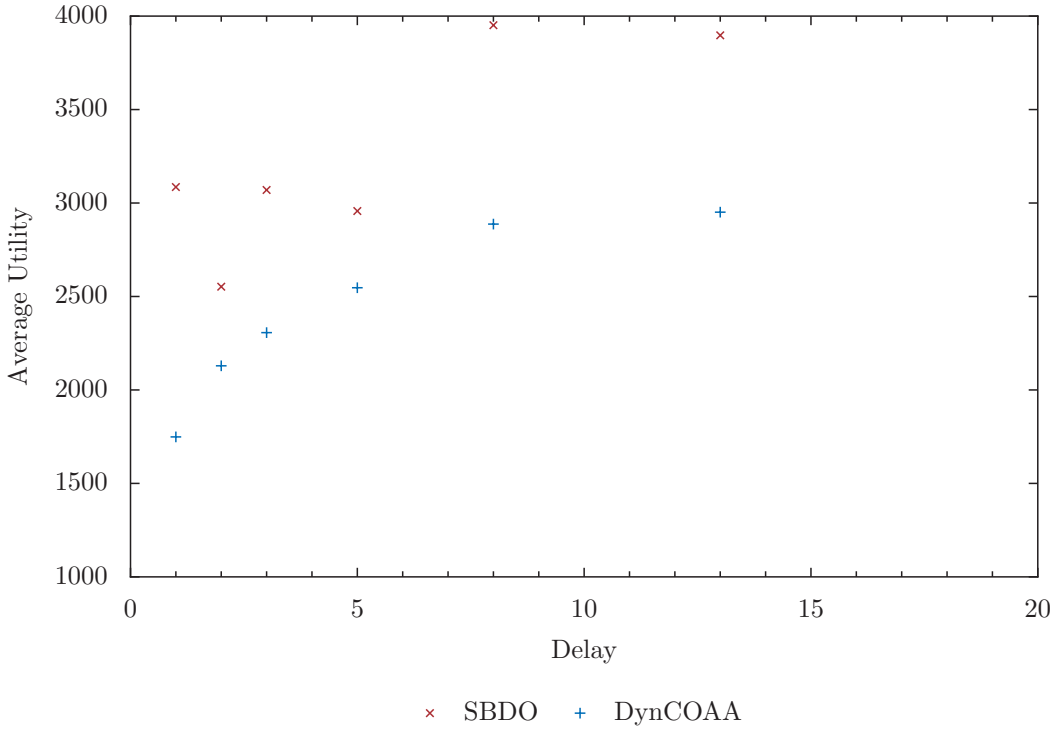

Figure 4.1: Comparison on dynamic problems

The order that SBDO agents are scheduled on the CPU and the initial solutions chosen by DynCOAA ants can have a significant impact on the performance.

Each problem instance was given a limit of fifteen minutes wall clock time and 2GB of memory. We acknowledge that the limit of 2GB of memory disadvantages DPOP, however it is a limitation of the java virtual machine.

4.1.1 Dynamic Problems

To evaluate SBDO's performance on dynamic problems we compared it against DynCOAA on the moderate and hard sets of problems. We were unable to compare with R-DPOP as Frodo does not support dynamic problems. Both algorithms were allowed to run for a set amount of time (1, 2, 3, 5, 8 and 13 seconds), after which they were paused. While paused the utility of the current solution is calculated, then two of the constraints were randomly replaced. The algorithm was then allowed to continue, leaving the objective function (soft constraints) unchanged. By using the same random seed we guarantee that the dynamic problems are the same for all trials.

Figure 4.1.1 shows that SBDO always outperforms DynCOAA. It is worth noting that the solutions found by SBDO are not monotonically non-decreasing. When allowed to run for two seconds it produces worse solutions than when allowed to run for only one second This is because it does not have a global communication mechanism to coordinate value changes.

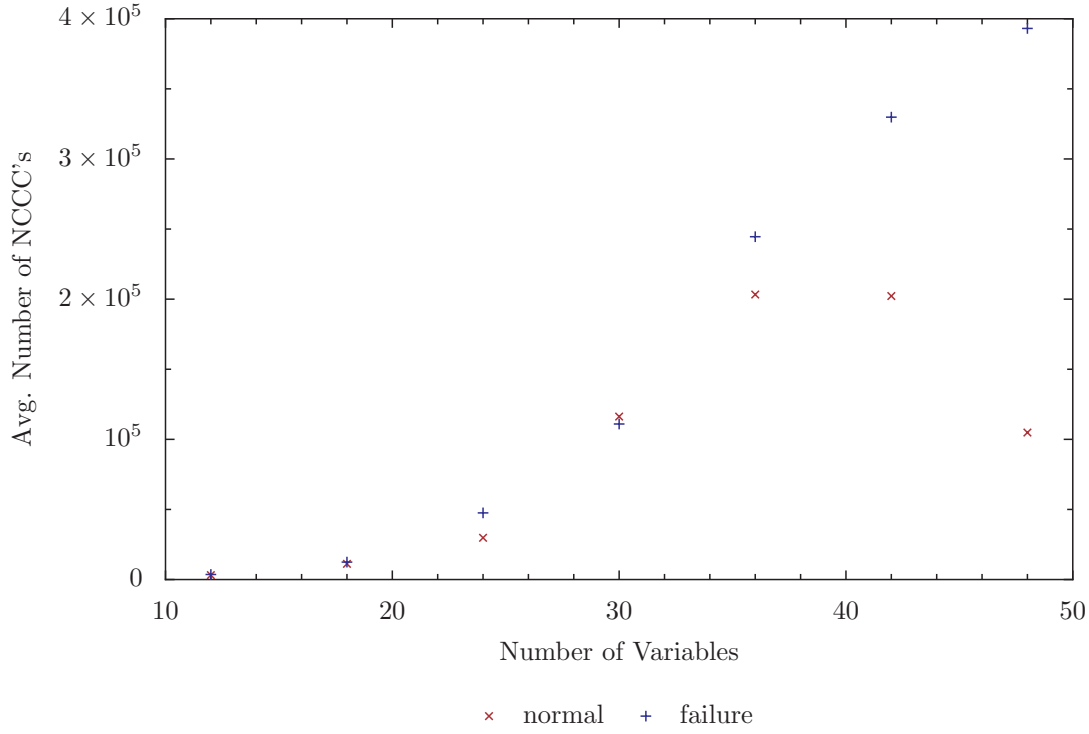

Figure 4.2: Performance degradation with unreliable agents

4.1.2 Fault Tolerance

To demonstrate the fault tolerance of SBDO it was run on the set of hard problems, as they require a significant amount of time to solve. Every 0.8 seconds a random agent was killed, then restarted 0.2 seconds later. When an agent is killed all the knowledge that agent had accumulated is lost and the other agents must continue without it. Other failure rates were tried, however they all produced similar results. As shown in Figure 4.1.2 the algorithm requires more NCCC's, so therefore more time and messages to reach quiescence. Though as shown in Figure 4.1.2 when it does terminate the solution is only slightly worse than when no agents fail.

4.1.3 Static Problems

To evaluate how SBDO performs on static problems we tested it against DPOP, ADOPT, DSA and MGM. For these results only problem instances where the solver found a consistent solution to the problem within the resource constraints are considered.

As can be seen in Figures 4.1.3 and 4.1.3, the number of non-concurrent constraint checks required by SBDO scales much better than DPOP and ADOPT as the size of the problem increases. The local search algorithms DSA and MGM scale better, but are not able to find a consistent solution to the larger problems. For small problems SBDO, DPOP and ADOPT require less NCCCs than DSA and MGM

All of ADOPT, DPOP and SBDO require about the same amount of effort for solving small problems, but the effort required for ADOPT and DPOP scales much faster than SBDO. DSA and MGM required significantly more effort to find a solution to the easy problems, but they scale better

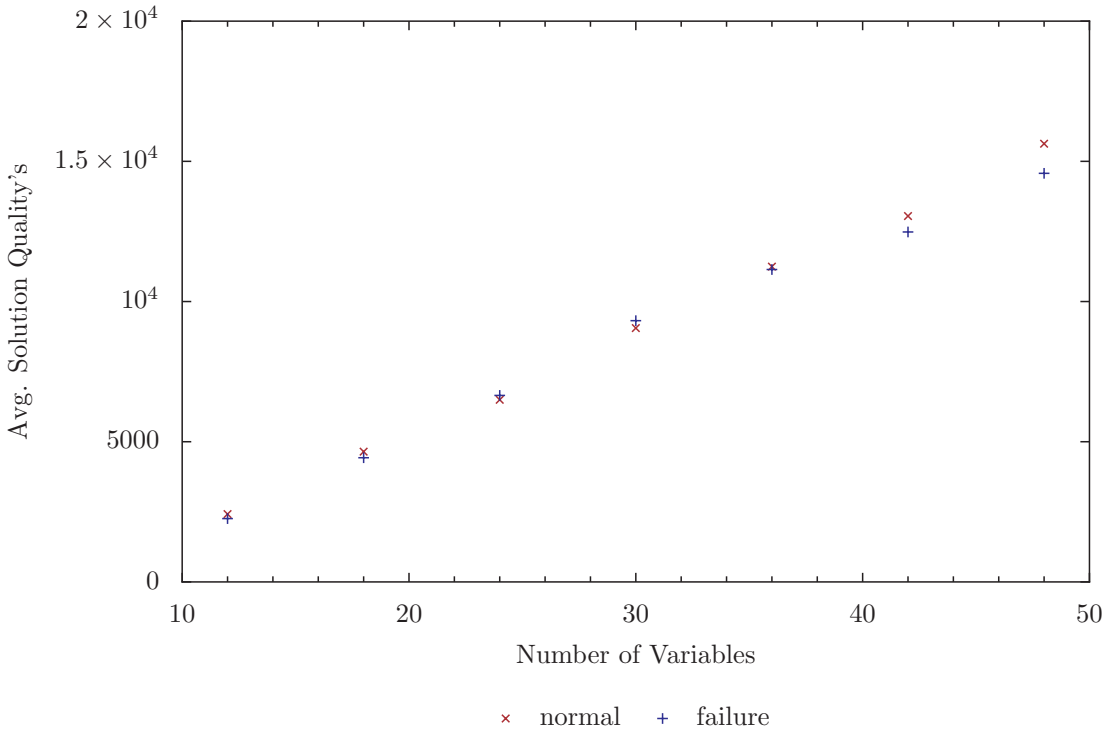

Figure 4.3: Quality degradation with unreliable agents

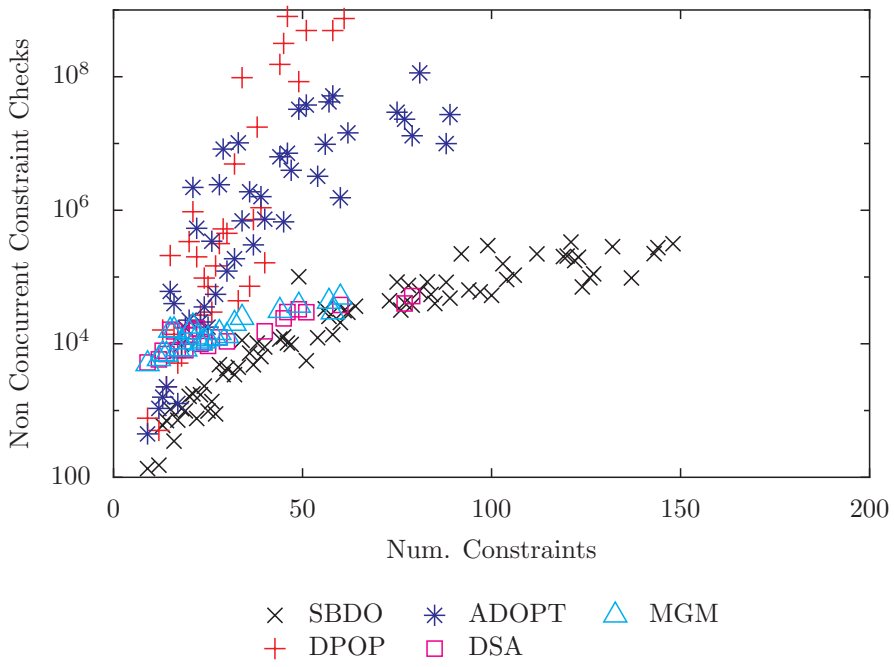

Figure 4.4: NCCCs vs number of constraints

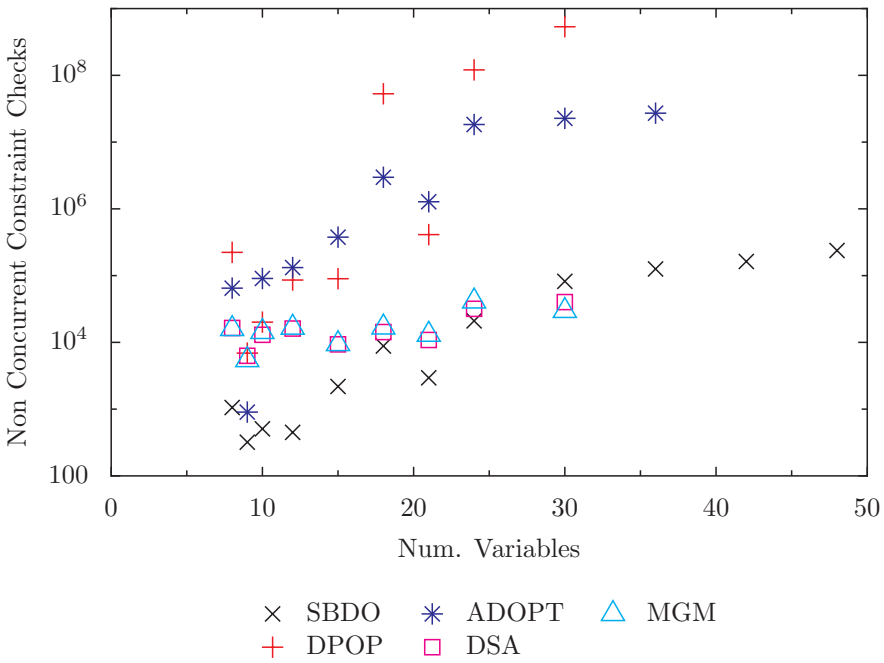

Figure 4.5: NCCCs vs number of variables

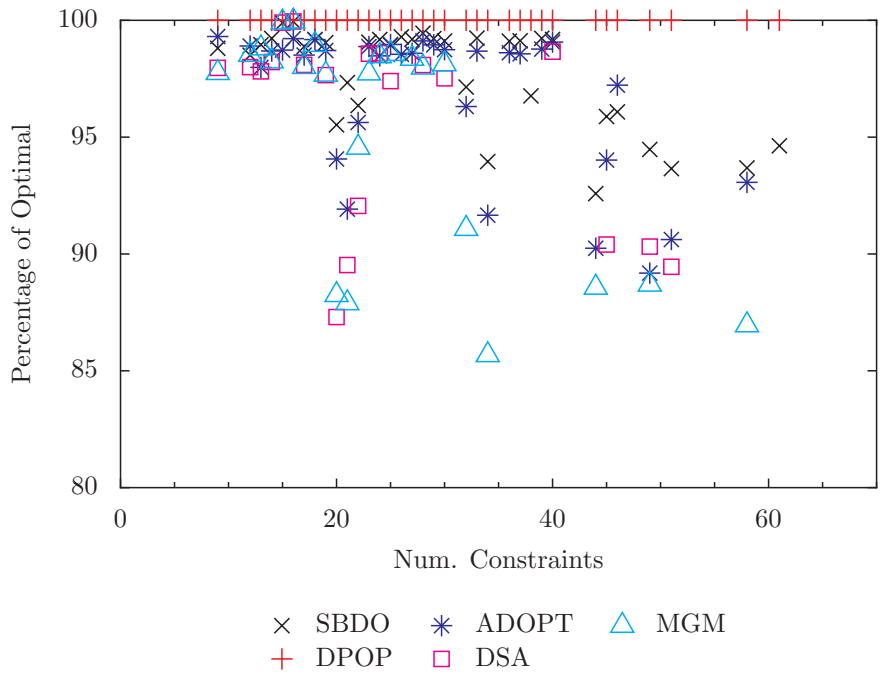

Figure 4.6: solution quality vs number of constraints

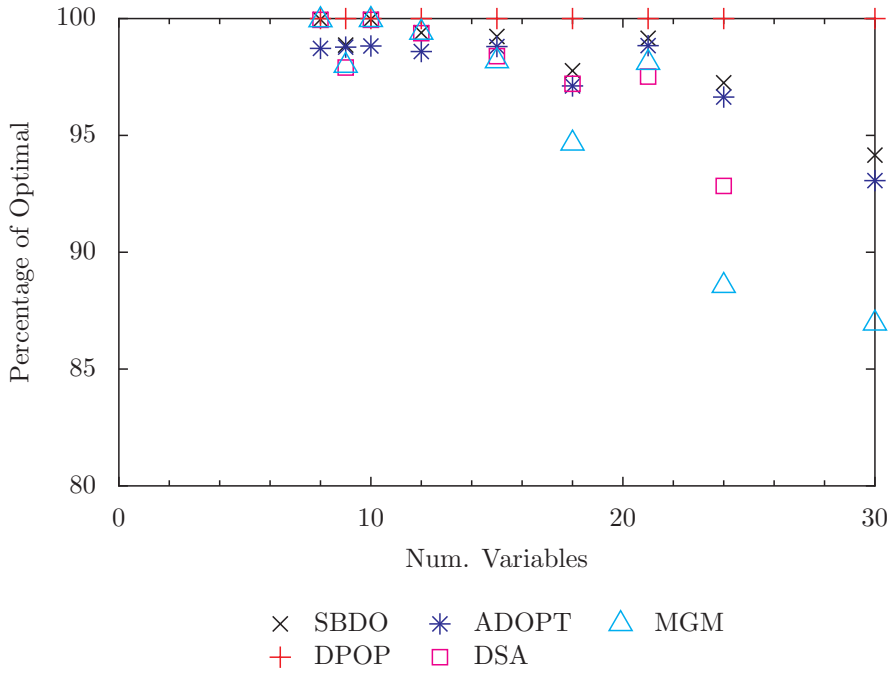

Figure 4.7: solution quality vs number of variables

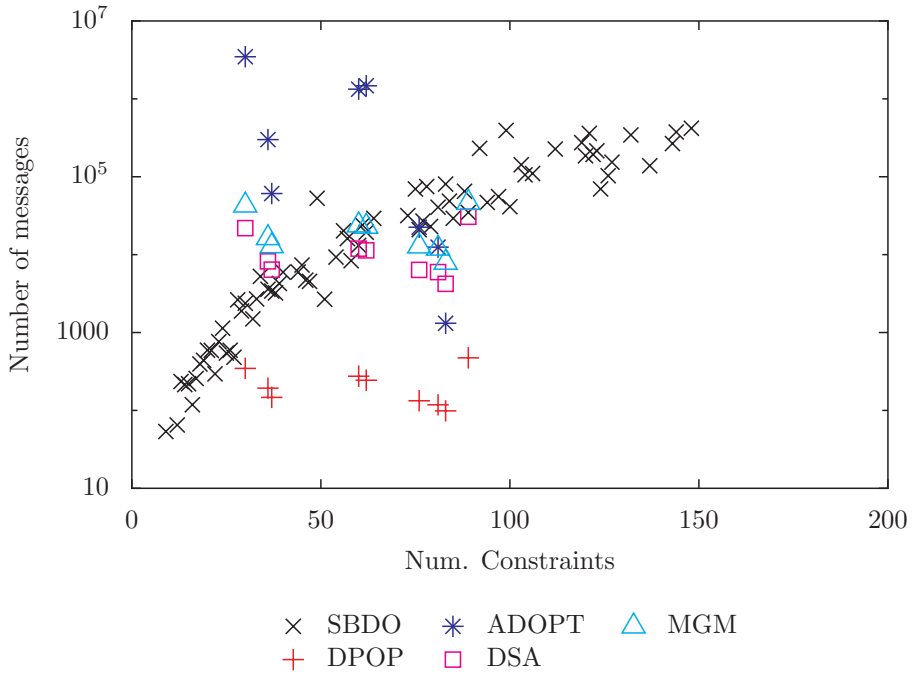

Figure 4.8: Messages vs number of constraints

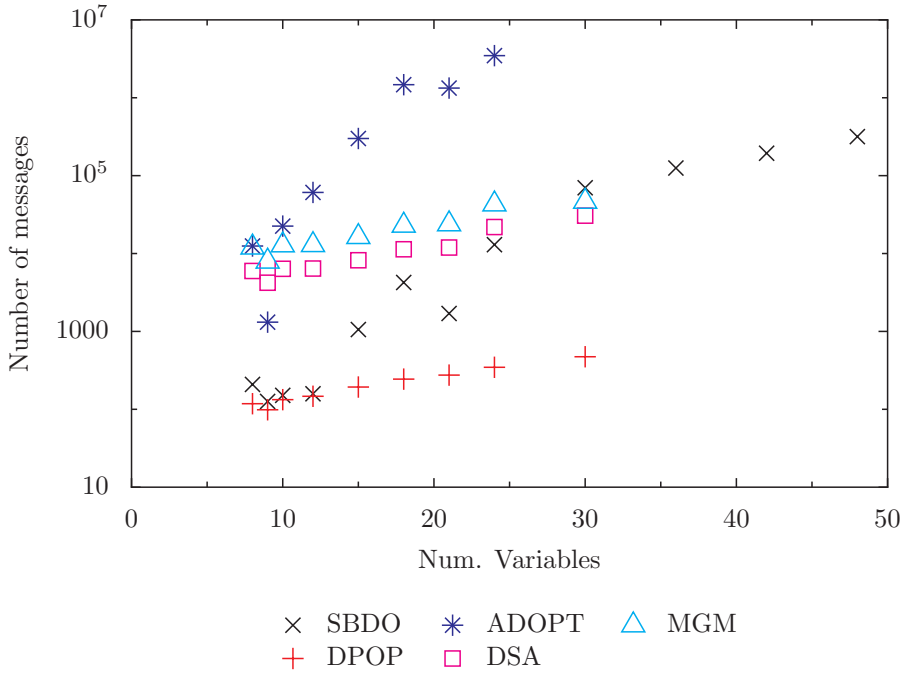

Figure 4.9: Messages vs number of variables

than SBDO as the size of the problem increases. Unfortunately they were not able to find consistent solutions to the harder problems.

