Robust Distributed Constraint Reasoning

Robert N. Lass, Evan A. Sultanik, Rachel Greenstadt, William C. Regli
{urlass, eas28, greenie, regli}@cs.drexel.edu

Department of Computer Science
College of Engineering
Drexel University

Abstract. Distributed constraint reasoning (DCR) has recently generated much interest due to its ability to solve many real world problems without centralizing all of the information. Many DCR algorithms, however, are prone to failure if even a single agent fails, creating a situation with not only a central point of failure, but with n-points of failure! There are three main contributions of this work. First, we define the robust DCR problem space in terms of communications failures, agent failures and observability of failed agents. Then we describe two new types of algorithm modifications and show where they and other algorithms fit into this problem space. Finally, we analyze these algorithms and discuss what future work is needed in this area.

1 Introduction

In the original work on distributed constraint satisfaction (DisCSP) [1], and in many papers that followed [2, 3], it is suggested that one of the advantages of distributed constraint reasoning (DCR)—in addition to privacy and parallelization—is that there is not necessarily a central point of failure. On the contrary, all current complete DCR algorithms[1] fail to terminate if even a *single* agents fails. Rather than a single point of failure there are actually n-points of failure! Furthermore, a "failure" might manifest itself in a number of different ways. After an "agent failure," are that agent's variables removed from the constraint graph? On the other hand, might the variables remain but the agent is no longer able to set its variables' values? Perhaps the variable has been permanently assigned, but the agent is no longer able to change its value. The correct behavior of a DCR algorithm will depend on the type of failure in question. This paper proposes a formal taxonomy of the possible types of agent failure, providing solutions for a number of types.

As an example, consider the partial constraint graph depicted in Figure 1. Agents A, B, and C are constrained with agent D. When agent D fails, these three agents need to continue solving the problem in the best way possible ("best" is made more precise later). Even though D has failed, in some cases the value it has assigned to its variable may be observable while in others it may not be observable.

This paper has two main contributions. First, we propose a new approach to DCR—called robust DCR (RDCR)—that considers robustness properties of the algorithm.

[1] S-DPOP [4] is a possible exception, and it is discussed below.

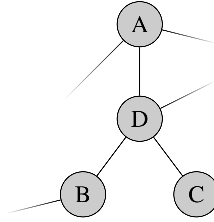

Fig. 1. A portion of a constraint graph used as the example throughout this paper. Each node represents an agent which is responsible for a single variable. The edges between nodes represent constraints between the agents' variables. In our example, agent D has failed, and agents A, B, and C find an according solution.

Second, we describe two techniques for creating DCR algorithms that solve classes of RDCR problems, and as an example show how to modify the Adopt algorithm [5] using these techniques.

The remainder of the paper is organized as follows: first an overview of DCR is presented, followed by a definition of the problem space for RDCR. We then analyze our approach to solve a class of RDCR problems, giving bounds on its performance. Finally, the exciting possibilities this research may hold for the future is discussed.

2 Background

The Distributed Constraint Satisfaction Problem (DisCSP) was introduced in [1], and was eventually generalized to the Distributed Constraint Optimization Problem (DisCOP or DCOP). There are several complete algorithms for solving DCOPs, including Adopt [5], DPOP [6], NCBB [7] and OptAPO [8]. Many variations exist on these algorithms. DCOP has been used to model many different problems in multiagent systems including those based on distributed graph coloring [8], the distributed multiple knapsack problem [9], scheduling [10], and sensor networks [10].

In the paper on S-DPOP [4], Petcu, *et al.* describe a self-stabilizing algorithm for DCOP. S-DPOP substitutes a self-stabilizing algorithm for each phase of the standard DPOP algorithm: pseudotree creation, utility propagation and value propagation. This is suitable in cases where an agent has failed and is no longer relevant to the problem. That is to say, if an agent fails and its variables are completely removed from the constraint graph, this algorithm can properly handle it.