In Figures 4.1.3 and 4.1.3 we can see the quality of the solution each algorithm finds, as a percentage of the optimal solution (as computed by DPOP). SBDO always computes a better solution than the other two local search algorithms, MGM and DSA, and similar quality to ADOPT. Note that ADOPT is running in bounded optimal mode. As such it terminates when a 'good enough' solution is found, rather than when the optimal solution is found.

Figures 4.1.3 and 4.1.3 show that DPOP uses the least messages, as it only requires a linear number of messages, while the number of messages used by SBDO increases steadily. ADOPT counter-intuitively requires less messages as the number of constraints in the problem increases and increases quickly as the number of variables increases. The number of messages used by the local search algorithms increase at about the same rate as DPOP, but start much higher.

We have only run experiments with up to 48 variables/agents. This is well short of real life problems which may have hundreds or even thousands of variables/agents. Nevertheless, the experiments shown here are sufficient to show how SBDO scales.

4.2 Conclusion

The results show that SBDO performs well compared to other algorithms. First, we can see that when applied to dynamic problems, SBDO dominates DynCOAA. Note that the solutions found by SBDO are not monotonicly increasing, while the solutions found by DynCOAA are. This can be a problem when there is a relatively short time to find a solution.

Next we show that in settings with unreliable agents, SBDO is still capable of finding a solution.

In this setting, more effort (NCCCs) is required before the algorithm converges on a solution. Also, the final solution is slightly worse than with reliable agents. No other published algorithm attempts to solve problems with unreliable agents, so a comparison with other algorithms is not possible.

The main comparison is with other algorithms on static problems. In this senario, the number of NCCCs SBDO requires scales similarly to the other local search algorithms, DSA and MGM. SBDO scales significantly better than the complete algorithms, ADOPT and DPOP. The number of messages SBDO requires also scales similarly to DSA and MGM. It scales significantly better than ADOPT and worse than DPOP. This is expected, as DPOP requires only a linear number of messages. Also, SBDO finds solutions which are closer to the optimal than DSA and MGM.

Chapter 5

Applications

5.1 Introduction

In this chapter we present several examples of how DCOP techniques can be applied to real world problems. These are not intended to be complete solutions to the presented problems, simply to serve as inspiration for how DCOPs can be used. Nevertheless, we touch on some important topics, such as the art of writing a good CSP encoding.

The first example is a fairly straightforward scheduling problem, scheduling radiotherapy treatment among many hospitals. This shows one of the many ways in which different solving algorithms can be combined to better solve a particular problem. In this case, SBDO for the patient agents and a more traditional centralized solver for the resource agents.

The second combines DCOP with auctions to solve a more complicated problem, optimal vehicle routing. In this case, the auction mechanic is used to determine the constraints between agents and cover failures in the DCOP solver. Once the constraints are known, the DCOP solver is used to decide the optimal route for each vehicle.

The third example shows that DCOP can be used to improve the results provided by a different tool, in this case Agent Based Modelling (ABM). By using DCOP for resource allocation within the simulation, the quality of the simulation can be improved. DCOP can even be used to generate plans for each agent within the simulation, increasing the 'intelligence' of the agents.

5.2 Radiotherapy Treatment Scheduling

Inefficiencies due to poor scheduling have been identified as a major problem within health care systems. A good schedule will not only reduce the amount of time wasted waiting for patients to arrive, but also improve patient satisfaction by providing care at times that suit them and with reasonable waiting times [10].

Research so far has focused on optimizing the utilization of operating rooms. It has been shown that reducing organizational barriers and applying sophisticated optimization techniques can improve the utilization of operating rooms in an already efficient hospital by 4.5% [64]. This shows that significant gains in efficiency can be made within the health system. In fact, the QE Foundation estimates "that 100 billion dollars over ten years can be saved in Medicare, Medicaid and VA spending alone by using [their] methodologies." [19].

5.2.1 Encoding

We represent this problem as a DynDCOP. Each of the actors in the scheduling system for a Radiation Oncology department are represented by agents. Only the core actors: Patients, Linear Accelerators (linacs) and Simulators are considered in this section. Other actors such as Nurses, Consultation rooms and Oncologists should also be considered in a complete solution. Each of these actors is represented by one agent within the DynDCOP. This agent has all the relevant knowledge of the actor and is responsible for getting the best outcome for its actor. The agents in the system can be grouped into two categories, person agents and resource agents.

A person agent has as part of its private knowledge a set of preferences ("I would like treatment between 8am and 8:45am"). A set of constraints ("I'm not available on Thursday"). It also has other private knowledge (treatment urgency, treatment details, pay rate, etc.). Not all of this knowledge is directly relevant to the scheduling problem. The agent's public knowledge consists of a set of variables, which represent the patients current schedule. A set of constraints ("Each subsequent fraction of radiotherapy must be delivered at least 6 hours after the previous fraction"). Finally it has a set of objectives ("Once committed, an appointment should not move by more than 15 minutes").

Person agents only communicate with resource agents. This is to reduce the links between agents and so minimize the exchange of messages. This also helps with privacy by having tighter control over the information flow. Only the agent representing a person within the system knows the treatment schedule which has been prescribed to that person. The schedule is represented as constraints within the agent.

Similarly, resource agents also have public and private knowledge. The agents' public knowledge is which time slots are open and which time slots are currently booked. The nature of the SBDO algorithm means that the agent which has booked a time slot is public, but that information can be concealed. The agents knowledge also contains consistency constraints, such as the times when the resource is unavailable and usage limits.

Resource agents communicate with both person agents and other resource agents. The resource agents will normally be another resource on which this resource depends. In general, resource agents do not communicate across hospital boundaries, preventing the information flow that may violate hospital privacy. The resource agents have no knowledge of the schedule of each person agent, or even other resource agents. They are responsible solely for ensuring that the resource they represent is optimally utilized.

This encoding intentionally does not include the concept of hospital boundaries. The intention is to allow easy communication between hospitals, and explicitly encoding them would restrict that. The boundaries do still exist and must be taken into account, both for privacy and for patient agents to find the best resource to use. This encoding is flexible enough that the basic types of agents can be extended to model most things in the health system. There are some resources, such as waiting rooms, that can not be easily modelled using the classes of agents presented here. New classes would have to be developed to represent them.

We shall illustrate how the agents interact using a simple example. Consider a radiotherapy treatment centre with 2 simulators, 2 Linear Accelerators (linac) and 20 patients. The schedule is the same for each patient, they must undergo a simulation in a simulator before being treated on a linac. Each patient will be assigned to one of the two simulators and one of the two linacs, resulting in a

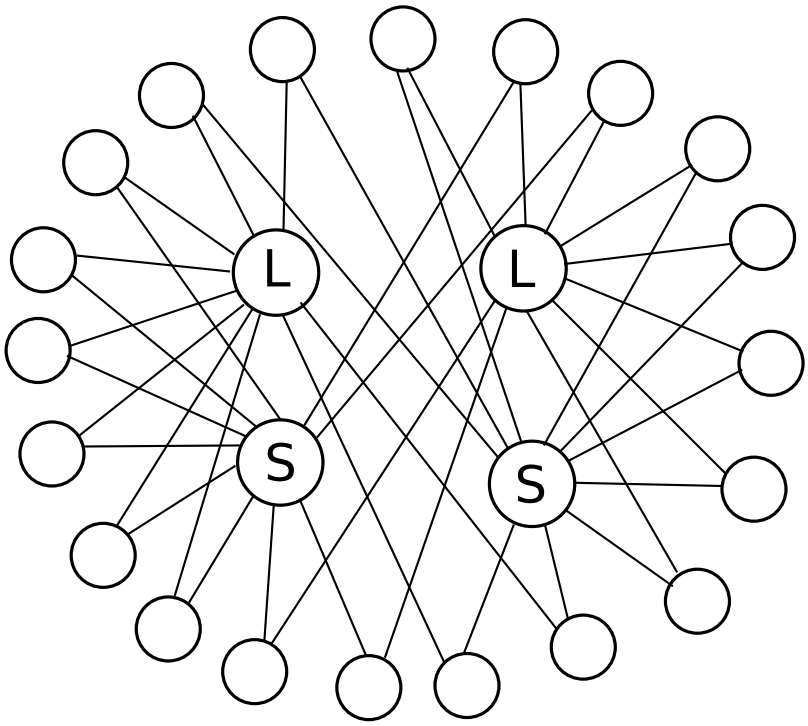

Figure 5.1: The connections between agents in this model.

neighbour graph that is similar to the one in Figure 5.1 . Initially they will be assigned randomly, so that each patient is then a neighbour of exactly one linac and one simulator. They will then start negotiating for their preferred time slot. Those who think they can get a better time slot on the other resource will change their resource assignment. An agent can do this by removing the constraints between itself and the previous resource, then adding new constraints between it and the new resource. In this way the patients will rearrange themselves, both on resources and time slots to get a solution that optimizes the objectives.

5.2.2 Solving

This problem formulation leads to a star shaped communication network, with resource agents as the hubs and person agents as the leaves. There will be significantly higher density of links within the bounds of each hospital than there is between hospitals. The links between hospitals correspond to patients that are using resources from both hospitals or resources in one hospital that requires the support of another hospital. The structure of the communication network shows that there is only a small amount of data transferred between hospitals. This results in minimal privacy loss for each hospital.

Unfortunately SBDO is not well suited to solving problems with this structure. This is because the algorithm relies on agents forming coalitions in order to influence the value of more powerful agents. It doesn't work in star shaped networks, as the weak (person) agents can not communicate with each other to form a coalition and influence the powerful (resource) agents. Because of this the resource agent will only take notice of one of the (potentially hundreds of) neighbouring person agents. In order to overcome this the way that resource agents operate must be modified.

To overcome this, resource agents must be SBDO wrappers around a centralized optimization algorithm. The centralized algorithm uses partial knowledge of each person agent's preferences to attempt to schedule treatments. When there is a conflict between two agents, A and B, it must elicit more preferences from the agents. To do so it constructs a proposal using A as support to send to B, and a proposal using B as support to send to A. If more than two agents conflict, the algorithm chooses as support the set of assignments that leads to the best utility. The person agent's response (or lack thereof) indicates their preferences. While the resource agent retains the person agent as a neighbour it caches the latest utilities received from the person agent.

In the standard SBDO algorithm, when an agent is removed from the set of neighbours all the knowledge related to that agent is also removed. Such loss poses problems in this setting. When a patient agent changes the resource it plans to use, it adds the new resource agent to its neighbours and removes the old one. However it must remember the utility gained from using the previous resource, both to ensure that the change leads to a better solution, and to ensure that it doesn't change back if it does not like the new solution. These saved utility values do not have to be kept up to date, they simply serve to inform the agent's later choices. Only person agents need to retain this information.

For every resource agent that a person agent contacts, the person agent must record the most recent local utility gain from the schedule offered by that agent. In other words, the value of the objective functions that depend, either directly or indirectly, on a variable controlled by that agent. The agent can then periodically compare the estimated utility gain from other resource agents, either a best guess or the stored value, to decide if it is worthwhile changing. To influence how often person agents change between resources, objective functions can be defined that emulate stability constraints in other algorithms. These serve to provide an estimate of both the cost associated with changing to a different resource, and the probable utility gain.

5.3 Traffic Scheduling

Due to the popularity of cars, congestion on city roads is a major problem. There have been many attempts to improve the efficiency of traffic flow. Some approaches simply aim to make traffic lights 'smarter', while others identify congestion points and encourage drivers to take a different route. As many of these solutions do not plan ahead, they often just move the congestion point to another part of the road network. In many cases, the new part of the road network is not designed to handle as much traffic load, so the actual congestion is worse.

In this approach, each vehicle in the traffic network is an agent in a DynDCOP problem. Each vehicle communicates with the other vehicles to find the best solution. The problem with this approach is that two vehicles can not know that they share a constraint (plan to use the same road at the same time) until they are on the road. At this point it is too late to do any optimization.

It is infesiable for each vehicle to broadcast its planned route, so any one agent can not know when it will conflict with other agents. In order to overcome this problem we describe a hybrid system. This system combines DynDCOP with auctions. The auctions are used to identify constraints between vehicles and to produce a good solution which the DynDCOP solver can improve. This auction system will be controlled by a central authority, typically the government department that is responsible for road infrastructure. This also allows the central authority to dynamically adjust the capacity of each road, due to an accident or changing the number of lanes available.

In this hybrid system we combine the best attributes of both a distributed system and a centralized system. It maintains the privacy of all people using the system while still allowing traffic flow to be managed by the central authority.

For the DynDCOP solver we use SBDO, while for the auction system we use the Contract Net Protocol (CNP) [62]. We assume that every vehicle has a computerized device and is capable of two way wireless communication with other nearby devices and fixed infrastructure. This device also needs to know the road network and be capable of planning a route between two points. This could be a dedicated in car navigation system or a smart phone. The device will function as both an SBDO agent and a CNP bidder.

The protocol starts when a route is requested to a destination. The agent then computes a route to use and the time required. Once an initial route has been chosen the device contacts each of the auctioneer agents that control the roads it intends to traverse. First it registers its intention to traverse the road at a given time. After which the auctioneer agent informs the agent of all other vehicles intending to use the time slot. These other vehicle agents are added as the agent's neighbours. It does not matter if one agent is not able to communicate with all of its neighbours, as the auction mechanism enforces the constraints.

When the auction for a time slot opens, all vehicles that registered their interest are notified. It is assumed they already know their local cost if they are granted privileges to traverse this road. To calculate their bid they must plan a different route, assuming they are not given privileges to traverse this road. They then submit their bid as the difference between the costs of the two routes. The auctioneer then selects the *capacity* highest bids and grants them the privilege to traverse this road during this time slot. Due to the DynDCOP, it is expected that the number of vehicles planning to use a given road segment does not exceed or only slightly exceeds the road's capacity. Any leftover capacity is made available as per reserved capacity. The vehicles not granted privileges must change their route to a different route, normally the second route used for bidding.

Due to the small lead time, the vehicles that have to redirect will often have missed the auctions for their new route. They will have to attempt to claim privileges from the roads reserve capacity. Reserve capacity is allocated on a first come first served basis.

Whenever a vehicle changes its planned route, either because its planned route was not available or SBDO's continuous optimization, it will be subscribed to auctions it no longer has an intention to bid in. If the time at which it intended to traverse that link has passed its subscription to the auction expires, otherwise it maintains the subscription. This is to ensure that the vehicle does not alternate between two different routes.

If a vehicle desires privacy it does not have to participate in the collective optimization provided by SBDO. In this case it informs each auctioneer that it desires privacy when it subscribes to the auction. None of the other subscribers will be informed about it and it is not informed about the other subscribers.

Each road has four parameters relating to the auctions, block time, lead time, capacity and reserve. Block time is the length of time that route traversal privileges are granted for. A vehicle must enter the road segment during this time window. Lead time is the amount of time between the close of an auction and the start of the block of time that was auctioned. Capacity is the maximum number of vehicles that are granted privileges for any block of time. Reserve is the percentage of the roads capacity that is kept for late arrivals. For major roads this will be low, 0-1%, while for minor roads

this will be high, 10-20%.

There is uncertainty as to whether a vehicle will traverse a road during the planning and bidding processes. The closer to the vehicle is to the road the higher the probability they will use the road. If the auctions for route traversal privileges are held too early then a significant amount of the vehicles that were granted the privilege to use the road will not be able to exercise it. While if the auction is held too late the vehicles that were not granted privilege will not have time to plan a new route and redirect. Due to different distances between exit ramps and cross roads, the required lead time for each road will be different.

5.4 Agent Based Modelling

Agent based modelling is generally used for two purposes. First is exploring the behaviour of highly complex systems. One example is simulating the cause and progress of epidemics [44]. Second is trying to improve the performance of complex systems, such as patient movement in a hospital. In both of these scenarios a very naive method of resource allocation is normally used, typically first in first served. This limitation compromises the accuracy of the models, hence reducing the quality of the results gained through simulation.

This is less important for the first category of models, but due to the high complexity it can easily impact the results. Consider the impact efficient operation of public transport, such as a train or bus route, can have on the overall environment. If buses run too frequently in the model, the simulation will show that the area has excellent public transport, leading to less car usage and congestion. However running the buses at a high frequency is not profitable in the real world, so the results of the simulation will not reflect reality. On the other hand, if the buses do not run frequently enough, the area will have poor public transport, leading to more car usage and congestion. In this case, if the bus operator knew of the larger demand, they would increase bus services in the area. So, again the simulation will not reflect reality. Over significant timescales the demand for bus services in the area will likely change, so it can not be 'hard coded' into the model at the start. Instead the optimal bus timetable must be calculated periodically within the simulation.

Efficient resource allocation is even more important for the second category of models, as more efficient use of resources is often key to the performance improvements desired. Even if the resource allocation currently used in reality is highly optimized and is accurately reflected in the model, it must be adapted for the change to the system that is being tested. Another factor is that finding the best to-be model often requires running the same model several times with different parameters, so that the best parameters are used when comparing this model with other models. Smarter systems in the simulation can automatically optimize the resource allocation and other parameters of the model, allowing a better solution to be found in less time.

5.4.1 Combining DCOP and ABM

The possibilities for combining ABM and DCOP fall on a continuous range. On one end, 'normal' agents are used to represent the actors and DCOP is only used to schedule access to important resources. On the other end, every actor in the system is represented by a DCOP agent and uses DCOP to find the optimal plan for their actions. The first approach requires less effort to model and

is better suited to the first use case for ABM (exploring the evolution of a complex system). Most actors do not optimize their daily life, but the people managing important resources want to ensure optimal use of their resource. The other end of the spectrum is more suited for the second use case (finding the optimal system configuration to solve a problem). The goal in this case is to optimize the interaction between all the actors and the resources, though it does require a lot of modelling effort.

Definition 17: *An Agent Based Model with DCOP is a tuple* $\langle \mathcal{A}, \mathcal{R}, \mathcal{E} \rangle$. $\mathcal{A}$ *is a set of agents,* $\mathcal{R}$ *is a set of resources and* $\mathcal{E}$ *is the environment. A resource consists of a label and a set of constraint schemas which must be true before the resource can be used. The environment is a set of constraint schemas.*

We place no restrictions on which classes of agents can be used and even allow mixing classes of agents.

Definition 18: *An optimized agent is a tuple* $\langle \mathcal{X}, \mathcal{K}, \mathcal{G}_a, \mathcal{G}_m \rangle$ $\mathcal{X}$ *is a set of actions, defined as pre/post-condition pairs as is common in the planning literature.* $\mathcal{K}$ *is the agents local knowledge, which consists of state variables and constraint schemas, both of which can be either public or private.* $\mathcal{G}_a$ *and* $\mathcal{G}_m$ *are the agents goals, defined as* $\langle S, t, c(t,t) \rangle$ *where* S *is a partial state describing the goal,* t *is the time at which the goal should be achieved, and* $c(\textit{desired time}, \textit{expected time}) \longrightarrow \Re$ *is a function which defines the cost for not achieving the goal at the desired time.* $\mathcal{G}_a$ *is the agent's achievement goals and* $\mathcal{G}_m$ *is the agent's maintenance goals.*

When running the simulation, a DCOP is automatically generated from each optimized agent's actions and constraints. Whenever there is a change in the agents or the environment, the DCOP is updated. Each agent then transforms it's part of the solution of the DCOP into a plan they can execute. Despite the planning, the plan may still fail, due to uncertainty in the environment or other agents.

5.4.2 Example

Problem Description

For this evaluation we created a very simple representation of a radiotherapy ward. There are several classes of actors in the system

1. Radiation oncologists
2. Patients
3. Physicists
4. Nurses

In addition, the following resources are modelled in the simulation.

1. Waiting room (unlimited)
2. Time (unlimited)
3. Consultation rooms (limited)
4. Linear accelerators (limited)

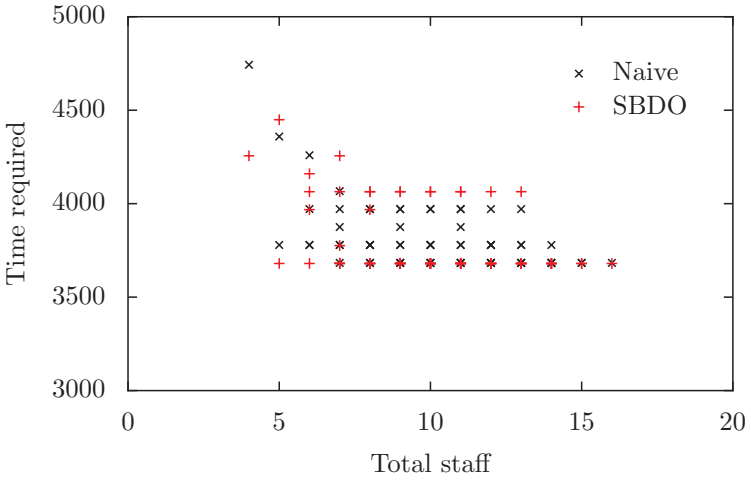

Figure 5.2: Time to treat all patients at different staffing levels.

5. Simulators (limited)

A patients goal is to receive treatment. To do so they must first have a consultation, then undergo treatment simulation, then many cycles of treatment. The start of each treatment cycle must be 24 hours or more after the start of the previous treatment cycle. For the consultation the patient must be in a consultation room with an oncologist for one hour. The simulation requires the patient to be in a simulation room with an oncologist and a physicist for 30 minutes. Each treatment cycle requires the patient to be a in a linear accelerator with a physicist and two nurses for 15 minutes.

The oncologists, physicists and nurses all share the same goal, to assist the patients to receive treatment. The objective for all actors is to minimize the time required to treat all the patients in the problem.

We modelled the problem in anylogic, using first in first served resource allocation in the naive approach and the SBDO algorithm for the DCOP approach. We then varied the staff available and identified the break point by a jump in the time required to treat all patients in the naive approach. We then used the DCOP approach and tried to reduce the staff further. For these tests, we kept the resources available constant.

5.4.3 Results

To evaluate the different approaches we fixed the number of consultation rooms (2), the number of simulators (2), the number of linear accelerators (4) and the number of patients (35). We then varied the staff levels available, the number of oncologists varied between 1 and 4, physicists between 1 and 4 and nurses between 2 and 8 in increments of 2.

The approach using SBDO required significantly less resources (2 nurses, 2 oncologists and 1 physicist) compared with the naive approach (2 nurses, 3 oncologists and 2 physicists) to achieve the minimum time to treat all the patients. The minimum time for the SBDO approach (3680 simulation ticks) was also slightly better than the minimum time for the naive approach (3683 simulation ticks).

5.5 Conclusion

It is well known that DCOP techniques can be used to solve many different types of problems. DCOP solvers can also be combined with other techniques to solve problems for which DCOP techniques alone are insufficient. Another approach, which we have not explored, is combining different DCOP solvers to solve a problem more efficiently.

In this chapter we have briefly described some problems and provided a way that DCOP can be combined with other techniques to solve them. It is our hope that these ideas will inspire other researchers and engineers to consider DCOP as a tool they can use when solving their problems.

The section 5.2 describes one way different solvers can be combined to find a better solution. To allow SBDO to perform well on this particular problem structure, we require different properties for some of the SBDO agents. Specifically, the solver used to compute local solutions must consider all of the agents neighbours, rather than just one. This also illustrates how SBDO can operate with heterogeneous agents.

Section 5.3 describes how two different techniques can support each other. The auction mechanic is used to discover the constraints in the system and provide a backup to the DCOP. While the DCOP allows the agents to communicate with each other to find an optimal solution. The resulting system is able to find better solutions than either approach individually.

Finally, section 5.4 describes a different way of combining different techniques. In this case, agents in an agent based model are augmented with DCOP techniques to improve the resulting simulation. The model performs better when using DCOP for resource allocation than with simple resource allocation.

Chapter 6

Semiring DCOP

6.1 Introduction

The simple approach of modelling DCOPs, where the cost/utility of a solution is measured as a single real number, is quite restrictive. This approach can only represent problems where there is a total pre-order over the solutions. Many real world problems only have a partial order over the solutions. Examples of these problems include problems with multiple objectives and problems with qualitative valuations. To further complicate things, a problem may include a mix of qualitative and quantitive valuations.

Bistarelli's c-semiring framework [9] is very successful for modelling problems in the Constraint Optimisation Problem (COP) domain. The Distributed Constraint Optimisation Problem (DCOP) domain introduces several new modelling challenges over centralised problems. These challenges include agent responsibility, privacy, co-operating/competing agents and no global objective. Further, these challenges lead to classes of DCOP problems which have not been addressed in the literature so far. The existing modelling frameworks do not support these new classes of problems, so we are proposing a new modelling framework inspired by c-semirings.

We have identified three classes of problems within the DCOP domain which can not be adequately described using c-semirings; equitable problems, maximisation problems and 'committee' problems. For equitable problems, we specifically refer to problems with an objective of the form 'minimise the maximum difference between the best and worst criteria'. Examples of these problems include distributing rewards to a team, assigning jobs to workers and minimise the impact of flood mitigation. Some examples of maximisation problems include maximise profit within a supply chain, maximise the value of targets tracked in a sensor network and maximise the area a team of drones can search. Committee problems are when a group of agents must agree on one (or more) shared decisions. This includes problems such as deciding on a requirements specification or a CEO's bonus package.

Equitable problems are particularly interesting within a distributed system. This is because there may not be a single 'dictator' agent who defines the entire problem. Instead the problem may grow organically, as individual agents discover they share a common goal and agree to limited co-operation. While the agents may loosely agree on the common goal, each agent has other goals, which may conflict. The end result is that there may not be a single objective which all agents subscribe to. Further, individual agents may be selfish or altruistic, which determines how much weight they put on their local objectives. In these cases, a compromise (such as an equitable solution) is required.

Equitable problems often occur in the medical setting, where the objective is to balance the patients quality of life and their survival chances. To do this, the doctor must find a balance between the most

aggressive treatment (maximum survival chances, minimal quality of life) and the least aggressive treatment (minimal survival chances, maximum quality of life). Two examples of this are treating laryngeal cancer and a methicillin-resistant staphylococcus aureus (MRSA) infection in a bone of a limb. Radiotherapy is an effective way to treat laryngeal cancer, but at high doses paralyses the patient's vocal cords. Similarly, amputation is the most effective way to treat the MRSA infection, but then the patient loses the affected limb.

One possible way to represent an equitable problem is using a tuple of n real numbers to represent the utility for each agent. This can be represented using the semiring $\langle \Re \times \Re \times \cdots \times \Re, \oplus, \otimes \rangle$. The objective is to minimise the difference between the highest and lowest values in the tuple. Defining the $\oplus$ operator as follows captures this idea.

$$\oplus(A, B) = \begin{cases} A \ if \ (\max(A) - \min(A)) \leq (\max(B) - \min(B)) \\ B \ otherwise \end{cases}$$

Where $A = \langle a_1, a_2, \ldots, a_n \rangle$ and $B = \langle b_1, b_2, \ldots, b_n \rangle$ are tuples of real numbers, so $\max(x)$ and $\min(x)$ return the largest or smallest value in the tuple respectively. The real numbers represent utility, so to get the total utility for an agent, the individual utilities are added together.

$$\otimes(A, B) = \langle a_1 + b_1, a_2 + b_2, \ldots, a_n + b_n \rangle$$

This problem can not be represented as a c-semiring. A c-semiring requires a $\top$ value which is the absorbing element of $\oplus$ and the unit element of $\otimes$. It also requires a $\bot$ value which is the unit element of $\oplus$ and the absorbing element of $\otimes$. In this problem, there does not exist a unique 'best' value to use as $\top$ or a unique 'worst' value to use as $\bot$.

Egalitarian problems, a more common class of problems with a similar intuition also can not be represented as a c-semiring. By an egalitarian problem we specifically refer to problems with the objective "maximise the minimum utility" or "minimise the maximum cost". In these problems there is still no unique best or worst value. So long as one of the values is ∞, then any assignment to the other values will compare as the equal best (or worst) value. To represent such a problem using a c-semiring, the objective must be defined as "minimse the maximum cost, then minimise the next highest cost, ... then minimise the lowest cost". This objective still satisfies "minimise the maximum cost", though it also considers the other values.

Maximisation problems are often represented using valued constraints. They can also be represented using the semiring $\langle \Re, \max, + \rangle$. These problems can not be directly represented as c-semirings, as ∞ is the absorbing element for both max and $+$. In centralised settings, maximisation problems are commonly transformed into minimisation problems (represented by the c-semiring $\langle \Re_0^+, \min, +, 0, \infty \rangle$). This transformation involves first mapping $\Re$ to $\Re_0^-$ by subtracting ∞, then changing it to a minimisation problem by multiplying all the values by -1. Performing this transformation in practice requires knowing the maximum possible utility value. In dynamic or distributed settings, this is not possible. For dynamic settings, changes to the problem may result in a larger maximum possible utility value. For distributed settings, computing the largest possible utility value requires complete knowledge of the problem, which violates one of the assumptions of distributed problems. As the transformation is not applicable in these settings, the framework has to support the maximisation problem directly.

Finally we consider committee problems. This is a class of problems where a group of agents must agree on the answer to one (or several) decisions. Committee problems can be modelled as

a standard DCOP by arbitrarily assigning agents control of decisions (variables). They could be modelled much more elegantly if agents are allowed to share control of variables. Problems of this form are usually modelled and solved as negotiation problems. If the committee problem is a sub-problem of a larger optimisation problem, then solving the larger problem requires a hybrid DCOP/negotiation algorithm or modelling the entire problem as a DCOP. Further, negotiation algorithms generally assume competitive agents, while for this work we assume cooperative, though not trusting, agents.

Committee problems are orthogonal to the utilitarian/equitable problems previously discussed. Whether a committee problem is a satisfaction, optimisation or even a multi-objective problem depends on the constraints and semiring chosen. If the agent's preferences are defined as valued constraints, then it is an optimisation problem.

We present a simple requirements engineering problem as an example of a committee problem. There are n candidate requirements (the variables, $\mathcal{X}$), which have been identified during the requirement elicitation process. Each candidate requirement can either be included (True) or excluded (False) from the specification (the domains, $D_1, \ldots, D_n$). In addition, there are m stakeholders (the agents, $\mathcal{A}$), each of whom has different preferences for which candidate requirements should be included (the constraints, $\mathcal{C}$). Finally, the objective is to maximise the total utility of the stakeholders (maximisation problem, $\mathcal{V}$).

In this setting all of the stakeholders know all of the candidate requirements and may propose a value for any of them. The stakeholders would like to keep their preferences private. Modelling this as a traditional DCOP problem would require arbitrarily assigning stakeholders control over the candidate constraints. Further, if we assume that each stakeholder has preferences regarding most of the constraints, then they will be forced to reveal their preferences to most of the other stakeholders. This is because most DCOP solvers require all agents involved in the constraint must know the details of the constraint. Recent work on asymmetric constraints [21] goes some way towards addressing this concern. By allowing multiple agents to share control of variables, this problem can be modelled naturally. The resulting model is also better at maintaining each agent's privacy.

These classes of problems may overlap, as shown in the following example. There are two agents $\{1, 2\}$ sharing control of two variables $\{\mathrm{X}, \mathrm{Y}\}$ with the same domain $\{-1, 0, 1\}$. The global objective is that the sum of the variables should be zero. Agent 1's objectives are to maximise X and minimise Y. Agent 2's objectives are to minimise X and maximise Y.

The (utilitarian) optimal solutions for this problem are:

- $X = 1, Y = -1$ (In favour of Agent 1)
- $X = -1, Y = 1$ (In favour of Agent 2)

There is one equitable solution:

- $X = 0, Y = 0$ (balance between both Agents)

A utilitarian solver will likely not return the equitable solution as a possible solution. While the equitable solution satisfies the global objective, it does not satisfy either of the local objectives.

In section 6.2 we describe the framework which we propose to model these classes of problems. Next we show the relationship between our framework and some of the established frameworks in section 6.2.3. We then demonstrate the properties of our framework in section 5.4.2. Finally the conclusion summarises our contribution.

6.2 Semiring-based Distributed Constraint Optimization Problems

In this section, we will introduce our proposed framework, the Semiring-based Distributed Constraint Satisfaction/Optimization Problem (SDCOP). The proposed framework draws inspiration from Bistarelli et al. [9]. As such, there exist many commonalities. Bistarelli et al. [9] proposed the use of a *c-semiring* for comparing different solutions. The original formulation of a c-semiring was for use within the CSP domain. We view this approach to be too restrictive for use within the CSOP domain. Specifically, a c-semiring does not support optimization problems where the objective is to maximize utility. Objective functions of this form are required for some recent algorithms (specifically SBDO [6]). Thus, we propose the use of an idempotent semiring. The use of the semiring structure allows for two main benefits. Firstly, in multi-objective problems where no total pre-order over the solutions are prescribed, the use of an abstract structure (i.e. semirings) allows the framework to induce a partial ordering. Secondly, using an abstract comparison and aggregation operators allows the framework to support comparison and combination across both qualitative and quantitative domains.

6.2.1 Preliminaries: Idempotent Semirings

A semiring consists of a set of abstract values and two operators. The two operators allows for the comparison and combination of the prescribed abstract values.

Definition 19: *A* **semiring** *[54] is a tuple $\mathcal{V} = \langle V, \oplus, \otimes \rangle$ satisfying the following conditions:*

- *V is a set of abstract values.*
- *$\oplus$ is a commutative, associative and closed operator over V.*
- *$\otimes$ is an associative and closed operator over V.*
- *$\otimes$ left and right distributes over $\oplus$.*

We shall call a semiring an **idempotent semiring** *if $\oplus$ is idempotent.*

Definition 20: *[9] A c-semiring is a tuple $\mathcal{V} = \langle V, \oplus, \otimes, \bot, \top \rangle$ satisfying (for all $\alpha \in V$):*

- *V is a set of abstract values with $\bot, \top \in V$.*
- *$\oplus$ is defined over possibly infinite sets as follows:*
  - *$\forall v \in V, \oplus(\{v\}) = v$*
  - *$\oplus(\emptyset) = \bot and \oplus(V) = \top$*
  - *$\oplus(\bigcup v_i, i \in S) = \oplus(\{\oplus(v_i), i \in S\})$ for all sets of indices S*
- *$\otimes$ is a commutative, associative and closed binary operator on V with $\top$ as unit element ($\alpha \otimes \top = \alpha$) and $\bot$ as absorbing element ($\alpha \otimes \bot = \bot$).*
- *$\otimes$ distributes over $\oplus$ (i.e., $\alpha \otimes (\beta \oplus \gamma) = (\alpha \otimes \beta) \oplus (\alpha \otimes \gamma)$).*

The idempotent property of the $\oplus$ operator can be used to obtain a partial order $\preceq_V$ over the set of abstract values V. Such a partial order is defined as: $\forall (v_1, v_2 \in V), v_1 \preceq_V v_2$ iff $v_1 \oplus v_2 = v_1$ (intuitively,

$v_1 \preceq_V v_2$ denotes that v_1 is at least as preferred as v_2). The $\oplus$ operator enables comparisons between two semiring values while the $\otimes$ operator allows us to aggregate two semiring values.

This idempotent semiring structure is now capable of representing all the different constraint schemes. As a c-semiring is an idempotent semiring, all constraint schemes that can be represented as c-semirings can also be represented as idempotent semirings.

Bistarelli has shown that classic, fuzzy, probabilistic, weighted and set based constraints are all instances of a c-semiring [9]. Valued constraints with a maximization objective are not an instance of a c-semiring, but can be represented by the idempotent semiring $\langle \Re, \max, + \rangle$ where $\Re$ is the set of values, max is the comparison operator and + is the combination operator.

Multiple idempotent semirings may be combined into a single idempotent semiring in the same way that c-semirings are combined [9]. The semirings being combined may involve evaluations on multiple heterogeneous scales - both qualitative and quantitative. We leverage this property in handling multi-objective DCOPs.

The following definition is based on that provided by Bistarelli et. al [9] (definition 7.1) for c-semiring. It formalizes the combination of idempotent semirings.

Definition 21: *Given the n idempotent semirings* $S_i = \langle V_i, \oplus_i, \otimes_i, \rangle$ *for* $i = 1, \ldots, n$ *we define the structure* $Comb(S_1, \ldots, S_n) = \langle \langle V_1, \ldots, V_n \rangle, \oplus, \otimes \rangle$ *where* $\oplus$ *and* $\otimes$ *are defined as follows: Given* $\langle a_1, \ldots, a_n \rangle$ *and* $\langle b_1, \ldots, b_n \rangle$ *such that* $a_i, b_i \in V_i$ *for* $i = 1, \ldots, n, \langle a_1, \ldots, a_n \rangle \oplus \langle b_1, \ldots, b_n \rangle = \langle a_1 \oplus_1 b_1, \ldots, a_n \oplus_n b_n \rangle$ *and* $\langle a_1, \ldots, a_n \rangle \otimes \langle b_1, \ldots, b_n \rangle = \langle a_1 \otimes_1 b_1, \ldots, a_n \otimes_n b_n \rangle$.

Theorem 8: *If* $S_i = \langle V_i, \oplus_i, \otimes_i \rangle$ *for* $i = 1, \ldots, n$ *are all idempotent semirings, then* $Comb(S_1, \ldots, S_n)$ *is an idempotent semiring.*

Proof: *From definition 6.2.1, the combined semiring uses the* $\oplus$ *and* $\otimes$ *operators from the component semirings directly. Hence, the properties that hold for the component semirings also hold for the combined semiring.*□

If $a \oplus b = a$ then for all components i, $a_i \oplus_i b_i = a_i$. This corresponds to the 'dominates' concept in Pareto-optimality. Hence the $Comb()$ operator implements the commonly accepted Pareto-optimal ordering over multiple objectives. If a different ordering is desired then the $\oplus$ operator can be redefined to implement the desired ordering.

6.2.2 Distributed Constraint Optimization Problems

In Constraint Satisfaction Problems (CSPs), the constraints within the problem and the criteria that the solution must satisfy can be viewed as one and the same. However Constraint Satisfaction/Optimisation Problems (CSOPs) allow both objectives and constraints as criteria that the solution must satisfy, so the three concepts must be addressed separately. Hence the core concept in CSOPs is to assign a value to each of the variables from the variables' domain such that not only the constraints are satisfied but also the set of optimization criteria are satisfied. Due to the structure of a CSOP, objectives cannot be realized directly, they must be realized via intermediate soft constraints. For example, in valued CSOPs, the criteria is to minimize (or maximize) the total value of the soft constraints. This requires soft constraints that return a real number, rather than true or false. Using optimization criteria such as minimize or maximize has other difficulties, as a naive solver must explore the entire solution space to determine if the criteria are satisfied. Furthermore, the relaxation of an over-constrained problem yields a CSOP. Over-constrained problems can be converted by either

relaxing the constraints or the solutions. Relaxing the constraints yields an optimization criteria on the constraints (satisfy the maximum number of constraints). While relaxing the solutions means that some variables can be left without a value assigned, leading to the objective: maximize the number of variables assigned a value. If there exists more than one optimization criterion, then often there is no solution that satisfies all of them. So, the criteria are normally relaxed to find a Pareto-optimal solution.

A unified formulation must also support the concept of agents. The concept of agents is required for distributed problems. An agent has knowledge of a subset of the entire problem and can only affect a subset of the sub-problem it knows. There are several common justifications (such as resource limits, privacy concerns and communication capacity) for limiting an agent's knowledge to only a subset of the entire problem. Furthermore, resource and communication limitations can lead to efficiencies, such as a reduction of memory requirements for each agent and a reduction in required messages to keep the knowledge of all agents synchronized. An agent may also not know of variables and constraints that represent privileged information held by another agent.

Definition 22: *A Semiring-Based Distributed Constraint Satisfaction/Optimization Problem (SD-COP) is a tuple* $\langle \mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{V}, \mathcal{C} \rangle$ *where*

- $\mathcal{A}$ *is a non-empty set of agents. An agent is a pair* $\alpha = \langle R_\alpha, W_\alpha \rangle$. R_a *is a set of variables for which the agent has* **read privileges**, *and* $W_\alpha \subseteq R_\alpha$ *is a set of variables for which the agent has* **write privileges**.
- $\mathcal{X}$ *is a set of variables.*
- $\mathcal{D}$ *is a set* $\{D_1, \ldots, D_n\}$ *where* $n = |\mathcal{X}|$ *and each* D_i *is a set of values to be assigned to a variable (the domain).*
- $\mathcal{V} = \langle V, \oplus, \otimes \rangle$ *is an idempotent semiring utilized to evaluate variable assignments.*
- $\mathcal{C}$ *is a non-empty set of constraints, where each constraint* c_i *is a pair* $\langle def_i, con_i \rangle$ *where* def_i *is a function* $def_i : (\bigcup \mathcal{D})^k \longrightarrow V$ *(where* $k = \mid con_i \mid$*) and* $con_i \subseteq \mathcal{X}$ *is the* **signature** *of constraint* c_i *(i.e., the set of variables referred to in that constraint).*

If an agent α *has write privileges for a variable* v, α *must also have read privileges for all variables which share a constraint with* v *i.e.* $\forall v \in W_\alpha, \forall c_i \in \mathcal{C}$, *if* $v \in con_i$, *then* $con_i \subset R_\alpha$.

The set of agents refers to the agents that must co-operate to solve the problem. There is of course much more to an agent than its privileges, such as the amount of resources available or the 'temperament' of the agent. We do not consider those attributes (and similar ones) of agents as they only apply to solving the problem, not to the description of the problem.

Read privileges serve a dual purpose. The primary purpose is to identify which agents must know the value assigned of a variable. The secondary purpose is to determine the required communication links between agents. Knowledge of variable assignments is only one aspect of privacy. Often it is also desirable to ensure that other agents can not discover the constraints on an agent's variables. The flow of such information is entirely dependent on the algorithm and so is outside the scope of this work.

As the required communication is already prescribed by read privileges, the write privileges are purely to determine which agents have permission to allocate a value to a variable. In most situations,

it is acceptable for there to be variables in the problem for which no agent has write privileges. These can be viewed as hard constraints or environmental constants. This concept is particularly useful in a mixed-initiative setting whereby assignments made by the human operator should not and cannot be overruled by the machine solver.