Furthermore, it has been shown that there is a tradeoff between privacy, efficiency, and optimality [11]. For example, in this work it was shown that when running the Adopt algorithm on a pseudotree with a chain ordering, less privacy loss occurred than when running Adopt on a tree. A chain, however, will be much less efficient than a tree, as it does not parallelize the problem like a tree. Similarly, we will show later that greater resistance to certain types of failure results in more information about the problem being spread among agents. Although this is not strictly privacy loss as defined in previous work it results in information being more widely available.

3 Formalization

Before attempting to create an algorithm, we must first formally define the problem, and decide what the correct behavior ought to be. Here we define DCOP, of which DisCSP is a special case.

A "**DCOP**" is a problem in which a group of agents must distributedly choose values for a set of variables such that the cost of a set of constraints over the variables is either minimized or maximized.

Formally, a DCOP may be represented as a tuple $\langle A, V, \mathcal{D}, f, \alpha, \sigma\rangle$, where:

A is an ordered set of agents, $\{a_1, a_2, \ldots, a_{|A|}\}$;

V is an ordered set of variables, $\{v_1, v_2, \ldots, v_{|V|}\}$;

$\mathcal{D}$ is a set of domains, $\{D_1, D_2, \ldots, D_{|V|}\}$, where each $D_i \in \mathcal{D}$ is either a finite or ordered set containing the values to which its associated variable may be assigned, $\{d_{i,1}, d_{i,2}, \ldots, d_{i,|D_i|}\}$;

f is a function

$$f : \bigcup_{S\in\mathcal{P}(V)} \prod_{v_i\in S} (\{v_i\} \times D_i) \rightarrow \mathbb{N} \cup \{\infty\}$$

(where "$\mathcal{P}(V)$" denotes the power set of V) that maps every possible variable assignment to a cost. This function can also be thought of as defining constraints between variables;

α is a function $\alpha : V \rightarrow A$ mapping variables to their associated agent. $\alpha(v_i) \mapsto a_j$ implies that it is agent a_j's responsibility to assign the value of variable v_i. Note that it is not necessarily true that α is either an injection or surjection; and

σ is an operator that aggregates all of the individual f costs for all possible variable assignments. This is usually accomplished through summation:

$$\sigma(f) \mapsto \sum_{s\in\bigcup_{S\in\mathcal{P}(V)} \prod_{v_i\in S}(\{v_i\}\times D_i)} f(s).$$

n is a function, $f : V \rightarrow \{v_i\}$, mapping a variable to its neighboring variables in the constraint graph (variables it shares a constraint with).

The objective of a DCOP is to have each agent assign values to its associated variables in order to either minimize or maximize $\sigma(f)$.

A "**Context**," T, is an assignment of values to variables for a distributed constraint reasoning problem. Essentially, a context is a (possibly partial) solution to a problem. Contexts may be represented as a set of ordered pairs, such as

$$T = \{\langle v_1, 5\rangle, \langle v_2, 9\rangle\},$$

meaning that variable v_1 has been assigned value 5 and variable v_2 has been assigned value 9.

The above is solely the definition of a DCOP; to this we must now add further notation needed later to precisely define failure.

$\widetilde{A}$ is the set of failed agents, $\{\widetilde{a}_1, \widetilde{a}_2, \ldots, \widetilde{a}_{|\widetilde{A}|}\} \subset A$;
$\widetilde{V}$ is the set of variables owned by agents in $\widetilde{A}$,

$$\widetilde{V} = \bigcup_{\widetilde{a} \in \widetilde{A}} \alpha^{-1}(\widetilde{a})$$

g is a function $g : \widetilde{V} \to D_i$ that maps failed variables to their assigned values.

3.1 Problem Space

The problem space is divided along three axes: type of agent failure, observability of failed agents, and communications environment.

To begin, we must clarify the problem we are trying to solve, and explicitly state some of the problems we are not trying to solve. This work addresses what to do when failure has occurred. This work does not address how to distinguish between different types of failure, nor does it deal with the case where the agent and its associated variables leave the problem (*i.e.*, the variables are no longer in the constraint graph).