The explicit assertion of read/write privileges is unique to this formulation. Traditional formulations implicitly prescribe global read privileges while maintaining only local write privileges. We feel that this approach over simplifies real world problems. Such an assumption creates additional communication overhead to maintain synchronization and impedes on the scalable and distributed nature of DCSOPs. Furthermore, to honestly reflect the privacy property in DCOPs, the read/write privileges prescribes access control machinery to control access to information. However, note that the traditional formulation (global read/local write) approach is a special case for our formulation where agents are given read privileges to all variables. A more detailed discussion on this deviation is covered in section 6.3. It also allows for parameters (variables for which no agent has write access) to be naturally represented within the problem and allows the community to explore problems without the simplifying assumption that agents have exclusive control over variables. Designing algorithms which support allowing multiple agents to write to a variable presents new challenges. The algorithm must have some mechanisim for either managing access to the variable or resolving the resulting inconsistencies.

Variables are identified by a unique name. All variables share the same domain, rather than having separate domains as per common practice. The common domain can be viewed as the union of all individual domains, with unary constraints on the individual variables restricting which subset of the domain values can be assigned to the variables. This approach generalizes existing practice. The idea of having the domain of each variable defined by unary constraints was discussed by Ross et. al. [55]

Our use of functions that return a value from an idempotent semiring allows for the representation of a range of different forms of constraints. In classic satisfaction problems, the semiring $\langle \{\text{True}, \text{False}\}, \vee, \wedge \rangle$ is sufficient to reflect the satisfaction of the constraints. For valued constraint optimization problems, the semiring $\langle \Re, \min, + \rangle$ is suitable for minimization problems and $\langle \Re, \max, + \rangle$ is suitable for maximization problems. In the cases where there are multiple objective functions, each objective is represented by its own idempotent semiring. The $Comb()$ operator can then be utilized to combine the individual semirings. Using this approach to combine the different objectives naturally leads to a search for a Pareto-optimal solution. By using this approach we do not impose an ordering on the objectives, however an ordering can be added if so desired. Furthermore, such an approach also allows for all the different types of constraints to be combined.

We also distinguish between two classes of constraints: local constraints and shared constraints. A local constraint is known by only one agent and can only be evaluated by that agent. For this to occur one agent must have write privileges to all of the variables in con_i (refer to definition 6.2.2). Shared constraints are known by all agents which have write privileges to a variable in the constraints signature and can be evaluated by any one of those agents. This occurs when more than one agent has write privileges to one of the variables in con_i (refer to definition 6.2.2).

Definition 23: *Given an SDCOP* $\langle \mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{V}, \mathcal{C} \rangle$, $\mathcal{S}$ *is the set of all possible assignments to variables. Each assignment is a set of variable-value pairs where no variable appears more than once. We will refer to an assignment* $s \in \mathcal{S}$ *as a* **complete assignment** *if it assigns a value to every variable in* $\mathcal{X}$. *For some assignment* $s \in \mathcal{S}$, *we will use* $s \downarrow_X$ *to denote the* projection *of the assignment*

to a set of variables X (i.e., the subset of s that refers to variables in X). Given an assignment $s = \langle\langle x_1, v_1\rangle, \langle x_2, v_2\rangle, \ldots, \langle x_k, v_k\rangle\rangle$, $val(s) = \langle v_1, v_2, \ldots, v_k\rangle$ and $var(s) = \langle x_1, x_2, \ldots, x_k\rangle$.

The different criteria are reflected in the definition of a solution to a SDCOP. The concept of a solution to a satisfaction problem is generalized in our definition of an acceptable solution.

Recall that our unified definition of a CSP is agnostic to the solving method, so while it has been shown [52] that combining semirings is not sufficient for solving multi-objective problems using inference based methods [5, 13, 14, 17, 47] (specifically the operators $\bigoplus$ and $\bigotimes$). However, combining semirings is sufficient for search based methods [6, 58], which are common in the DCOP literature. For this reason we define the solutions to a SDCOP in terms of the properties they must satisfy, which are independent of the solving method and the semiring used to evaluate assignments. Furthermore, we utilize a notion of acceptable threshold to define an acceptable solution to an optimization problem. The idea of an acceptable threshold on an optimization problem has been proposed by J. Larrosa [28].

Definition 24: *An* **acceptable solution** *to a SDCOP with $\mathcal{C} = \{c_1, \ldots, c_n\}$ is a complete assignment s such that $def_1(val(s \downarrow_{con_1})) \otimes \ldots \otimes def_n(val(s \downarrow_{con_n})) \preceq_V v$, where $v \in V$ is a minimum threshold on the value of the solution.*

The solution to a classic satisfaction problem can be represented as an acceptable solution with a threshold of True. An acceptable solution is also useful to represent problems where the optimal solution is not required or it is too expensive to search for the optimal solution.

Definition 25: *An* **optimal solution** *to a SDCOP with $\mathcal{C} = \{c_1, \ldots, c_n\}$ is a complete assignment s such that there does not exist another complete assignment s' where $def_1(val(s' \downarrow_{con_1})) \otimes \ldots \otimes def_n(val(s' \downarrow_{con_n})) \prec_V def_1(val(s \downarrow_{con_1})) \otimes \ldots \otimes def_n(val(s \downarrow_{con_n}))$ (note: $a \prec_V b$ iff $a \preceq_V b$ and $b \not\preceq_V a$).*

Theorem 9: *At least one optimal solution exists for any SDCOP.*

Proof: *The associative property of $\oplus$ means that the ordering $\preceq_V$ derived from $\oplus$ is transitive. Due to the ordering being transitive, cycles can not exist within the ordering. Because there are no cycles there must be at least one abstract value which is not dominated by another abstract value.*

The constraints map each assignment to exactly one abstract value, which is used to order the assignments. As such, the ordering over assignments is also transitive. Therefore there must be at least one complete assignment which is not dominated by another complete assignment.□

Note that the definition of an optimal solution is equivalent to a non-dominated solution, as such the set of all optimal solutions is the Pareto-frontier. If an acceptable solution (for a given threshold) does not exist, one way to get a solution which is acceptable is to relax the concept of a solution. This is done by not requiring a complete solution to the SDCOP.

Definition 26: *Given a constraint $c_i = \langle def_i, con_i\rangle$ the projection of c_i onto a set of variables X, $c_i \downarrow_X = \langle def'_i, con_i \cap X\rangle$. If $con_i \subseteq X$ then c_i is returned unchanged. def'_i is defined such that for a given input V it returns one of the values def_i can return for the input V expanded to include values for the other variables in con_i.*

There are two different intuitions about which value to return in the projected version of the constraint. First, the projected constraint should return the best possible value using the assigned variables. Second, the projected constraint should return the worst possible value using the assigned variables. It is possible that there is more that one best or worst value, as the semiring allows a partial ordering. In this case the choice between the best or worst values is arbitrary.

Definition 27: *Given a set of constraints $\mathcal{C} = \{c_1, \ldots, c_n\}$ and an assignment s, let $\mathcal{C}' =$*

$\{c_1 \downarrow_{var(s)}, \ldots, c_n \downarrow_{var(s)}\}$. *s is a* **relaxed solution** *with reference to the constraints* $\mathcal{C}'$ *iff* $def_1(val(s \downarrow_{con_1}))\otimes\ldots\otimes def_n(val(s \downarrow_{con_n})) \preceq_V v$, *where v is a minimum threshold on the value of the solution and there does not exist another assignment* s' *where* $|s'| > |s|$ *and* $def_1(val(s' \downarrow_{con_1})) \otimes \ldots \otimes def_n(val(s' \downarrow_{con_n})) \preceq_V v$.

There are many other ways to define a solution to a SDCOP, such as a k-optimal solution [45]. These other solution concepts can be easily modified for the SDCOP structure.

6.2.3 Example Instantiations

In this section, we will illustrate the usefulness of our framework by the SDCOP instances that correspond to several common DCOPs. We will do so by presenting three instantiations: a CSP instance, a DCOP instance and a DCOP variation. Bistarelli's c-semiring framework [9] is one of the commonly accepted frameworks for generalizing CSP problems. Both Bistarelli's c-semiring framework and our SDCOP framework are based on using a variation of a semiring to order the solutions. As highlighted earlier, CSPs are special instances of DCOPs. Hence, it is befitting to illustrate the generality of SDCOP by demonstrating that Bistarelli's c-semiring framework is a special instance of an SDCOP.

Theorem 10: *Any problem represented as a c-semiring can be represented as an idempotent semiring. Any idempotent semiring which also satisfies the following properties can be represented as a c-semiring.*

- $\bot_i \in V$ *is the absorbing element of* $\otimes$ *and the unit element of* $\oplus$.
- $\top_i \in V$ *is the unit element of* $\otimes$ *and the absorbing element of* $\oplus$.

Proof: *We consider two representations of a constraint optimisation problem to be equivalent iff the solutions of the representations are equal. The solution to a constraint optimisation problem is defined in terms of the* $\oplus$ *and* $\otimes$ *operators. Therefore it is sufficient to show that the* $\oplus$ *and* $\otimes$ *operators in c-semirings and idempotent semirings have the same properties.*

Given a c-semiring $\langle V_c, \oplus_c, \otimes_c, \bot_c, \top_c\rangle$ *and an idempotent semiring* $\langle V_i, \oplus_i, \otimes_i, \bot_i, \top_i\rangle$ *as described above, we show that the two are equivalent.*

V_i *and* V_c *are equivalent by definition, as are* $\otimes_i$ *and* $\otimes_c$. $\bot_i, \bot_c$ *are the absorbing elements of* $\otimes_i, \otimes_c$ *respectively, as per the definition. Similarly* $\top_i, \top_c$ *are the unit elements of* $\otimes_i, \otimes_c$ *respectively.*

It remains to show that $\oplus_i$ *is equivalent to* $\oplus_c$. $\oplus_c$ *is defined over a set of operands, while* $\oplus_i$ *is strictly a binary operator, as such the behaviour of* $\oplus_c$ *with zero or one operands is not relevant.* $\oplus_i$ *is commutative, associative and idempotent, so it follows that* $\oplus_i(\{v_i, v_{i+1}, \ldots, v_j\}) = v_i \oplus v_{i+1} \oplus_i \cdots \oplus_i v_j$ *for all sets of indices* $i, i+1, \ldots, j$. *It is shown in Bistarelli's paper that* $\oplus_c$ *is idempotent, commutative and associative. By specifying that* $\top$ *is the absorbing element of* $\oplus_i, v_1 \oplus v_2 \oplus_i \cdots \oplus_i v_n = \top$ *for* $v_1, \ldots, v_n \in V$, *which is equivalent to* $\oplus_c(V) = \top_c$. $\oplus_i(\mathcal{V}_\rangle) = \top$ *will be the case iff* $\top$ *is the absorbing element of* $\mathcal{V}_\rangle$. $\square$

We have just shown that the idempotent semiring used in our framework is more general than the c-semiring used in Bistarelli's framework [9]. Therefore, all the constraint schemes which can be represented in Bistarelli's framework can also be represented in our framework. Namely classical, fuzzy, probabilistic, weighted (minimization), set based, and multi-objective constraint optimization problems.

There exist several formulations of DCOPs [6, 17, 39, 46, 58]. All of these formulations capture the important aspects of DCOPs equally well. We will focus on the instantiation of Petcu's formulation [46] into SDCOP as it is the most formal definition.

Petcu [46] defines a COP as follows:

Definition 28: *[46] A* discrete *constraint optimization problem (COP) is a tuple* $\langle \mathcal{X}, \mathcal{D}, \mathcal{R} \rangle$ *such that:*

- $\mathcal{X} = \{X_1, \ldots, X_n\}$ *is a set of* variables *(e.g. start times of meetings);*
- $\mathcal{D} = \{d_1, \ldots, d_n\}$ *is a set of discrete, finite variable* domains *(e.g. time slots);*
- $\mathcal{R} = \{r_1, \ldots, r_m\}$ *is a set of* utility functions*, where each* r_i *is a function with the scope* $(X_{i_1}, \ldots, X_{i_k}), r_i : d_{i_1} \times \cdots \times d_{i_k} \longrightarrow \Re$*. Such a function assigns a* utility (reward) *to each possible combination of values of the variables in the scope of the function. Negative amounts mean costs. Hard constraints (which forbid certain value combinations) are a special case of utility functions, which assign* 0 *to feasible tuples, and* $-\inf$ *to infeasible ones;*

Definition 29: *[46] A discrete distributed constraint optimization problem (DCOP) is a tuple of the following form:* $\langle \mathcal{A}, \mathcal{COP}, \mathcal{R}^{ia} \rangle$ *such that:*

- $\mathcal{A} = \{A_1, \ldots, A_k\}$ *is a set of agents (e.g. people participating in meetings);*
- $\mathcal{COP} = \{COP_1, \ldots, COP\}$ *is a set of disjoint, centralized COPs; each* COP_i *is called the local sub-problem of agent* A_i*, and is owned and controlled by agent* A_i*;*
- $\mathcal{R}^{ia} = \{r_1, \ldots, r_n\}$ *is a set of inter-agent utility functions defined over variables from several different local sub-problems* COP_i*. Each* $r_i : d_{i_1} \times \cdots \times d_{i_k} \longrightarrow \Re$ *expresses the rewards obtained by the agents involved in* r_i *for some joint decision. The agents involved in* r_i *have full knowledge of* r_i *and are called 'responsible' for* r_i*. As in a COP, hard constraints are simulated by utility functions which assign* 0 *to feasible tuples, and* $-\infty$ *to infeasible ones;*

Further, Petcu defines a solution as:

Definition 30: *[46] The goal is to find a complete instantiation* $\mathcal{X}$ *for the variables* X_i *that* maximizes *the sum of utilities of individual utility functions.* The definition of a solution provided by Petcu [46] corresponds to the definition of an optimal solution within SDCOP. This definition of a DCOP corresponds to the following SDCOP: **Theorem 11:** *Petcu's definition of a DCOP is a SDCOP* $\langle \mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{V}, \mathcal{S}, \mathcal{C} \rangle$ *with the additional properties:*

- *Exactly one agent has write access for each variable.*
- $\bigcup \mathcal{D}$ *is finite.*
- $\mathcal{V} = \langle \Re, max, + \rangle$*.*

Proof: *The two formulations are equivalent iff every problem that can be described by Petcu's definition has an equivalent definition within the SDCOP and every problem that can be described by the SDCOP has an equivalent definition within Petcu's definition.*

We start by showing that every problem that can be described by Petcu's definition has an equivalent definition within the SDCOP. It is clear that both formulations support the concept of agents. Petcu's

definition allows an agent to control a set of variables described as a sub-problem while SDCOP directly specifies the variables that an agent controls. Petcu's use of real numbers to measure utility can easily be represented by the idempotent semiring $\langle \Re, max, + \rangle$

Now we show that every problem that can be described by the SDCOP has an equivalent definition within Petcu's definition. Both definitions support the concept of agents. The first additional restriction on a SDCOP results in each agent controlling a distinct set of variables. The variables which an agent controls form the agent's sub-problem in Petcu's definition. There is no formal concept of read privileges in Petcu's definition, instead it is simply assumed that if an agent knows a constraint, it has read privileges for all variables in the constraint, which is the minimal read privileges required for SDCOP. SDCOP uses a single domain for all variables, as opposed to separate finite domains for each variable. The second restriction limits SDCOP to a finite domain, which can be transformed into variable specific domains by first assigning each variable the entire domain, then propagating all the unary constraints on that variable. The resulting domain is specific domain for that variable. After this the unary constraints can be removed. Restriction three limits SDCOP to using real numbers as the utility, which is equivalent to Petcu's use of real numbers. Finally the use of n-ary constraints in SDCOP is almost identical to the constraints in Petcu's definition.□

Recently, Grinshpoun et al. proposed the Asymmetrical DCOP (ADCOP) model [21]. In this model, the utility gained by each agent participating in a constraint is tallied separately, however the objective is to minimize (or maximize) the sum of the utilities of all agents. The motivation is to preserve the privacy of each agent regarding its local utility. Grinshpoun et al. [21] define a DCOP as follows:

Definition 31: *[21] A DCOP is a tuple* $\langle \mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{R} \rangle$. $\mathcal{A}$ *is a finite set of agents* $A_1, A_2, \ldots, A_n$. $\mathcal{X}$ *is a finite set of variables* $X_1, X_2, \ldots, X_m$ *. Each variable is held by a single agent (an agent may hold more than one variable).* $\mathcal{D}$ *is a set of domains* $D_1, D_2, \ldots, D_m$ *. Each domain* D_i *contains a finite set of values which can be assigned to variable* X_i *.* $\mathcal{R}$ *is a set of relations (constraints). Each constraint* $C \in R$ *defines a non-negative cost for every possible value combination of a set of variables, and is of the form:*

$$C : D_{i1} \times D_{i2} \times \cdots \times D_{ik} \longrightarrow \Re^{+} \tag{6.1}$$

Grinshpoun et al. [21] define an ADCOP as follows:

Definition 32: *[21] An ADCOP is defined by the following tuple* $\langle \mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{R} \rangle$ *, where* $\mathcal{A}$*,* $\mathcal{X}$ *, and* $\mathcal{D}$ *are defined in exactly the same manner as in DCOPs. Each constraint* $C \in R$ *of an asymmetric DCOP defines a set of non-negative costs for every possible value combination of a set of variables, and takes the following form:*

$$C : D_{i1} \times D_{i2} \times \cdots \times D_{ik} \longrightarrow \Re^{+^k} \tag{6.2}$$

Grinshpoun's definition of a DCOP is comparable to Petcu's, which we have already shown to be an instance of of an SDCOP. It remains to show that the constraint definition in an ADCOP is an instance of an idempotent semiring.

Theorem 12: *Asymmetrical constraints as defined in a ADCOP can be represented as the an idempotent semiring* $\langle V, \otimes, \oplus \rangle$ *where: V is the Cartesian product of the set of utilities for each agent.*

$\Re_+^n$, given agents $a_1, \ldots, a_n$, $\otimes$ is defined as $\otimes(v, v') = \langle v_1 + v'_1, v_2 + v'_2, \ldots, v_n + v'_n \rangle$ and $\oplus$ is defined as

$$\oplus(v, v') = \begin{cases} v & \Sigma_n^1 v \leq \Sigma_n^1 v' \\ v' & \textit{otherwise} \end{cases} \tag{6.3}$$

In addition, if there does not exist a variable which is owned by an agent α, in the signature of a constraint c, then the cost for α in the result of the constraint c must be zero.

Proof:

The ADCOP definition allows the cost for each agent to be recorded separately, by having one integer/real number for each agent, the idempotent semiring described also allows the cost for each agent to be recorded separately. The aggregation operator in ADCOP is sum, with the cost for each agent aggregated separately, this is reflected in the $\otimes$ operator. The comparison operator in ADCOP is a utilitarian minimization operator, this is reflected in the $\oplus$ operator.

The idempotent semiring always includes a cost for all agents, even if the agent is not involved in the constraint, thus it may represent constraints that can not be represented in an ADCOP. The restriction on the cost for an agent if the agent is not part of the constraint prevents this case. □

6.3 Alternate Modelling Approaches

There is a spectrum of different ways to structure an SDCOP. At one end, each agent has exclusive control over a set of variables and shares the constraints. On the other end, each agent has exclusive control over a set of constraints and shares the variables. Existing DCOP research has focused exclusively on on the former problem structure.

By allowing shared variables in a problem, some problems can be represented and solved more efficiently. On example of this is meeting scheduling problems [32]. It is common to give every agent who must attend the same meeting a different variable, representing the time of the meeting, with an equality constraint between them, this is the Private Events as Variables approach (PEAV). SDCOP permits the same problem to be modeled such that all agents who must attend a meeting have write privileges to a single variable representing the time of the meeting, as per the Events as Variables approach (EAV). This permits the problem to be written with less variables and less constraints, reducing the complexity of solving the problem. In addition, Grinshpoun et al. [21] point out that any consistent solution to a problem modeled using PEAV must be a local optimal. Which reduces the effectiveness of local search algorithms when applied to these problems.

Changing equality constraints to a single shared variable does change the semantics of the problem slightly. Equality constraints can be violated or relaxed, while the same constraints expressed as a shared variable can not be violated. Consider a problem where a group of people have chartered a small plane. If one of the people who wishes to board the plane turns up late, the plane still departs as planned and all the people who turned up on time can board. As such modelling the time people intend to board the plane as a shared variable doesn't make sense, for it would not allow people to arrive early or late. On the other hand, deciding when the plane will depart should be a shared variable, as there is only one plane it can not depart at two different times.

The discussion between shared variables and shared constraints highlights the question of an agents responsibility. If it is a new system, then the responsibilities of each agent can be determined

at design time. However if an existing system is being retrofitted with agent technology, then the responsibilities of each agent are often already defined. It is common practice to ensure a complete separation of concerns when designing agent systems, as this makes the system easier to design, increases the efficiency of the system and helps to protect each agents privacy. On the other hand, having overlap between agents increases the robustness of the system. For instance, if two agents have write privileges to each variable and at least two agents know each constraint, then the system has n+1 redundancy. Having redundancy within the system and an efficient way of identifying and restarting failed agents will result in a system that is robust against failures in the environment. Future work may be able to minimize the amount of duplicated effort between redundant agents, reducing the efficiency loss.

To highlight the intended use of a SDCOP, consider a consortium of three small distributors who operate from the same area. These distributors have formed a consortium to increase their overall efficiency. Efficiency is gained by sharing resources and reducing competition between their companies. Each distributor maintains their own truck, and there is another truck which is shared between the three distributors (note that the shared truck is an instance of a committee problem as described in the introduction). So long as two distributors wish to transport product to the same destination, the can use the shared truck and share the transport costs. For simplicity, we assume that all the distributors supply the same product, they all have the same amount of product to distribute, and the capacity of each truck is unlimited. Each distributor earns money by matching what they supply to a destination with the demand at that destination. This results in the following SDCOP:

- $\mathcal{A} = \{\text{agent-1}, \text{agent-2}, \text{agent-3}\}$.
- $\mathcal{X}$ = {truck-1-d, truck-1-q, truck-2-d, truck-2-q, truck-3-q, truck-3-d, s-truck-d, s-truck-q1, s-truck-q2, s-truck-q3, demand-1, demand-2, demand-3, demand-4}.
- $R_{\text{agent-1}}$ = {truck-1-d, truck-1-q, s-truck-d, s-truck-q1, demand-1, demand-2, demand-3, demand-4}
- $R_{\text{agent-2}}$ = {truck-2-d, truck-2-q, s-truck-d, s-truck-q2, demand-1, demand-2, demand-3, demand-4}
- $R_{\text{agent-3}}$ = {truck-3-d, truck-3-q, s-truck-d, s-truck-q3, demand-1, demand-2, demand-3, demand-4}
- $W_{\text{agent-1}}$ = {truck-1-d, truck-1-q, s-truck-d, s-truck-q1}
- $W_{\text{agent-2}}$ = {truck-2-d, truck-2-q, s-truck-d, s-truck-q2}
- $W_{\text{agent-1}}$ = {truck-3-d, truck-3-q, s-truck-d, s-truck-q3}
- $\mathcal{D} = \{0, 1, \ldots, 100, \text{destination-1}, \text{destination-2}, \text{destination-3}, \text{destination-4}\}$
- $\mathcal{V} = \langle \Re, \max, + \rangle$
- $\mathcal{C} = \{...\}$

As all the distributors must agree on the destination of the shared truck (s-truck-d), all of the distributors have write access to that variable. Note also the use of 'parameter' variables (demand-1 to demand-4) which all agents can read, but not write to.

6.4 Meta-SDCOP

What is the purpose of Meta-SDCOP? add aditional properties to SDCOP show how SDCOP supports the more specialised instances of DCOPs show how SDCOP supports useful instances of DCOPs which other formalisations can not

Is it actually required? What are the interesting properties of meta-SDCOPs? defines a class of problems

Meta-SDCOPs are a class of transformations that can be applied to an SDCOP. They are used to add additional properties to an existing SDCOP. The intention is to add (and make explicit) the properties that differentiate a sub-class of (D)COP problems, such as an open constraint problem, from a general (D)COP. These properties include such things as self modifying abilities to allow over-constrained problems to be relaxed automatically or to support open CSPs. This differs from dynamic problems because in dynamic problems the changes are imposed by the environment, while in meta-SDCOPs the solver has control over the changes. Some of the transformations presented here are straight-forward adaptions of existing work (disjunctive temporal CSPs [40]), while others require a different perspective on existing work to fit them into a meta-SDCOP (open constraint problems [55]).

The transformations are created by modifying each existing constraint to include a control variable. All agents who have write privileges to one of the variables in the original constraint get read and write privileges to the control variable. The behaviour of the constraint is then determined by the value assigned to the control variable. The domain of each control variable and any relationships between control variables are then defined via new constraints. This approach allows the domain of variables to be modified (as the domain is defined by a unary constraint) and existing constraints to be modified or removed. In order to add constraints a null constraint[1] must already exist, which can then be modified to add the constraint.

We choose to modify the original problem rather than creating a separate problem because the separate problem would be ill-defined. Specifically, the solution to the meta-SDCOP is tied to the original problem. The optimal solution to the meta-SDCOP is the solution which, when transformed to a solution to the original SDCOP, is optimal. As the meta-SDCOP can not be solved without reference to the resulting SDCOP, it is more elegant to combine the meta-SDCOP and resulting SDCOP into a single SDCOP.

Definition 33: *Given a SDCOP* $\langle \mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{V}, \mathcal{C} \rangle$ *A* **meta-SDCOP** $\mathcal{M}$*(SDCOP) is a SDCOP* $\langle \mathcal{A}', \mathcal{X}', \mathcal{D}', \mathcal{V}', \mathcal{S}', \mathcal{C}' \rangle$ *where:*

- $\mathcal{A}' = \{\langle R_a \cup \{f_1(c_i) | \forall c_i \in C \text{ where } a \in con_i\}, W_a \cup \{f_1(c_i) | \forall c_i \in C \text{ where } a \in con_i\}\rangle | \forall a \in \mathcal{A}\}$
- $\mathcal{X}' = \mathcal{X} \cup \{f_1(c_1), f_1(c_2), \ldots, f_1(c_n)\}$
- $\mathcal{D}' = \mathcal{D} \cup f_2(c_1) \cup f_2(c_2) \cup \cdots \cup f_2(c_n)$
- $\mathcal{V}'$ *is an idempotent semiring.*
- $\mathcal{C}' = \{f_3(c) | \forall c \in \mathcal{C}\} \cup (\forall X \in 2^{\mathcal{X}'}, \bigcup f_4(X))$
- $f_1(c)$ *takes a constraint as input and returns a control variable for that constraint.*

[1] The definition of the null constraint depends on the semiring being used. For CSPs it would always return true, while for valued constraints it would always return 0.

- $f_2(c)$ *takes a constraint* $\langle def, con\rangle$ *as input and returns a set of functions* $f : X \longrightarrow V$ *with signature con.*
- $f_3(c)$ *takes a constraint* c *with signature* s *as input and returns a constraint with signature* $s \cup f_1(c)$*.*
- $f_4(X)$ *takes a set of variables and returns a set of constraints with signature* X *(possibly* $\emptyset$*). These constraints must include unary constraints restricting the domain of the control variables,* $\{f_1(c)|\forall c \in \mathcal{C}\}$*.*

The meta-SDCOP created from a given SDCOP is defined by four functions. The first, $f_1(c)$ defines a new variable for each of the constraints in the original SDCOP. Each of these new variables is writable and readable by all agents who have write access to one of the variables in the original constraint. The domain of each of these new variables is defined by the function $f_2(c)$. Each domain is a set of functions that could be used in place of the function in c. The function $f_3(c)$ replaces the constraints in the original problem with constraints that take the control variable as a parameter. For each new constraint $c'_i = \langle def'_i, con'_i\rangle$, $con'_i = con_i \cup f_1(c)$ and def'_i is defined such that it returns the same value that the function selected from the domain of $f_1(c)$ would return if given the same input projected over con_i. The value assigned to the variable $f_1(c)$ defines the actual constraint between variables con_i, which is used when the constraint must be evaluated. Finally, the function $f_4(X)$ defines the constraints on the new variables $\mathcal{X}'$. This includes unary constraints such as the restrictions on a variable's domain and n-ary constraints which represent relations between variables. Assuming that $\oplus$ has an identity element $\oplus_I \in V$, the constraints to restrict the domain of the control variables can be written as follows:

$$\left\langle def(x) = \begin{cases} \oplus_I & x \in f_2(c) \\ \bot & \text{otherwise} \end{cases}, f_1(c) \right\rangle \tag{6.4}$$

Following are three different sub-classes of a meta-SDCOP.

The first one allows the constraints in the SDCOP to be relaxed, by replacing them with different constraints. Precisely what is meant be 'relaxed' in this context is defined by the function f_2, which provides the replacement constraints, and the function f_4, which provides the ordering over the replacement constraints via the new constraints it defines.

Definition 34: *A* **Relaxation meta-SDCOP** $\mathcal{M}$*(SDCOP) is a meta-SDCOP with the additional properties that:*

- $\mathcal{V}' = Comb(\mathcal{V}, \mathcal{V}_2)$ *where* $\mathcal{V}_2$ *is a c-semiring used to measure the degree of relaxation of the problem.*
- $f_2(x)$ *takes a constraint as input and returns a set of all possible relaxations for that constraint.*
- $f_4(X)$ *takes a set of variables and returns a set of constraints with signature* X *(possibly* $\emptyset$*). In each of these constraints,* $\top$ *of* $\mathcal{V}_2$ *is the value returned when the constraints are equivalent to the original constraints.*

There are two separate objectives in this problem, one to optimize the solution and one to minimize the amount the problem is relaxed. The use of a c-semiring to measure the amount of relaxation ensures

that the values form a lattice with $\top$ as the most preferred value (no constraints relaxed) and $\bot$ as the least preferred value (constraints fully relaxed).

Theorem 6.4 shows that the relaxation is correct, by which we mean that if the original problem is not over-constrained, then a solution to the new problem (projected onto the smaller set of variables) is also a solution to the original problem.

Theorem 13: *Given a SDCOP d, if s is an acceptable solution with threshold $V = \langle v_1, \top \rangle$ for $\mathcal{R}(d)$ then $s \downarrow_{\mathcal{X}}$ is an acceptable solution with threshold v_1 for d.*

Proof: *The use of a c-semiring for $\mathcal{V}_2$ and the definition of $f_4(X)$ ensures that solutions which have been relaxed less are preferred over solutions that have been relaxed more. To meet the threshold of $\langle v_1, \top \rangle$ requires that none of the constraints have been relaxed. The way $\mathcal{C}'$ is defined, the non-relaxed version of each constraint is equivalent to the original constraint. Therefore a solution which satisfies all the constraints in $\mathcal{R}(d)$ also satisfies all the constraints in d.* □

The second meta-SDCOP models a common approach to solving problems with disjunctive constraints. In a disjunctive temporal CSP [40] the value assigned to each variable must fall within one of several disjunctive ranges. Solving these problems using a meta-SDCOP involves first identifying which of many disjuncts each variable will satisfy, then selecting a value that falls within the selected range.

Definition 35: *A* **disjunctive temporal meta-SDCOP** $\mathcal{T}(SDCOP)$ *is a meta-SDCOP with the additional properties that:*

- $\mathcal{V}' = \mathcal{V}$
- $f_2(x)$ *takes a constraint as input and returns a set of all the disjuncts in the constraint.*

In open constraint problems it is assumed that there is a significant cost involved in identifying the domain of variables. Because of this cost it is assumed that the domain of each variable in the problem is a subset of the true domain. Further the domain of a variable in the problem can be expanded (for some cost) by querying an outside agent. This is very similar to the relaxation meta-SDCOP above, except only the unary constraints may be relaxed. However, it can not capture the restriction that a variable's domain can only be expanded, not tightened.

Definition 36: *An* **Open meta-SDCOP** $\mathcal{M}(SDCOP)$ *is a meta-SDCOP with the additional properties that:*

- $\mathcal{V}' = Comb(\mathcal{V}, \mathcal{V}_2)$ *where $\mathcal{V}_2$ is a c-semiring used to measure the degree of relaxation of the problem.*
- $f_2(x)$ *takes a constraint as input and if it is a unary constraint, returns the set of constraints which describe all possible domains for the associated variable, otherwise it returns a singleton set containing the original constraint.*
- $f_4(X)$ *takes a set of variables and returns a set of constraints with signature X (possibly $\emptyset$). In each of these constraints, $\top$ of $\mathcal{V}_2$ is the value returned when the constraint describes the tightest possible domain for the associate variable.*

6.5 Dynamic Problems

The SDCOP structure is good for describing problems that do not change over time, unfortunately, many real life problems do change over time. In these cases it is beneficial to consider each change as a different state of an overall problem, rather than individual problems. This approach allows knowledge gained when solving one state of the overall problem to be reused when solving the next state It also allows the solver to reason about the changes between states. The overall problem, including the relationships between states, is modelled as a Dynamic SDCOP (DynSDCOP). Each state of the overall problem is described as a SDCOP.

Most approaches to modelling dynamic problems only consider the DynSDCOP solver to be acting in a decision support role. With our formalization of dynamic problems, we wish to account for the DynSDCOP solver acting on the environment either directly or indirectly. For the purposes of this discussion, we assume that there is a separate agent with the power to adopt solutions proposed by the DynSDCOP solver and to modify the SDCOP being solved in response to changes in the environment.

To correctly reason in this environment, the DynSDCOP solver must know both the value assigned to each variable in the solution currently being explored and the value assigned in the solution which has been adopted. This is based on the idea of solution stability [66, 67, 68], where there is a cost associated with changing the value of a variable.

Petcu [48] extends the basic idea of solution stability with deadlines. Each variable in the problem is annotated with either a hard or a soft commitment deadline, when the deadline is reached the variable is considered to be committed to its current value. If it is a hard deadline the value assigned to the variable can not be changed, if it is a soft deadline then the value can still be changed, but at an extra cost.

In a perfect world the DynSDCOP solver will know what the future problems to be solved are. It will then be able to chose a solution that simultaneously minimizes the cost to transition from the previous solution to the current solution, maximizes the utility of the current solution and minimizes the cost to transition from the current solution to the future solution. In the real world the best we can do is guess possible future problems and the probability associated with each of them.

Conjecture 1: *The solution to a single problem within a dynamic SDCOP depends on both the current problem and the sequence of previous problems (the history).*

Assuming that the previous state of the problem has no direct impact on the current problem, a single solution exists and the solving algorithm has enough time to prove that the solution it has found is the only solution, then the solution will be the same irrespective of the previous states of the problem. Those assumptions do not hold in general. Keeping the assumption that the previous problem has no direct impact on the current problem, the previous problems and the algorithm chosen will introduce a search bias. The nature of the search bias depends heavily on the solver used, but includes things like minimum change from the previous solution, variable ordering currently used and nogoods [56] discovered. This search bias will make the algorithm more likely to choose a particular solution out of the options available. If you assume that the previous problem does have a direct impact on the current state, which is often the case when previously made decisions have to be undone, Then it is obvious that the previous problems (or more accurately, their solutions) influence the solution to the current problem.

In order for the DynSDCOP solver to benefit from preserving knowledge from one problem to the

next, some of the problem must be preserved in the transition. Otherwise they are separate problems and better solved individually. The minimum information that must be preserved between states is; the semiring used, some of the agents and some of the constraints. In order for constraints to remain the same, the variables in their signature must also remain the same. The constraints in the problem does not refer solely to the constraints as specified in the problem, it is sufficient if some of the implied constraints are retained.

Based on the previous discussion, we choose to define a dynamic problem in a similar way to an extensive multi-player game [57] from game theory. There are three players in this game: The *solver* represents the agent(s) that are collaborating to solve the current problem. The solver's move is to propose a complete solution to the current problem. In many situations the different agents may not act in unison, when this happens we assume that the proposal from each agent is combined with the most recent proposals from the other agents to make a complete solution, which is presented as the overall solution. The *organization* represents the real world entities which 'own' the problem being solved. The organization's move is to adopt one of the solutions which have been proposed by the solver. As with the solver, the organization may be made up of many individual actors which do not always act in unison. When one actor acts separately to the others, it is modelled as the overall organization adopting a solution which only changes that actor's variables. The *world* represents all actors that are external to the organization. The world's move is to change the problem, such as adding and removing constraints or variables.

The organization has no control over the world's actions, so the world is treated as acting randomly. The solver and organization both act rationally, so they attempt to maximize the total utility for themselves. The utility for the solver is defined by the constraints in the problem it is solving, while the utility for the organization is based on the constraints in the problem and possibly additional information which is not encoded in the problem. We make no assumptions regarding the order in which the different players act. It is possible for the world to move, creating a new problem, for which the current solution is still the optimal solution, so neither the solver or the organization have to move. Similarly, it is possible for the solver to propose two different solutions to the same problem before one of them is adopted.

To model solution stability within the SDCOP framework, we create a new variable for each variable in the problem. These new variables are called the committed variable for variable x_i, or x_i^c. All agents who have write permissions on the original variable x_i must have read permissions on the committed variable x_i^c. No agents have write permission on the committed variable x_i^c. It is assumed that the organization updates the value of the variable when it adopts one of the solutions proposed by the solver. The cost of changing the value assigned to a variable can then be modelled as a constraint between a variable x_i and its committed version x_i^c. The idea of deadlines must be modelled as changes to the constraints as part of the dynamic problem. Due to this, the problem being solved by the solver changes whenever either of the other two players makes a move. When the organization makes a move the value assigned to the committed variables changes, while when the world makes a move, the constraints change.

Definition 37: *A Dynamic Semiring-Based Distributed Constraint Optimization Problem (DynS-DCOP) is a (possibly infinite) tree* $\langle N, E, P, f_p, f_w \rangle$. *Where* N *is a set of nodes,* E *is a set of edges* $\langle source, destination \rangle$, P *is a set of SDCOP's,* $f_p : N \longrightarrow P$ *is mapping function that associates a node with a state of the problem and* $f_w : E \longrightarrow (0..1]$ *is a mapping function that associates an edge with*

the probability that the destination node will be the next state of the DynSDCOP.

The following relationships must hold between a node n each of its children $N' = \{n'_1, \ldots .n'_j\}$*: Given* $f_p(n) = \langle \mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{V}, \mathcal{S}, \mathcal{C} \rangle$ *and* $f_p(n'_i) = \langle \mathcal{A}'_i, \mathcal{X}'_i, \mathcal{D}'_i, \mathcal{V}'_i, \mathcal{S}'_i, \mathcal{C}'_i \rangle$

- *All states in the problem must share the same semiring used to evaluate the quality of a solution.* $\forall n'_i \in N', \mathcal{V} = \mathcal{V}'_i$
- *Each state must have some overlap between its constraints and the constraints in its predecessor state.* $\forall n'_i \in N', \mathcal{C} \cap \mathcal{C}'_i \neq \emptyset$

The following relationship must hold between all outgoing edges E' *of a node n: The sum of the weights on all the outgoing edges of a node must be 1 (unless there are no outgoing edges). Let* E'_n *be the set of outgoing edges of a node n.* $\forall n \in N, E'_n = \emptyset \vee \sum_{e \in E'_n} f_w(e) = 1$

The information about the previous and future problems can now be included in the problem being solved. As discussed this comes in two forms, the actual cost to transition from the current adopted solution to the proposed solution, and the predicted cost to transition from the proposed solution to the solution of the (predicted) future problem. The actual cost is calculated by evaluating the stability constraints included in the problem. How the predicted cost is calculated and used is defined by the type of solution desired.

The solution to a DynSDCOP is a sequence of solutions to the SDCOPs along a path through the dynamic problem. As with the solutions for a SDCOP, there are several different intuitions based on how the future problems are considered. In addition, the solution concepts for a DynSDCOP combine with the solution concepts for SDCOP's. For example, you can have an optimistic optimal solution to a DynSDCOP.

Definition 38: *Given a DynSDCOP solution* $\langle s_0, \ldots, s_i \rangle$ *wrt. to the problem* $\langle N, E, P, f_p, f_w \rangle$ *and path* $\langle n_0, E_0, \ldots, n_{i-1}, e_{i-1}, ni \rangle$*, where a given problem* $p_j \in P = \langle \mathcal{A}_j, \mathcal{X}_j, \mathcal{D}_j, \mathcal{V}, \mathcal{S}_j, \mathcal{C}_j \rangle$*. The* transition cost *between SDCOP solution* s_a *and SDCOP solution* s_b *is given by the function* $f_t : \mathcal{S}_a \times \mathcal{S}_b \longrightarrow V$*.*

The transition cost is calculated by first assigning x_i^c *in* s_b *the value assigned to* x_i *in* s_a*. All constraints which have a committed variable in their signature are then evaluated and the results aggregated using* $\otimes$*.*

The concept of a solution to a SDCOP must be generalized for a solution to a problem within a DynSDCOP. The utility of the solution should also include the cost of changing from the previous solution to the current solution and the cost of changing from the current solution to the predicted future solution. The cost of changing from the previous solution to the current solution is represented by the constraints on the committed variables in the current problem. The cost of changing from the current solution to the predicted future solution is represented by the constraints on the committed variables in the predicted future problem. We only show how to generalize an acceptable solution for brevity, the modifications to the other intuitions are similar.

Definition 39: *An* **acceptable solution** *to an SDCOP* $\langle \mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{V}, \mathcal{S}, \mathcal{C} \rangle$ *where* $\mathcal{C} = \{c_1, \ldots, c_n\}$ *and successor states* $F = \{\langle P_0, s_0 \rangle, \ldots, \langle P_m, s_m \rangle\}$ *is a complete assignment s such that* $def_1(val(s \downarrow_{con_1})) \otimes \ldots \otimes def_n(val(s \downarrow_{con_n})) \otimes f_t(s, s_0) \otimes \ldots \otimes f_t(s, s_m) \preceq_V v$*, where* $v \in V$ *is a minimum threshold on the value of the solution.*

Similar to the definition of a solution to an SDCOP, there are many intuitions to calculate the cost of changing from the current state to the next state. We only formalize and present some of them,

there are other intuitions that are equally valid. We do not prescribe a particular intuition to use for a solution to an SDCOP in order to keep the framework as general as possible.

Definition 40: *A* naive solution *to the DynSDCOP $\langle N, E, P, f_p, f_w \rangle$, is a pair $\langle S, f_s \rangle$ where S is a set of solutions and $f_s : N \longrightarrow S$ is a mapping function which associates nodes in the DynSDCOP to solutions such that:*

- $\forall_{n \in N} f_s(n)$ *is a solution to* $f_p(n)$ *with successor states* $= \emptyset$.

Definition 41: *An* optimistic solution *to the DynSDCOP $\langle N, E, P, f_p, f_w \rangle$, is a pair $\langle S, f_s \rangle$ where S is a set of solutions and $f_s : N \longrightarrow S$ is a mapping function which associates nodes in the DynSDCOP to solutions such that:*

- *Let* E'_n *be the set of outgoing edges* $\langle source, destination \rangle$ *from node* n
- $\forall_{n \in N} f_s(n)$ *is a solution to* $f_p(n)$ *with successor states* $= \{f_p(destination) | \max_{f_w(e)} e \in E'_n\}$.

Definition 42: *A* minimax solution *to the DynSDCOP $\langle N, E, P, f_p, f_w \rangle$, is a pair $\langle S, f_s \rangle$ where S is a set of solutions and $f_s : N \longrightarrow S$ is a mapping function which associates nodes in the DynSDCOP to solutions such that:*

- *Let* E'_n *be the set of outgoing edges* $\langle source, destination \rangle$ *from node* n
- $\forall_{n \in N} f_s(n)$ *is a solution to* $f_p(n)$ *with successor states* $= \{f_p(destination) | \max_{f_t(f_s(n), f_s(destination)} e \in E'_n\}$.