Determining what type of failure has occurred is a difficult problem, which has been studied in a number of different contexts. One good introduction to this is a survey of control theoretic techniques for fault detection, fault isolation, and fault identification [12]. Although this is still an area of active research, it has been well-studied and we do not consider it in this paper.

In some cases, a failed agent may be irrelevant to a problem and can simply be removed from the constraint network. For example, in a distributed sensor network a failed sensor may not be able to sense or otherwise. This is a simplified form of the problem considered here, and as it can be handled by existing algorithms [4] we will not consider it further. There may be more practical solutions that can take further advantage of constraint propagation that occurred before the agent left the problem, but it is not considered in this paper.

Type of Agent Failure There are two types of failure that are generally considered in the distributed systems community: stopping and Byzantine [13]. Stopping failure means that the process has stopped; it no longer sends and receives messages which, in distributed systems, renders any further computation on the part of that agent irrelevant. In addition, permanent link failure could be equivalent to stopping failure. In the case of Byzantine failure processes may exhibit arbitrary (even adversarial) behavior [14]. DCOP Algorithms exist for strictly adversarial behavior [15], but their resilience to arbitrary behavior is not clear.

Observability Even if an agent has failed, its neighbors may be able to observe the value of its variable. In this case it is *observable*. If the neighbors cannot access the value of its variable, it is *unobservable*. An example of this might be a multiagent system coordinating the flight paths for a group of airplanes [16]. If one of the agents controlling an airplane were to fail, the system may no longer have any control over it, but it could still see where it is using radar.

Communications Failure There are three communications environments that we consider. The first is *good communications*. Messages are sent and received without any loss. No real networking environment will be perfect, but this may be a reasonable assumption for many types of environments such as computers communicating over a dedicated wired LAN.

The second is an environment where messages may be delayed for an arbitrary amount of time or even lost. Due to the ubiquity of TCP this may seem unnecessary, but there are several reasons why the DCR community (and indeed, the distributed AI community) should care about message loss [17]. First, it is generally more efficient for an asynchronous algorithm to continually compute and deal with messages as they come in, rather than blocking while waiting for the next message. Secondly, if message loss will not irrevocably damage the algorithm, a communications protocol with less overhead such as UDP can be used. Finally, a message that is lost between one pair of agents may cause other messages to queue in a buffer until this is resolved by TCP, rather than allowing the agents to send messages in the meanwhile, which delays the entire system by reducing parallelization.

The third type of communications failure that can occur is link failure, meaning that the agent can no longer communicate with the rest of the agent society. Since the agent can compute but not communicate, this is functionally equivalent to stopping failure, discussed above. In both cases the agent is still constrained and can assign a value to its assigned variable(s), but it is not collaborating with its constraint graph neighbors.

4 Algorithm

This section presents two different approaches to dealing with failure. The portions of the problem space to which they are suitable is shown in Table 1. These approaches could be applied to many different algorithms, and examples are given of the Adopt algorithm [18] modified in each manner. It should be noted that these algorithms do not deal with *how* to detect failure, nor do they work if the failure is detected after the agents have assigned values.

4.1 Unary Constraint Modification

The first technique for modifying algorithms is based on the idea of unary constraints, which are already commonly used in DCOP. A unary constraint is simply a constraint over a single variable; there is a cost associated with assigning some value(s) in the variables domain in addition to costs that may be incurred by constraints with other variables. The technique is simply for each functioning neighbor of the failed agent to add a unary constraint representing the cost of a context with the values chosen by the failed agents. This allows the functioning agents to continue assigning values while taking into account the costs incurred by the constraints with the failed agent(s). This is suitable for failure in which the value chosen by the failed agent is observable.