Note that computing many of these solutions using existing algorithms requires that the solutions to the future states of the problem are already known, which is clearly impossible in real life. Instead a new class of algorithms will be required to efficiently solve problems of this form. One possible approach to solving these problems is to consider multiple states of the problem simultaneously, essentially solving the current problem and the next problem concurrently.

6.6 Conclusion

In this chapter we have identified three classes of DCOPs that can not be adequately represented using existing DCOP frameworks, which are maximisation, egalitarian and committee problems. These new classes of DCOPs are related to the extra challenges introduced when attempting to solve distributed problems. Some of these challenges include agent responsibility, privacy, co-operating/competing agents and no global objective. To address these challenges, we proposed a new framework, SDCOP, inspired by the c-semiring framework. This framework includes an explicit many-to-many mapping between agents and variables to support agent responsibility. It also makes use of an idempotent semiring to represent and reason with the quality of each solution.

We then extended SDCOP to support dynamic problems. Rather than assuming a linear sequence of future problems, we instead assume a branching tree of future problems. This allows uncertainty about the future to be modelled and taken into account when solving the problem. Further, by conceptually separating the solution the solver is currently exploring, the latest solution proposed by the solver and the solution adopted by the outside agency, it is possible to define constraints between

these different solutions. These constraints can be used to implement stability constraints, such as the increase in cost to change a committed variable.

Finally we defined a class of transformations which can be used to add additional properties to the SDCOP framework. These transformations modify the constraints and variables in the problem. Some problems that can be modelled via these transformations are automatic constraint relaxation and disjunctive temporal problems.

In the next chapter we describe how SBDO can be extended to support most of the problems which can be described in this framework.

Chapter 7

Support Based Distributed Optimization with Semirings

7.1 Introduction

In chapter 3 we presented a novel algorithm which is able to solve problems with most of the properties allowed by SDCOP (chapter 6, however it still has some limitations:

- The domain of variables ($\mathcal{D}$) must be finite.
- There must be a total order over solutions, i.e. $\forall a, b \in V, a \prec b \vee b \prec a$.
- The quality of a solution must be monotonically non-decreasing as the solution is extended, i.e. $\forall a, b \in V, a \otimes b \preceq a$.

In this chapter, we present an extension to the SBDO algorithm, Semiring Support Based Distributed Optimization (SSBDO). SSBDO removes one of the limitations of the SBDO algorithm, namely the requirement for a total order over solutions. A partial order over solutions occurs whenever there is more than one objective function being used to evaluate the quality of solutions. This can occur when there is only one objective function, when the objective function is 'fuzzy', leading to the possibility of incomparable solution qualities.

By being able to solve problems which do not have a total order, many real life problems can be solved by this instantiation. A common problem is production line configuration. In these problems the different objectives of minimize time, minimize cost and maximize quality have to be balanced. There are many other similar cases, another example is distributed co-ordination of search and rescue robots. The robots have to search an area while minimizing time, minimizing risk and minimizing battery usage. In addition there may or may not be a human supervisor, who guides the overall search strategy of the robots.

Another situation where there is not a total pre-order over the solutions is when a utility function returns qualitative values. In this case, the value assigned to two different solutions may not be comparable even when there is only one objective function.

Multi-objective problems are a common outcome of applying the meta-SDCOP transformations described in section 6.4. This further increases the utility of this algorithm, as it can solve many of the problems which are described by meta-SDCOPs. Some of these are the automatic relaxation of over-constrained problems and efficient solving of disjunctive temporal constraints.

SSBDO is designed as an instantiation of the SDCOP framework described in chapter 6. It supports many agents having write access for a variable. It also supports a wide range of different semirings for

measuring the utility of a solution. If the problem has hard constraints, then the semiring must include a bottom value, which represents an inconstant solution. Further, as with SBDO, the aggregation operator must always return a more preferred value. Except if one of the inputs is the bottom value, then the output must be bottom. The commonly accepted c-semiring framework [9] does not allow semirings with these properties.

This extension works by allowing agents to maintain many proposed solutions simultaneously. Rather than simply keeping the best partial solution so far and reasoning with it, all non-dominated solutions are maintained and reasoned with. At termination the partial solutions maintained by each agent can be aggregated into a set of complete solutions. These solutions approximate the Pareto-frontier of the problem.

The ability to maintain multiple proposed solutions also allows the algorithm to be proactive when solving dynamic problems as described in section 6.5. When proactively solving a dynamic problem, the algorithm must consider many problems simultaneously. These being the current version of the problem and the expected future versions of the problem. This algorithm can find and maintain a solution to each of the versions simultaneously. Constraints between solutions, such as the transition costs, can not be represented directly within this instantiation. They can be included as additional reasoning in the agent when it is implemented.

First, section 7.2 describes the changes required to the basic SBDO algorithm. Section 7.2.1 presents the modified version of the algorithm. Finally section 7.4 summarizes the results. The empirical results of this algorithm are presented in chapter 4.

Parts of this chapter have previously been published in conference proceedings [7].

7.2 Semi-Ring Support Based Distributed Optimization

SSBDO is a DynSDCOP solver based on SBDS [26] and SBDO [6]. Several changes are required to extend SBDO to use the DynSDCOP definition, these are discussed in section 7.2.1.

Solving a SDCOP via a distributed approach requires communication between agents. We use inspiration and techniques from formal argumentation, where the notion of an argument is used to encode viewpoints and attack to describe conflict between arguments. From this a notion of an 'proposal' message (definition 10) is conceived.

The same as in SBDO, most communication is in the form of proposal messages. These messages are inspired by formal argumentation, where the notion of an argument is used to encode viewpoints and attack to describe conflict between arguments. A proposal message contains the values an agent has selected for its variables and the context in which the decision was made.

Recall that we distinguish between two classes of constraints: local constraints and shared constraints. A local constraint is known by only one agent and can only be evaluated by that agent. For this to occur one agent must have write privileges to all of the variables in con_i (refer to definition 6.2). Shared constraints are known by all agents which have write privileges to a variable in the constraints signature and can be evaluated by any one of those agents. This occurs when more than one agent has write privileges to one of the variables in con_i (refer to definition 6.2).

Definition 43: *Given an SDCOP* $\langle \mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{V}, \mathcal{C} \rangle$*, an* **assignment** *is a triple* $\langle a, V, u \rangle$ *where* $a \in \mathcal{A}$*,* V *is a set of variable-value pairs and* u *is the utility of this assignment. The variable-value pairs indicate the values currently assigned to a subset of the variables which are controlled by* a*. The*

utility of this assignment is the sum of the value returned by all of a's local constraints, given the current assignments to a's variables.

The function f_u : assignment $\longrightarrow \mathbb{R}$ returns the utility of the assignment.

Definition 44: *Given an SDCOP $= \langle \mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{V}, \mathcal{C} \rangle$, a* **proposal** *is a pair $\langle VA, SCE \rangle$. Where VA (variable assignments) is a sequence $\langle ass_1, \ldots, ass_n \rangle$ of assignments such that the sequence of agents forms a simple path through the neighbourhood graph and there are no conflicting assignments. SCE (shared constraint evaluations) is a set of evaluations of shared constraints. A shared constraint can only be evaluated if an assignment to every variable involved in the constraint is included in VA. Each evaluation is a tuple $\langle c, u \rangle$ where o is a shared constraint and u is the utility returned by the constraint c given the assignments in VA.*

As an example consider the two agents A and B, who share a constraint O, as well as having their own local constraints. When A first creates a proposal it simply contains an assignment to A, $\langle \langle \langle A, \{\langle a, 1\rangle\}, 3\rangle\rangle, \{\}\rangle$. Later when B extends the proposal, B then has enough information to evaluate the shared objective, producing $\langle \langle \langle A, \{\langle a, 1\rangle\}, 3\rangle, \langle B, \{\langle b, 2\rangle\}, 2\rangle\rangle, \{\langle O, 10\rangle\}\rangle$.

As the idempotent semirings we are using to represent the utility of a partial solution generalize the concepts of 'constraint' and 'objective' used in SBDO, it is no longer necessary to differentiate between them. So the concept of 'objective' has been removed from SSBDO.

The sequence of variable assignments indicates the order in which this proposal has been constructed and is required both to store the utility of the solution and for the generation of nogoods. The set of shared constraint evaluations is required to record the utility of constraints shared by more than one agent. If the utility of the shared constraints is combined with the local constraints they might be double counted when cycles form. Note that in situations where privacy is important, the assignment to a variable only needs to be disclosed if another agent has read privileges for the variable.

The utility of an assignment can be determined by evaluating the applicable functions in $\mathcal{C}$ and aggregating them using $\otimes$. The total utility of an proposal is determined by applying the aggregation ($\otimes$) operator over the utility of each assignment and evaluation. As such a proposal encodes a partial solution to the problem as well as the relative utility of the partial solution. When a proposal is considered as an argument the first $n-1$ assignments form the justification and the last assignment is the conclusion.

The utility value provides a partial order over the the proposals (partial solutions). Comparison between proposals can be performed by first applying the $\otimes$ operator over the utility of each assignment and evaluation to determine its utility value and then the $\oplus$ operator to determine which proposal is better. Whenever we refer to one proposal being better than another in this paper it is with respect to this induced ordering.

The counterpart of a proposal is a nogood. A nogood represents a partial solution that violates at least one constraint and should never be reconsidered or included in the final solution. In order to benefit from SBDO's use of nogoods SSBDO requires that there is a value $\perp$ in $\mathcal{V}$ which represents an inconsistent solution. This way when the total utility of an proposal is $\perp$ SSBDO can generate a nogood. Nogoods with justifications [56] are used as these allow us to guarantee that all the hard constraints are satisfied (as shown in [24, 25]) as well as allowing obsolete nogoods to be identified after the constraints that the nogood violated are removed from the problem. For our purposes, a nogood with justification (originally defined in [56]) is treated as follows: **Definition 45:** *Given an SDCOP $\langle \mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{V}, \mathcal{C} \rangle$, a* **nogood** *is a pair $\langle s, C \rangle$ where $s \in S$ is a partial solution and $C \subseteq \mathcal{C}$ is the*

set of constraints that provides the **justification** *for the nogood, such that the combination of s and C is inconsistent (results in ⊥). As such a nogood represents a partial solution that is proven to not be part of any global solution.*

A **minimal nogood** *is a nogood $n = \langle s, C\rangle$ such that there does not exist a nogood $n' = \langle s', C\rangle$ where $s' \subset s$ or a nogood $n'' = \langle s, C'\rangle$ where $C' \subset C$.*

In static environments, detecting that the network has reached a quiescent state is sufficient to detect termination. This can be achieved by taking a consistent global snapshot [11]. The algorithm will also terminate if it detects that there is no solution to the problem, by generating the empty nogood. Otherwise, due to the dynamic nature of the input problem, the algorithm will only terminate when instructed to by an outside entity. Detecting that the network of agents has reached a quiescent state, or detecting that the problem is over-constrained are in themselves insufficient as terminating criteria. New inputs from the environment, in the form of added or deleted variables/constraints, might invalidate the current solution or allow a solution when there was not one before. The solver must continue running so that it can adapt the previous solution to the new problem.

7.2.1 Algorithm

```
begin
    Let N be a nogood derived from I
    Send N to A
    Delete I
end
```

Algorithm 6: send_nogood(I)

```
begin
    Let C be the constraint referenced
    for Each received nogood N do
        if N is in the remove-nogood message then
            Delete N from nogoods
            delete N from the remove-nogood message
        end
    end
    if counter ≠ 0 then
        Add the remove-nogood message to removed-constraints
    end
    pre-remove constraint(C)
end
```

Algorithm 7: process remove-nogood message

The core of SSBDO is very simple. First the agent reads any messages it has received from other agents and updates its knowledge. If it has received a proposed solution from another agent that is inconstant, it responds with a nogood message. Second it chooses an assignment for all variables for which it has write privileges. Third the agent sends a message to each of its neighbours, informing them of any change to its proposals. Finally the agent waits until it receives new messages.

All agents continue in this fashion until all their proposals are consistent. When this happens all agents will no longer send any new messages, as their proposed solutions do not change. If deployed

```
begin
    Let C be the removed constraint
    for Each neighbour A do
        for Each nogood N sent to A do
            Let obsolete = {}
            if N contains C as part of its justification then
                Add N to obsolete
                Delete N from sent-nogoods
            end
        end
        if |obsolete| > 0 then
            Let M be a new remove-constraint message with C and nogoods
            Send M to A
        end
    end
    for Each received nogood N do
        if N contains C as part of its justification then
            Mark N as obsolete
        end
    end
end
```

Algorithm 8: process remove-constraint message

```
begin
    Let A be a valid assignment to all local variables, chosen greedily
    Choose support and A such that all the following hold:

      • view is recv(support) extended by A

      • for All received proposals I do
    view ≺ I or I is consistent with A
        end

end
```

Algorithm 9: update_view()

in a static environment, termination can be detected by taking a consistent global snapshot [11]. Otherwise they continue to wait until they are informed that the environment has changed, or they are requested to terminate.

Because there is no ordering defined over the agents, this algorithm can very easily adapt to changes in the problem, such as adding or removing constraints. It also degrades gracefully when agents fail, making the overall system fault tolerant.

In order to generalize SBDO to support SDCOP, agents have been adapted to maintain more than one proposed solution at a time. This allows the algorithm to find many solutions in one execution. Changing the agents view from a single proposal to a set of proposals requires changes to the way the agents view is created as well as how proposals are sent to other agents. After these changes the information an agent γ stores is:

- support. The agent that γ is using as the basis for almost all decisions it makes. The support's beliefs about the world (its view) are considered to be facts.

```
begin
    while Not Terminated do
        for All received nogoods N do
            if this nogood is obsolete then
                decrement counter on the removed-constraint message
                if counter = zero then
                    delete constraint-removed message
                end
            else
                Add N to nogoods
                for All neighbours A do
                    if There is no valid assignment to myself wrt recv then
                        send_nogood(A)
                    end
                end
            end
        end
        for All received environment messages do
            Process message
        end
        for All received sets of proposals S_i do
            Let A be the agent who sent I
            Set recv to I
            for Each proposal I in S_i do
                if There is no valid assignment to myself wrt I then
                    send_nogood(A)
                end
            end
        end
        Set view to the non-dominated sub-set of all consistent extensions to all received
        proposals
        for All neighbours A do
            Set proposed_proposals to an empty set
            for Each proposal I in view do
                if I is part of a cycle then
                    if I is dominated by an proposal in recv then
                        Postpone this proposal
                    else
                        Add I to proposed_proposals
                    end
                else
                    Set preferred such that it meets the criteria
                    Set I' to a tail of I, such that the length of I is min(max_length, preferred)
                    Add I' to proposed_proposals
                end
            end
            if proposed_proposals ≠ sent(A) then
                Set sent(A) to proposed_proposals
                Send proposed_proposals to A
            end
        end
        Wait until at least one message has been received
    end
end
```

Algorithm 10: main()

- view. This is a set of proposals consisting of the proposals received from support with an assignment to γ's variables appended. This represents the γ's current beliefs about the world, or its world view.
- recv. This is a mapping from an agent α to the last set of proposals received from α. This stores the other agents most recent arguments.
- nogoods. This is an unbounded multi-set of all current nogoods received. It contains pairs $\langle \text{sender}, \text{nogood} \rangle$.
- sent(A). This is a mapping from an agent α to the last set of proposals sent to α. This stores the arguments most recently sent to the agents neighbours.
- sent-nogoods. This is an unbounded set of all nogoods sent by γ. It contains pairs $\langle \text{destination}, \text{nogood} \rangle$.
- removed-constraints. An unbounded set of known obsolete nogoods. This stores references to all the nogoods that are known to be obsolete, but have not yet been deleted.
- constraints. A set of all constraints γ knows. It must include all of an agent's local constraints and all constraints this agent shares with other agents.

As with SBDO each agent must first update its view based on its current support, as new information may have made its current view obsolete. The approach used in SBDO is to choose the neighbour that has sent the best proposal as this agents support, which clearly does not apply to SSBDO. Instead the agent that has sent the largest number of non-dominated proposals is chosen as this agents support. Specifically, all proposals this agent knows of, i.e. those it has received from its neighbours and those it has generated as its current view, are considered. Out of those proposals the set of non-dominated proposals is computed and they are partitioned based on their source. If the source of the largest partition is this agents view, then this agents support does not change. Otherwise this agent changes its support to the agent which is the source of the largest partition. If there is a tie for the largest partition, it is broken by considering the following criteria, in lexicographical order:

1. Largest number of proposals received from each source.
2. Largest total length of received proposals from each source.
3. Consistent random choice. i.e. if the set of proposals A is preferred over the set of proposals B, A will always be preferred over B[1].

To illustrate this procedure, consider an agent α which has received the following proposals from it's neighbour β:

- $\langle \langle \langle \epsilon, \{\langle e, 0\rangle\}, (3,0)\rangle, \langle \beta, \{\langle b, 1\rangle\}, (0,3)\rangle\rangle, \{\}\rangle$ with a total utility of (3, 3).
- $\langle \langle \langle \gamma, \{\langle c, 0\rangle\}, (2,1)\rangle, \langle \epsilon, \{\langle e, 1\rangle\}, (0,3)\rangle, \langle \beta, \{\langle b, 1\rangle\}, (0,3)\rangle\rangle, \{\}\rangle$ with a total utility of (2,7).

α has also received the following proposals from it's neighbour γ:

- $\langle \langle \langle \epsilon, \{\langle e, 0\rangle\}, (3,0)\rangle, \langle \gamma, \{\langle c, 0\rangle\}, (2,1)\rangle\rangle, \{\}\rangle$ with a total utility of (5,1).

[1] Hash functions can provide a suitable comparison.

Finally, α's view consists of the following proposals:

- $\langle\langle\gamma, \{\langle c, 1\rangle\}, (1, 2)\rangle, \langle\alpha, \{\langle a, 1\rangle\}, (0, 3)\rangle\rangle, \{\}\rangle$ with a total utility of (1,5).
- $\langle\langle\gamma, \{\langle c, 1\rangle\}, (1, 2)\rangle, \langle\alpha, \{\langle a, 0\rangle\}, (3, 0)\rangle\rangle, \{\}\rangle$ with a total utility of (4,2).

Given this information, the set of non-dominated proposals are:

- $\langle\langle\epsilon, \{\langle e, 0\rangle\}, (3, 0)\rangle, \langle\beta, \{\langle b, 1\rangle\}, (0, 3)\rangle\rangle, \{\}\rangle$ with a total utility of (3, 3).
- $\langle\langle\gamma, \{\langle c, 0\rangle\}, (2, 1)\rangle, \langle\epsilon, \{\langle e, 1\rangle\}, (0, 3)\rangle, \langle\beta, \{\langle b, 1\rangle\}, (0, 3)\rangle\rangle, \{\}\rangle$ with a total utility of (2,7).
- $\langle\langle\epsilon, \{\langle e, 0\rangle\}, (3, 0)\rangle, \langle\gamma, \{\langle c, 0\rangle\}, (2, 1)\rangle\rangle, \{\}\rangle$ with a total utility of (5,1).
- $\langle\langle\gamma, \{\langle c, 1\rangle\}, (1, 2)\rangle, \langle\alpha, \{\langle a, 0\rangle\}, (3, 0)\rangle\rangle, \{\}\rangle$ with a total utility of (4,2).

One of the non-dominated proposals originate from α, two originate from β and one from γ. This will cause α to change its support to β. If there was a tie between α and β, then α would win, as it has three total proposals compared to β's two. In the case of a tie between β and γ, β would win, as the total length of its proposals is five compared to γ's four.

In the case of SBDO, where there is at most one proposal from each source, there is normally only one proposal in the non-dominated set. It is possible for two (or more) proposals with equal utility to form the non-dominated set. When this happens the tie breaking procedure is equivalent in SSBDO and in SBDO.

If this agents support has changed, then it must re-compute its view. To generate its view, this agent extends the proposals it has received from its support. For each proposal it has received from its support, this agent computes the set of non-dominated proposals which can be generated by extending the received proposal with an assignment to this agent. If the agent changes the value of a variable which already has a value assigned to it in this proposal (i.e. more than one agent has write privileges), it must trim the proposal such that there are not two different values assigned to the same variable. The resulting reduction in the total utility of the proposal ensures that variables which have already been assigned by another agent will only be changed if there is significant benefit in doing so.

The procedure in SBDO for updating an agent's neighbours assumes that one proposal has been sent to and received from each neighbour. It must be generalized for sending sets of proposals, taking care to ensure the properties required for the proof of termination and completeness still hold. These are: postponing of proposals that are involved in a cycle, sending a new proposal if this agent is in conflict with the destination agent and not sending a new message if the content has not changed since the last message.

As with SBDO, each proposal is treated individually, then the proposals that will be sent to this neighbour are grouped and sent in one message. Cycle elimination is the same as in SBDO. If there are two consecutive assignments in the received proposal which were generated by this agent and the destination agent respectively, then this proposal is part of a cycle. If the proposal is dominated by one of the proposals previously sent to the destination agent then it must not be sent at this time. Instead the proposal which dominates it should be resent. If the proposal dominates all of the previously sent proposals, then the entire proposal should be sent. Otherwise, when neither proposal dominates the other, both should be sent. If the proposal is not part of a cycle the next consideration is how much of the proposal to send to the destination agent. The proposal must be long enough to meet the following criteria:

1. If one of the proposals previously sent to the destination agent is a sub-proposal of this proposal, then this proposal is an update. In which case the length of the newly sent proposal should be the length of the previously sent proposal +1.

2. It must contain enough assignments to evaluate shared objectives/constraints. Specifically, if there exists a constraint/objective involving at least the destination agent B and another agent C in this agents' view, then the proposal must contain the assignment to C.

3. If the assignment to A is not consistent with any proposal received from B, then A should send a counter-proposal that is more preferred than the conflicting proposal.

4. The proposal should be equal to or longer than the shortest proposal previously sent to B.

It is not always possible to send a proposal of the desired length, as the length of the sent proposal is limited by the length of the proposal in view. Once the correct length of the proposal has been decided a new proposal is created. Once all proposals in view have been considered, the new proposals are checked against the proposals previously sent to this agent. If any of them have changed a proposal message is sent to the destination agent containing all of the proposals.

To illustrate this procedure, again consider an agent α. α's view is currently:

- $$\langle\langle\langle\beta, \{\langle b, 0\rangle\}, (2,0)\rangle, \langle\epsilon, \{\langle e, 1\rangle\}, (0,3)\rangle, \langle\gamma, \{\langle c, 0\rangle\}, (2,1)\rangle, \langle\alpha, \{\langle a, 1\rangle\}, (0,3)\rangle\rangle, \{\}\rangle$$
  with a total utility of (4,7).

- $$\langle\langle\langle\epsilon, \{\langle e, 1\rangle\}, (0,3)\rangle, \langle\gamma, \{\langle c, 1\rangle\}, (1,2)\rangle, \langle\alpha, \{\langle a, 1\rangle\}, (0,3)\rangle\rangle, \{\}\rangle$$
  with a total utility of (1,8).

- $$\langle\langle\langle\gamma, \{\langle c, 0\rangle\}, (2,1)\rangle, \langle\alpha, \{\langle a, 0\rangle\}, (3,0)\rangle\rangle, \{\}\rangle$$
  with a total utility of (5, 1).

Further, α has previously sent the following proposals to β:

- $$\langle\langle\langle\beta, \{\langle b, 1\rangle\}, (0,3)\rangle, \langle\epsilon, \{\langle e, 0\rangle\}, (3,0)\rangle, \langle\gamma, \{\langle c, 1\rangle\}, (1,2)\rangle, \langle\alpha, \{\langle a, 0\rangle\}, (3,0)\rangle\rangle, \{\}\rangle$$
  with a total utility of (7,5).

- $$\langle\langle\langle\gamma, \{\langle c, 0\rangle\}, (2,1)\rangle, \langle\alpha, \{\langle a, 0\rangle\}, (3,0)\rangle\rangle, \{\}\rangle$$
  with a total utility of (5,1).

Finally, α has received (along with other proposals) the following proposal from β:

$$\langle\langle\langle\epsilon, \{\langle e, 0\rangle\}, (3,0)\rangle, \langle\gamma, \{\langle c, 1\rangle\}, (1,2)\rangle, \langle\alpha, \{\langle a, 0\rangle\}, (3,0)\rangle, \langle\beta, \{\langle b, 1\rangle\}, (0,3)\rangle, \rangle, \{\}\rangle$$

with a total utility of (7, 5).

We can see that the proposal received from β contains an assignment to α then an assignment to β, as such they are part of a cycle. The first proposal in α's view has assignments to the same variables in the same order as the proposal received from β, so it might need to be postponed. Neither proposal dominates the other, so both proposals are sent to β. The second proposal in α's view is an update to the second proposal α sent previously, so a slightly longer subset of that proposal is sent to β. Finally, the third proposal is new, so α attempts to send a length three version of it, but as it is only length two, the entire proposal is sent. Therefore the new proposal message α sends to β is:

- $$\langle\langle\langle\beta, \{\langle b, 1\rangle\}, (0, 3)\rangle, \langle\epsilon, \{\langle e, 0\rangle\}, (3, 0)\rangle, \langle\gamma, \{\langle c, 1\rangle\}, (1, 2)\rangle, \langle\alpha, \{\langle a, 0\rangle\}, (3, 0)\rangle\rangle, \{\}\rangle$$
  with a total utility of (7,5).

- $$\langle\langle\langle\beta, \{\langle b, 0\rangle\}, (2, 0)\rangle, \langle\epsilon, \{\langle e, 1\rangle\}, (0, 3)\rangle, \langle\gamma, \{\langle c, 0\rangle\}, (2, 1)\rangle, \langle\alpha, \{\langle a, 1\rangle\}, (0, 3)\rangle\rangle, \{\}\rangle$$
  with a total utility of (4,7).

- $$\langle\langle\langle\epsilon, \{\langle e, 1\rangle\}, (0, 3)\rangle, \langle\gamma, \{\langle c, 1\rangle\}, (1, 2)\rangle, \langle\alpha, \{\langle a, 1\rangle\}, (0, 3)\rangle\rangle, \{\}\rangle$$
  with a total utility of (1,8).

- $$\langle\langle\langle\gamma, \{\langle c, 0\rangle\}, (2, 1)\rangle, \langle\alpha, \{\langle a, 0\rangle\}, (3, 0)\rangle\rangle, \{\}\rangle$$
  with a total utility of (5, 1).

The procedure for determining the length of each proposal to send is the same as in SBDO. However there are some changes due to it acting on two sets of proposals, rather than two proposals. First, for each new proposal, it is not clear which old proposal it should be compared with. In this case it is compared with all of them and the longest required proposal is sent. Further, when a proposal is postponed, the old version of the proposal must be sent. Otherwise the proposal will be lost when the destination agent updates its received proposals. Finally, an updated proposal message must be sent when any of the individual proposals change.

The changes made to the procedure for sending updates to an agents neighbours invalidate the proof of termination for SBDO. Specifically the change from sending a single proposal to a set of proposals makes lemma 2 not applicable. Here we present a generalization of that lemma for sets of proposals. This is based on the proof of termination for SBDS [23].

Lemma 6: *If no new nogoods are generated, then eventually the utility of view will become stable for each agent.*

Proof: *Let $W_i \subseteq \mathcal{X}$ be the set of agents whose view dominates $i \in \mathcal{Z}$ of the possible solutions to the problem. An agent's view v dominates a solution s iff there exists a proposal $p \in v$ such that the utility of p is greater than or equal to the utility of s (not less than or incomparable). We will prove that any decrease in $|W_i|$ must be preceded by an increase in $|W_j|$, where $j < i$.*

First, we note that an agent will never willingly reduce the number of solutions its view dominates, as per the proposal ordering and the requirements of `update_view()`*. So, in the usual case, $|W_i|$ will be monotonically increasing, for all i. However, in limited circumstances an agent may receive a*

weaker proposal from its support, and so the number of solutions its view dominates could be forced to decrease. Such events are rare, but they can occur whenever a cycle of supporting agents is formed. Let us assume that some agent v receives a worse proposal I from its current support, and so v is forced to choose a view which dominates less solutions. Let i number of solutions v's old view dominates, and j be the number of solutions v's new view dominates, respectively. The new, worse view for v will obviously decrease each $|W_k|$, where $j < k \leq i$.

However, for v to have received the worse proposal I, some agent w must have formed a cycle by changing its support. Note that w will only have selected a new support if it could increase the number of solutions its view dominates, as per the proposal ordering and the requirements of `update_view()`*. Also note that the newly-formed cycle cannot have a total utility of more than j, else there would have been no reason to reduce the number of solutions v's view dominates. Therefore, the number of solutions w's new view dominates must then be less than or equal to j, but is certainly more than it's old view.*

So, if an agent v is forced to reduce the number of solutions its view dominates, then there must be some preceding agent w which increased the number of solutions its view dominates. Further, w's new view is guaranteed to dominate no more solutions than v's new view. Therefore, the term $|W_1|.|W_2|.|W_3|\ldots.$ must increase lexicographically over time. As the term is bounded above, we can conclude that the utility of view must eventually become stable for each agent. □

Theorem 14: *SSBDO is an instance of SDCOP with the following limitations:*

- *The domain of variables ($\bigcup \mathcal{D}$) must be finite.*
- *The quality of a solution must be monotonically non-decreasing as the solution is extended, i.e. $\forall a, b \in V, a \otimes b \preceq a$.*

Proof: *The elements $\mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{V}$ and $\mathcal{C}$ are defined the same in SSBDO and SDCOP.*

SSBDO does not support problems where the domain of a variable may be finite. This case is excluded by the first restriction on an SDCOP.

Further, the SSBDO algorithm is only sound when aggregating two semiring values does not produce a less preferred semiring value. This case is excluded by the second restriction on an SDCOP. □

7.2.2 Example

We present the following graph colouring problem as an example of how SSBDO functions. There are three agents, Γ, Δ and Θ, each of which have write privileges for one variable, γ, δ and θ respectively. Each agent has read privileges for all the variables. Each variable can take one of three 'colours', 0, 1 and 2. Neighbouring variables share a constraint of colour difference, maximize the difference between the values assigned to each agent. Each variable also has a unary constraint of colour affinity, minimize the distance between its value and an ideal value. The ideal value is 0, 1 and 1 for γ, δ and θ respectively. These constraints implement two separate objectives, as described above. When showing utility, the first value is for the colour difference objective and the second value is for the colour affinity objective.

When the algorithm starts each agent has not received any other proposals to build upon. So all of them choose an assignment based on the colour affinity constraint. Γ adopts the proposal

$\langle\langle\langle\Gamma, \{\langle\gamma, 0\rangle\}, (0, 2)\rangle\rangle, \{\}\rangle$, Δ adopts the proposal $\langle\langle\langle\Delta, \{\langle\delta, 1\rangle\}, (0, 2)\rangle\rangle, \{\}\rangle$ and Θ adopts the proposal $\langle\langle\langle\Theta, \{\langle\theta, 1\rangle\}, (0, 2)\rangle\rangle, \{\}\rangle$. Each agent then sends their choice to each of the other agents.

Now we concentrate only on Θ, the reasoning for the other agents is similar. None of the proposals Θ has dominate any of the others, and each of itself, Δ and Γ have supplied one non-dominated proposal, further all proposals are of length 1. As such Θ randomly chooses Γ as its support and extends each of Γ's proposals to find the following non-dominated proposals:

- $\langle\langle\langle\Gamma, \{\langle\gamma, 0\rangle\}, (0, 2)\rangle, \langle\Theta, \{\langle\theta, 2\rangle\}, (0, 0)\rangle\rangle, \{\langle(\gamma, \theta), (2, 0)\rangle\}\rangle$ with a utility of $(2, 2)$
- $\langle\langle\langle\Gamma, \{\langle\gamma, 0\rangle\}, (0, 2)\rangle, \langle\Theta, \{\langle\theta, 1\rangle\}, (0, 1)\rangle\rangle, \{\langle(\gamma, \theta), (1, 0)\rangle\}\rangle$ with a utility of $(1, 3)$

As the first proposal found is new, only the front of it, $\langle\langle\langle\Theta, \{\langle\theta, 2\rangle\}, (0, 0)\rangle\rangle, \{\}\rangle$, is sent to the other agents, while the entirety of the second proposal is sent.

In Θ's next cycle it receives the following proposals from Γ and Δ:

- $\langle\langle\langle\Delta, \{\langle\delta, 1\rangle\}, (0, 1)\rangle, \langle\Gamma, \{\langle\gamma, 0\rangle\}, (0, 2)\rangle\rangle, \{\langle(\gamma, \delta), (1, 0)\rangle\}\rangle$ with a utility of $(1, 3)$
- $\langle\langle\langle\Theta, \{\langle\theta, 1\rangle\}, (0, 2)\rangle, \langle\Delta, \{\langle\delta, 1\rangle\}, (0, 2)\rangle\rangle, \{\langle(\delta, \theta), (0, 0)\rangle\}\rangle$ with a utility of $(0, 4)$
- $\langle\langle\langle\Delta, \{\langle\delta, 0\rangle\}, (0, 0)\rangle\rangle, \{\}\rangle$ with a utility of $(0, 0)$, (because it is a new assignment it starts at length one)
- $\langle\langle\langle\Delta, \{\langle\delta, 2\rangle\}, (0, 0)\rangle\rangle, \{\}\rangle$ with a utility of $(0, 0)$

Θ then decides to retain Γ as its support because the largest number of non-dominated proposals are from Θ's view (Δ and Θ supply one each). Next Θ extends the proposal received from Γ to get:

- $\langle\langle\langle\Delta, \{\langle\delta, 1\rangle\}, (0, 1)\rangle, \langle\Gamma, \{\langle\gamma, 0\rangle\}, (0, 2)\rangle, \langle\Theta, \{\langle\theta, 1\rangle\}, (0, 1)\rangle\rangle,$
  $\{\langle(\gamma, \delta), (1, 0)\rangle, \langle(\gamma, \theta), (1, 0)\rangle, \langle(\delta, \theta), (0, 0)\rangle\}\rangle$ with a utility of $(2, 4)$
- $\langle\langle\langle\Delta, \{\langle\delta, 1\rangle\}, (0, 1)\rangle, \langle\Gamma, \{\langle\gamma, 0\rangle\}, (0, 2)\rangle, \langle\Theta, \{\langle\theta, 2\rangle\}, (0, 0)\rangle\rangle,$
  $\{\langle(\gamma, \delta), (1, 0)\rangle, \langle(\gamma, \theta), (2, 0)\rangle, \langle(\delta, \theta), (1, 0)\rangle\}\rangle$ with a utility of $(4, 3)$

Again Θ then informs Δ and Γ of the solutions it has chosen.

These two proposals represent the optimal solutions for this problem, so execution continues for one more cycle, as Δ and Γ accept them as the optimal solutions.

7.3 Results

To assess the performance of SSBDO, we setup an empirical evaluation. To do so we implemented SSBDO in C++[2] and ran tests on a set of graph colouring problems. The tests were run on an Intel Xeon X3450 CPU with 8GB of RAM.

At this time there is only one other published algorithm that is capable of solving problems with many objective functions, B-MUMS [17]. We do not compare SSBDO with B-MUMS as B-MUMS does not support hard constraints, as are used in our experiments, and only returns one solution[3].

In our test problem there is a one to one mapping from agents to variables and each variable can take a value from the domain $\{0, 1, 2, 3, 4\}$. Each variable is identified by a unique integer. There

[2]Source code available from http://www.geeksinthegong.net/svn/sbdo/trunk/ .

[3]We acknowledge that B-MUMS could easily be modified to return many solutions, as it finds them during processing.

are three constraints between each pair of neighbouring variables. The first is a hard constraint that neighbouring variables must not have the same value. The second is a valued constraint to maximize the distance between the two values, given that the values wrap around i.e. the distance between 0 and 4 is 1. The third is a valued constraint where the variable with the higher identifier should be assigned a larger value. Finally, every variable has a unary valued constraint to minimize the distance between the variables value and an ideal value, being the variables identifier modulo 5.

For our tests we varied the number of variables in each problem and the number of constraints. The parameter for the graph connectedness varies linearly between 0, where the constraints form a spanning tree over the variables, and 1, which is a fully connected graph. We used a number of variables from the set {4, 5, 6, 7, 8, 9, 10, 15, 20, 25}, a number of constraints from the set {0.0, 0.1, 0.2, 0.3. 0.4}, and randomly generated five problems for each pair of parameters. Each problem was solved five times and the performance averaged to give the results presented here. Individual runs were terminated after half an hour of wall clock time.

We present the performance of SSBDO based on five metrics and the standard deviation for each metric:

1. (Terminate) Time for the algorithm to terminate.
2. (Aggregate) Time to aggregate the partial solutions.
3. (number of solutions) Total number of solutions found.
4. (solution quality) The average of the minimum euclidean distance from the utility of each non-optimal solution to the utility of an optimal solution. Formally:

$$\text{quality} = \frac{\sum_{n \in N} \min\left(f\left(n, o_1\right), \ldots, f\left(n, o_x\right)\right)}{|N|}$$

   where $f(x, y)$ is the euclidean distance between x and y, S is the set of the utilities of the solutions found by SSBDO, O is the set of utilities of the optimal solutions and $N = S/O$.

5. (Proportion) The proportion of optimal solutions found. Formally:

$$\text{proportion} = \frac{|O \cap S|}{|O|}$$

Note that the set of optimal solutions must be known to compute metrics four and five. We used exhaustive search to find all the optimal solutions for problems with up to ten variables. it proved to be infeasable to solve bigger problems using exhaustive search.

Each agent only has a local view of the problem, so a post-processing step is required to combine all the partial solutions into complete solutions. Whether this step is required depends on the problem being solved. Decision support applications will require complete solutions, but some things like autonomous robots may be able to function using only local knowledge. It also depends on how many solutions are desired, for these experiments we extracted all pareto-optimal solutions found by SSBDO. Due to these factors, we have presented the time required for the algorithm to terminate and the time to aggregate the partial solutions into global solutions separately.

The performance of SSBDO is shown in Table 7.1 . The number of constraints in the problem had very little effect on the performance, so we have not reported those results here. As expected, the

variables	Terminate (s)	Aggregate (s)	num. of solutions	sol. quality	Proportion
4	0.15 (0.13)	0.01 (0.00)	9.66 (5.20)	0.13 (0.13)	0.46 (0.26)
5	0.20 (0.12)	0.01 (0.01)	11.38 (6.33)	0.22 (0.32)	0.49 (0.26)
6	0.35 (0.33)	0.02 (0.01)	13.58 (6.11)	0.25 (0.29)	0.52 (0.24)
7	0.76 (0.83)	0.04 (0.04)	17.54 (10.18)	0.23 (0.27)	0.47 (0.24)
8	1.87 (3.24)	0.05 (0.04)	21.87 (14.65)	0.35 (0.34)	0.38 (0.24)
9	3.68 (5.54)	0.15 (0.24)	31.63 (24.70)	0.38 (0.33)	0.35 (0.24)
10	6.02 (8.33)	0.16 (0.20)	30.27 (17.72)	0.50 (0.33)	0.25 (0.21)
15	125.10 (233.45)	5.43 (20.17)	52.47 (66.61)	-	-
20	287.72 (259.97)	64.67 (184.73)	76.44 (189.77)	-	-
25	433.88 (548.73)	269.87 (546.00)	50.00 (158.81)	-	-

Table 7.1: Performance of SSBDO. See text for description of metrics.

number of variables in the problem has a large impact on the performance. Both the time required for the algorithm to terminate and the number of solutions found when the algorithm does terminate increases exponentially with the number of variables. Further the time required to aggregate all solutions is dependent on the number of solutions, so it also rises exponentially.

The standard deviations show that the performance of SSBDO is highly unstable, often the standard deviation is greater than the mean. This is due to the highly non-deterministic nature of the SSBDO algorithm, as the order agents are scheduled on the CPU can strongly influence the search direction, which determines which solutions are found and how much effort is required to terminate.

The proportion of the optimal solutions that SSBDO finds drops off as the number of variables increases. While the average distance from each non-optimal, found solution to an optimal solution remains constant, representing a change to the assignment to one variable by about one unit. This shows that while the number of points discovered on the actual Pareto-front drops off as the number of variables increases, the solutions found remain close to the actual Pareto-front.

In the process of solving these problems, SSBDO generates a large number of nogoods. As our implementation of SSBDO only has relatively simple code for searching and checking nogoods, this represents a significant performance bottleneck and contributes to the time required for the larger problems.

7.4 Conclusion

We have modified SBDO to support problems where there does not exist a total pre-order over the set of solutions. This is usually due to there being several competing objectives, but can occur in other situations.

In order to solve problems of this form, each agent maintains multiple candidate solutions simultaneously. Otherwise processing proceeds similarly to SBDO. The partial solutions maintained by each agent can then be combined into a set of complete solutions.

By allowing each agent to maintain multiple separate solutions, DynSDCOP problems can be solved. This requires identifying which state of the problem the solution corresponds to in the utility of the solution. While not described in this instantation of SDCOP or supported by this implementation, constraints can be defined between solutions. This allows stability constraints to be implemented.

Chapter 8

Conclusion

Current work on Distributed Constraint Optimisation problems has ignored many of the properties that make a distributed problem unique. When a problem is distributed between many agents, the solving algorithm must account for the resulting challenges. Agents may be self-interested, unreliable or even malicious. Communication between agents is expensive, and may not be reliable. Finally, as agents may be self-interested and distributed, problems may be composed 'bottom-up', there may not be a universally agreed objective.

The main contribution of this thesis is a new algorithm for solving Distributed Constraint Optimization Problems, called Support Based Distributed Optimization (SBDO). This algorithm has been designed from the beginning to be a distributed algorithm, with a focus on maintaining the autonomy and equality of agents, as well as being fault tolerant. By basing the communication between agents on argumentation principles, we avoid the imposed hierarchy of agents that most other DCOP algorithms require. We have also attempted to minimize any imposed order within the communication protocol, so any agent can send a message to its neighbours at any time. These properties make SBDO suitable for solving dynamic problems. Finally, with of the unrestricted communication and leveraging the redundant information held by each agent, SBDO can continue solving when agents in the problem fail. Also, when an agent restarts, it can quickly recover its previous state from the information held by other agents.

We have also presented several modifications that should improve the performance of SBDO. By using region nogoods, less nogoods will be generated and sent to other agents. Though significantly more computation is required to generate each nogood and to test if a partial solution is eliminated by a nogood. Similarly, each proposal not sent due to forward checking saves two messages (the proposal and corresponding nogood). Only a small number of constraint checks is required for each agent to test its proposals against the additional constraints. The total impact of these changes will have to be empirically evaluated.

Empirical results show that SBDO is very competitive with existing DCOP algorithms. We have shown that SBDO outperforms DynCOAA on dynamic problems, achieving a better utility with the same amount of processing time. We have also shown that SBDO's performance only degrades slightly when agents in the problem fail. The final solution is only slightly worse, though more NCCCs are required for the algorithm to terminate. The number of NCCCs and messages required scales well with the size of the problem. Finally, the solution found by SBDO often has a utility greater than 95% of the utility of the optimal solution.

We briefly describe some possible applications of the SBDO algorithm in chapter 5. The first is scheduling patient treatment for radiotherapy. Second is combining SBDO with auctions to optimally

Algorithm	Time Complexity	Space Complexity	Properties
ADOPT	Exponential	Linear	Bounded error, Complete, Sound
DGibbs	Exponential	Linear	Anytime, Complete, Sound
DPOP	Polynomial	Exponential	Complete, Dynamic, Sound
DynCOAA	Exponential	Linear	Anytime, Dynamic, Global communication, Sound
Max-Sum	Exponential	Linear	Anytime, Autonomy, Bounded error, Dynamic, Equality, Fault tolerant, Multi-objective
MGM	Exponential	Linear	Anytime, Dynamic, Equality
SBDS	Exponential	Exponential	Anytime, Autonomy, Equality, Sound
SBDO	Exponential	Exponential	Anytime, Autonomy, Dynamic, Equality, Fault tolerant, Multi-objective

Table 8.1: Summary of existing algorithms

route traffic in a road network. Third is combining SBDO with agent based modelling to improve the quality of the models.

The common formalisation for representing DCOPs is not sufficent to represent all of the real world problems. In chapter 6 we propose a unifying framework, Semiring Distributed Constraint Optimization Problem (SDCOP), which is designed to be able to represent all of the different classes of distributed problems. We focus on three classes of problems which the current accepted approach, c-semirings, can not represent: Egalitarian/Equitable problems, Maximisation problems and Committee problems.

We then propose two extensions to SDCOP which allow it to represent the remaining CSP variations. First, Dynamic SDCOP (DynSDCOP) allows SDCOP to represent problems which change over time. The possible states of the problem is represented as a tree of SDCOPs. This allows proactive algorithms to reason with the possible future states of the problem when solving. Further, we explicitly distinguish between the value of a variable which has been adopted (or committed) and the value which is proposed. Using this, any additional cost of changing a committed variable can be modelled as a constraint between the adopted and proposed versions of the variable. Second, Meta SDCOP is a class of transformations which can be applied to an SDCOP. These transformations are used to add or make explicit additional properties in an SDCOP.

In chapter 7 we have presented an extension of the SBDO algorithm to support multi-objective problems, called Semiring Support Based Distributed Optimization (SSBDO). This is achieved by allowing an agent to maintain multiple candidate solutions simultaneously. Whenever an agent has two (or more) possible solutions which have equal or incomparable utility it maintains all of them. When an agent sends a proposal to another agent, it sends all of its candidate solutions. A limitation of this approach is that it is difficult to combine each agents set of partial solutions into a set of complete solutions. Currently this is done a centralized post-processing step. In some settings the agents may not need to know the complete solutions in order to act, which will avoid this limitation.

SSBDO does a good job of solving multi-objective problems. The empirical results show that it is capable of finding between 25% and 50% of the actual Pareto-frontier of the problem. With larger problems it finds less of the actual Pareto-frontier. Further, the solutions which are not on the actual Pareto-frontier are only a small distance from one of the solutions on the actual Pareto-frontier. This is comparable to the base SBDO algorithm, which often finds solutions which are within 5% of the optimal solution.

In Table 8.1 the properties of the SBDO algorithm described in this thesis are compared with the other algorithms identified in the literature review. In the worst case, the complexity of the algorithm remains exponential. Based on the empirical results shown in section 4.1, the average performance of SBDO is competitive with the other algorithms. SBDO is still sound with respect to hard constraints, but is not sound with respect to objectives, so it loses the sound property in the optimization domain. We have added the dynamic, fault tolerant and multi-objective properties to the SBDS algorithm. Of those properties, dynamic is supported by many other algorithms. Max-Sum is the only other algorithm which has the fault tolerant and multi-objective properties. SBDO is fault tolerant with respect to agents failing and messages arriving in random order, while Max-Sum is fault tolerant with respect to messages being lost.

The SBDO algorithm achieves our goal of an algorithm for solving DCOP problems with a focus on the properties of equality, autonomy and fault tolerance. From the empirical results we have shown that these properties do not compromise the performance of the algorithm. Further, with the semiring and continuous extensions to SBDO, we have shown that SBDO is a highly flexible algorithm for solving DCOP problems.

With the advancements in this thesis, we believe that DCOP technology is much closer to being ready for use in the real world. The autonomy and equality properties allow agents to easily join and leave the overarching problem. They also ensure that the agents can trust the fairness of the protocol. For full fairness, sabotage resistance is required, so that agents can trust the other agents. The algorithm is highly tolerant of agents failing, which can often happen in real world settings. It is less tolerant of message loss or message corruption, but those properties can be gained by choosing an appropriate message transfer protocol, such as TCP/IP. The result is DCOP technology is now applicable to many new problem domains.

8.0.1 Future Work

There are several remaining ways the SBDO algorithm can be extended. The first is to finish the support for problems with bounded infinite domains. Most of the work required for this extension is described in section 3.5. Secondly, SBDO can be extended to support more of the problems which can be described using SDCOP. Specifically by adding support for allowing more than one agent to have write access to a single variable. Third, currently SBDO has to store a lot of information to ensure it operates correctly. It may be possible to discard some of that information to reduce the memory requirements of each agent. This is more difficult due to supporting dynamic problems. Finally the results show that SBDO generally finds good quality solutions. There may be a lower bound on the quality of solutions.

Bibliography

[1] S.M. Aji and R.J. McEliece. The generalized distributive law. *Information Theory, IEEE Transactions on*, 46(2):325–343, mar 2000.

[2] Syed Muhammad Ali, Sven Koenig, and Milind Tambe. Preprocessing techniques for accelerating the DCOP algorithm ADOPT. In Frank Dignum, Virginia Dignum, Sven Koenig, Sarit Kraus, Munindar P. Singh, and Michael Wooldridge, editors, *Autonomous Agents and Multiagent Systems*, pages 1041–1048. ACM, 2005.

[3] J. Allen. Maintaining Knowledge About Temporal Intervals. In *Communications of the Association for Computing Machinery*, volume 26, pages 832–843, Nov 1983.

[4] Valerie Barr and Zdravko Markov, editors. *Proceedings of the Seventeenth International Florida Artificial Intelligence Research Society Conference, Miami Beach, Florida, USA*. AAAI Press, 2004.

[5] U. Bertele and F. Brioschi. *Nonserial Dynamic Programming*. Academic Press, 1972.

[6] Graham Billiau, Chee Fon Chang, and Aditya Ghose. SBDO: A New Robust Approach to Dynamic Distributed Constraint Optimisation. In *Principles and Practice of Multi-Agent Systems*, 2010.

[7] Graham Billiau, Chee Fon Chang, and Aditya Ghose. Multi-objective distributed constraint optimization using semi-rings. In Hoa Khanh Dam, Jeremy V. Pitt, Yang Xu, Guido Governatori, and Takayuki Ito, editors, *Principles and Practice of Multi-Agent Systems*, volume 8861 of *Lecture Notes in Computer Science*, pages 407–422. Springer, 2014.

[8] Graham Billiau and Aditya Ghose. SBDO: A New Robust Approach to Dynamic Distributed Constraint Optimisation. In Jung-Jin Yang, Makoto Yokoo, Takayuki Ito, Zhi Jin, and Paul Scerri, editors, *Principles of Practice in Multi-Agent Systems*, volume 5925 of *Lecture Notes in Computer Science*, pages 641–648. Springer, 2009.

[9] Stefano Bistarelli, Ugo Montanari, and Francesca Rossi. Semiring-based constraint satisfaction and optimization. *Journal of Association for Computing Machinery*, 44(2):201–236, 1997.

[10] Debra C. Cascardo. Smart scheduling: The key to practice efficiency. *Medscape Today*, 2000.

[11] K. Mani Chandy and Leslie Lamport. Distributed Snapshots: Determining global states of distributed systems. *Association for Computing Machinery Transactions on Computer Systems*, 3(1):63–75, Feb 1985.

[12] B. A. Davey and H. A. Priestly. *Introductions to Lattices and Order*. Cambridge University Press, 1990.

[13] R. Dechter. Bucket elimination: A unifying framework for reasoning. In *Artificial Intelligence*, volume 113, pages 41–85, 1999.

[14] R. Dechter and I. Rish. Mini-buckets: A general scheme for bounded inference. In *Journal of the Association for Computing Machinery*, volume 50, pages 107–153, Mar 2003.

[15] M. Dorigo, M. Birattari, and T. Stutzle. Ant colony optimization. *Computational Intelligence Magazine, IEEE*, 1(4):28–39, Nov 2006.

[16] Alessandro Farinelli, Alex Rogers, and Nick Jennings. Bounded Approximate Decentralised Coordination using the Max-Sum Algorithm. In *IJCAI-09 Workshop on Distributed Constraint Reasoning (DCR)*, pages 46–59, July 2009. Event Dates: 13th July 2009.

[17] Francesco M. Delle Fave, Ruben Stranders, Alex Rogers, and Nicholas R. Jennings. Bounded Decentralised Coordination over Multiple Objectives. In Liz Sonenberg, Peter Stone, Kagan Tumer, and Pinar Yolum, editors, *Autonomous Agents and Multiagent Systems*, pages 371–378. IFAAMAS, 2011.

[18] Francesco Maria Delle Fave, Alessandro Farinelli, Alex Rogers, and Nick R. Jennings. A Methodology for Deploying the Max-Sum Algorithm and a Case Study on Unmanned Aerial Vehicles. In Markus P. J. Fromherz and Hector Muñoz-Avila, editors, *Innovative Applications of Artificial Intelligence*. AAAI, 2012.

[19] QE Foundation. online. http://www.qefoundation.org, accessed 2010-08-20.

[20] Stuart Geman and D. Geman. Stochastic relaxation, gibbs distributions, and the bayesian restoration of images. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, PAMI-6(6):721–741, Nov 1984.

[21] Tal Grinshpoun, Alon Grubshtein, Roie Zivan, Arnon Netzer, and Amnon Meisels. Asymmetric Distributed Constraint Optimization Problems. *Journal of Artificial Intelligence Reasoning*, 47:613–647, 2013.

[22] Patricia Gutierrez, Pedro Meseguer, and William Yeoh. Generalizing ADOPT and BnB-ADOPT. In Toby Walsh, editor, *International Joint Conference on Artificial Intelligence*, pages 554–559. IJCAI/AAAI, 2011.

[23] Peter Harvey. *Solving Very Large Distributed Constraint Satisfaction Problems*. PhD thesis, University of Wollongong, 2010.

[24] Peter Harvey, Chee Fon Chang, and Aditya Ghose. Support-based distributed search: a new approach for multiagent constraint processing. In Nakashima et al. [42], pages 377–383.

[25] Peter Harvey, Chee Fon Chang, and Aditya Ghose. Support-Based Distributed Search: A New Approach for Multiagent Constraint Processing. In Nicolas Maudet, Simon Parsons, and Iyad Rahwan, editors, *Argumentation and Multi-Agent Systems*, volume 4766 of *Lecture Notes in Computer Science*, pages 91–106. Springer, 2006.

[26] Peter Harvey and Aditya Ghose. Relaxation of Soft Constraints Via a Unified Semiring. In Luc Lamontagne and Mario Marchand, editors, *Canadian Conference on Artificial Intelligence*, volume 4013 of *Lecture Notes in Computer Science*, pages 122–133. Springer, 2006.

[27] K.B. Lakshmanan, N. Meenakshi, and K. Thulasiraman. A time-optimal message-efficient distributed algorithm for depth-first-search. *Information Processing Letters*, 25(2):103–109, 1987.

[28] J. Larrosa. On arc and node consistency in weighted CSP. In *AAAI Conference on Artificial Intelligence*, pages 48–53, 2002.

[29] Thomas Léauté, Brammert Ottens, and Radoslaw Szymanek. FRODO 2.0: An Open-Source Framework for Distributed Constraint Optimization. In *Proceedings of the IJCAI'09 Distributed Constraint Reasoning Workshop (DCR'09)*, pages 160–164, Pasadena, California, USA, July 13 2009. http://frodo2.sourceforge.net.

[30] Alan K. Mackworth. Consistency in Networks of Relations. *Artificial Intelligence*, 8(1):99–118, 1977.

[31] Rajiv T. Maheswaran, Jonathan P. Pearce, and Milind Tambe. A Family of Graphical-Game-Based Algorithms for Distributed Constraint Optimization Problems. In Paul Scerri, Régis Vincent, and Roger Mailler, editors, *Coordination of Large-Scale Multiagent Systems*, pages 127–146. Springer US, 2006.

[32] Rajiv T. Maheswaran, Milind Tambe, Emma Bowring, Jonathan P. Pearce, and Pradeep Varakantham. Taking DCOP to the Real World: Efficient Complete Solutions for Distributed Multi-Event Scheduling. In *Autonomous Agents and Multiagent Systems*, pages 310–317. IEEE Computer Society, 2004.

[33] S.A.M. Makki and G. Havas. Distributed algorithms for depth-first search. *Information Processing Letters*, 60(1):7–12, 1996. cited By (since 1996)10.

[34] Amnon Meisels, Eliezer Kaplansky, Igor Razgon, and Roie Zivan. Comparing Performance of Distributed Constraints Processing Algorithms. In *AAMAS-2002 Workshop on Distributed Constraint Reasoning*, pages 86–93, July 2002.

[35] Koenraad Mertens. *An Ant-Based Approach for Solving Dynamic Constraint Optimization Problems*. PhD thesis, Katholieke Universiteit Leuven, Dec 2006.

[36] Koenraad Mertens and Tom Holvoet. Csaa: A distributed ant algorithm framework for constraint satisfaction. In Barr and Markov [4], pages 764–769.

[37] Koenraad Mertens, Tom Holvoet, and Yolande Berbers. The DynCOAA algorithm for dynamic constraint optimization problems. In Nakashima et al. [42], pages 1421–1423.

[38] Koenraad Mertens, Tom Holvoet, and Yolande Berbers. Which Dynamic Constraint Problems Can Be Solved By Ants? In Geoff Sutcliffe and Randy Goebel, editors, *Florida Artificial Intelligence Research Society Conference*, pages 439–444. AAAI Press, 2006.

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

[40] Michael D. Moffitt. On the modelling and optimization of preferences in constraint-based temporal reasoning. *Artificial Intelligence*, 175(7-8):1390–1409, 2011.

[41] Ugo Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Information Sciences*, 7:95–132, 1974.

[42] Hideyuki Nakashima, Michael P. Wellman, Gerhard Weiss, and Peter Stone, editors. *5th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2006), Hakodate, Japan, May 8-12, 2006*. ACM, 2006.

[43] Duc Thien Nguyen, William Yeoh, and Hoong Chuin Lau. Distributed Gibbs: a memory-bounded sampling-based DCOP algorithm. In Maria L. Gini, Onn Shehory, Takayuki Ito, and Catholijn M. Jonker, editors, *Autonomous Agents and Multiagent Systems*, pages 167–174. IFAAMAS, 2013.

[44] Jon Parker and Joshua M. Epstein. A distributed platform for global-scale agent-based models of disease transmission. *Association for Computing Machinery Transactions on Modelling and Computational Simulation*, 22(1):2, 2011.

[45] Jonathan P. Pearce and Milind Tambe. Quality Guarantees on k-Optimal Solutions for Distributed Constraint Optimization Problems. In Veloso [65], pages 1446–1451.

[46] Adrian Petcu. *A class of algorithms for Distributed Constraint Optimistation*. PhD thesis, École Polytechnique Fédérale de Lausanne, Oct 2007.

[47] Adrian Petcu and Boi Faltings. DPOP: A Scalable Method for Multiagent Constraint Optimization. In *International Joint Conference on Artificial Intelligence*, pages 266–271, Aug 2005.

[48] Adrian Petcu and Boi Faltings. R-DPOP: Optimal Solution Stability in Continuous-Time Optimization. In *Intelligent Agent Technology*, Nov 2007.

[49] Adrian Petcu, Boi Faltings, and Roger Mailler. PC-DPOP: A New Partial Centralization Algorithm for Distributed Optimization. In Veloso [65], pages 167–172.

[50] Christopher P. Portway. USC DCOP Repository, 2008. http://teamcore.usc.edu/dcop.

[51] Alex Rogers, Alessandro Farinelli, and Nicholas R. Jennings. Self-organising Sensors for Wide Area Surveillance Using the Max-sum Algorithm. In Danny Weyns, Sam Malek, Rogério de Lemos, and Jesper Andersson, editors, *Self-Organizing Architectures*, volume 6090 of *Lecture Notes in Computer Science*, pages 84–100. Springer, 2009.

[52] Emma Rollon. *Multi-objective Optimization in Graphical Models*. PhD thesis, Universitat Politècnica de Catalunya, 2008.

[53] Emma Rollon and Javier Larrosa. Improved Bounded Max-Sum for Distributed Constraint Optimization. In Michela Milano, editor, *Constraint Programming*, volume 7514 of *Lecture Notes in Computer Science*, pages 624–632. Springer, 2012.

[54] Azriel Rosenfeld. *An Introduction to Algebraic Structures*. Holden-Day, 1968.

[55] Francesca Rossi, Peter van Beek, and Toby Walsh, editors. *Handbook of Constraint Programming*. Elsevier, 2006.

[56] T. Schiex and G. Verfaillie. Nogood Recording for Static and Dynamic Constraint Satisfaction Problem. *International Journal of Artificial Intelligence Tools*, 3(2):187–207, 1994.

[57] R. Selten. Reexamination of the perfectness concept for equilibrium points in extensive games. *International Journal of Game Theory*, 4(1):25–55, 1975.

[58] Marius C Silaghi and Makoto Yokoo. Nogood based asynchronous distributed optimization (ADOPT-ng). In Nakashima et al. [42], pages 1389–1396.

[59] Marius-Calin Silaghi and Makoto Yokoo. Dynamic DFS Tree in ADOPT-ing. In *AAAI Conference on Artificial Intelligence*, pages 763–769. AAAI Press, 2007.

[60] Marius-Calin Silaghi and Makoto Yokoo. Revisiting ADOPT-ing and its Feedback Schemes. In *Intelligent Agent Technology*, pages 3–9. IEEE Computer Society, 2007.

[61] Marius-Calin Silaghi and Makoto Yokoo. ADOPT-ing: unifying asynchronous distributed optimization with asynchronous backtracking. *Autonomous Agents and Multi-Agent Systems*, 19(2):89–123, 2009.

[62] Reid G. Smith. The contract net protocol: High-level communication and control in a distributed problem solver. *Computers, IEEE Transactions on*, C-29(12):1104–1113, Dec 1980.

[63] Ruben Stranders, Alessandro Farinelli, Alex Rogers, and Nick R. Jennings. Decentralised Coordination of Continuously Valued Control Parameters Using the Max-Sum Algorithm. In *Autonomous Agents and Multiagent Systems*, pages 601–608, 2009.

[64] Mark Van Houdenhoven, Jeroen M. van Oostrum, Erwin W. Hans, Gerhard Wullink, and Geert Kazemier. Improving operating room efficiency by applying bin-packing and portfolio techniques to surgical case scheduling. In *Anesthesia & Analgesia*, 2007.

[65] Manuela M. Veloso, editor. *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, 2007.

[66] Gérard Verfaillie and Narendra Jussien. Constraint Solving in Uncertain and Dynamic Environments: A Survey. *Constraints*, 10(3):253–281, 2005.

[67] Gérard Verfaillie and Thomas Schiex. Solution Reuse in Dynamic Constraint Satisfaction Problems. In Barbara Hayes-Roth and Richard E. Korf, editors, *AAAI Conference on Artificial Intelligence*, pages 307–312. AAAI Press / The MIT Press, 1994.

[68] Richard J. Wallace and Eugene C. Freuder. Stable Solutions for Dynamic Constraint Satisfaction Problems. In Michael J. Maher and Jean-Francois Puget, editors, *Constraint Programming*, volume 1520 of *Lecture Notes in Computer Science*, pages 447–461. Springer, 1998.

[69] William Yeoh, Ariel Felner, and Sven Koenig. BnB-ADOPT: an asynchronous branch-and-bound DCOP algorithm. In *Autonomous Agents and Multiagent Systems*, pages 591–598. IFAAMAS, 2008.

[70] M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara. The distributed constraint satisfaction problem - formalization and algorithms. *IEEE Transactions on Knowledge and Data Engineering*, 10(5):673–685, 1998.

[71] Weixiong Zhang, Guandong Wang, Zhao Xing, and Lars Wittenburg. Distributed stochastic search and distributed breakout: properties, comparison and applications to constraint optimization problems in sensor networks. *Artificial Intelligence*, 161(1-2):55–87, 2005.

[72] Lingzhong Zhou, Abdul Sattar, and Scott D. Goodwin. Handling over-constrained problems in distributed multi-agent systems. In Balázs Kégl and Guy Lapalme, editors, *Advances in Artificial Intelligence*, volume 3501 of *Lecture Notes in Computer Science*, pages 13–24. Springer, 2005.

[73] Lingzhong Zhou, John Thornton, and Abdul Sattar. Dynamic agent ordering in distributed constraint satisfaction problems. In Tamás D. Gedeon and Lance Chun Che Fung, editors, *Australian Conference on Artificial Intelligence*, volume 2903 of *Lecture Notes in Computer Science*, pages 427–439. Springer, 2003.

[74] Lingzhong Zhou, John Thornton, and Abdul Sattar. Dynamic agent-ordering and nogood-repairing in distributed constraint satisfaction problems. In Barr and Markov [4], pages 20–26.