Previously we had defined function f as:

$$f : \bigcup_{S \in \mathcal{P}(V)} \prod_{v_i \in S} (\{v_i\} \times D_i) \to \mathbb{N} \cup \{\infty\}.$$

	OBSERVABLE		UNOBSERVABLE	
	GOOD COMMUNICATIONS	MESSAGE LOSS	GOOD COMMUNICATIONS	MESSAGE LOSS
NO AGENT FAILURE	Any	Timeout Adopt	Any	Timeout Adopt
STOPPING FAILURE	Any Unary	Unary Modified Timeout Adopt	Any MiniMax	MiniMax Timeout Adopt
BYZANTINE FAILURE	Possibly M-DPOP	?	Possibly M-DPOP	?

Table 1. The RDCR problem space. The cells indicate which algorithms can be used to solve that type of RDCR problem. *Any* means that any DCR algorithm can be used, *Any Unary* means any DCR algorithm modified using the unary method in section 4 can be used, *Any MiniMax* means any DCR algorithm modified using the minimax method in section 4 can be used, and ? indicates that no known algorithm exists. Timeout Adopt refers to the algorithm described in [17] and *MiniMax Timeout Adopt* and *Unary Timeout Adopt* refer to Timeout Adopt modified using the minimax method or unary methods respectively as described in Sections 4. *Possibly M-DPOP* refers to [15] being acceptable for some situations (see Section 3.1).

This function can be thought of as defining constraints between variables by mapping every possible variable assignment to a cost. Using the unary constraint modification technique the cost function requires no modification; only the context, T, needs to be modified to take into account the values that have been assigned to failed agents. Formally, the cost function, f, calculates uses the new context, T^u, which is calculated in the following fashion:

$$T^u = \left(\bigcup_{\widetilde{v} \in \widetilde{V}} T - \{\langle \widetilde{v}, * \rangle\} \right) \cup \left(\bigcup_{v_j \in n(v_i) \wedge v_j \in \widetilde{V}} \{\langle v_j, g(v_j) \rangle\} \right).$$

What this says is that the new context, T^u, will contain the same variable assignments as T, but will include the values that have already been assigned to failed, observable variables.

The Adopt paper[2] [5] does not explicitly give the procedures (pseudocode) for calculating cost, but does give a formal definition of how costs are calculated. The cost of a given, singular value assignment is calculated by summing the cost of that value assignment with each of the other value assignments in the current context. It is given as:

$$\delta(d_i) = \sum_{(v_j, d_j) \in T} h_{ij}(d_i, d_j),$$

[2] The use of x_i and f_{ij} in the original paper are replaced with v_i and h_{ij} in this paper to maintain consistency with this paper's notation and to avoid overloading terms.

which can be amended to handle failure by replacing T with T^u:

$$\delta(d_i) = \sum_{(v_j,d_j)\in T^u} h_{ij}(d_i,d_j),$$

4.2 MiniMax Modification

The second technique is based on the well-known MiniMax algorithm [19] from game theory, in which a player seeks to minimize the maximum cost that an opponent can force him or her to endure. The failed, unobservable agents are treated as opponents, and surviving agents must seek to minimize the maximum cost that the failed agents can inflict upon the system. This is suitable for failure in which the value chosen by the failed agent in unobservable.

Formally, instead of using T as defined above, we now use T^m, which is the context modified such that each failed variable is assigned the most expensive v_i, the cost of a given context, T, in the following fashion:

$$T^m = \left(\left(\bigcup_{\widetilde{v}\in\widetilde{V}} T - \{\langle\widetilde{v},*\rangle\} \right) \cup \left(\bigcup_{v_j\in n(v_i)\wedge v_j\in\widetilde{V}} \{\langle v_j,\lambda(v_j)\rangle\} \right) \right),$$

where λ is defined as:

$$\lambda(T,v_j) = \max_{d\in D_j} f(T\cup\{\langle v_j,d\rangle\}).$$

As in the previous section, modifications to Adopt to comply with this technique will now be described. Adopt can be amended to handle failure by using T^m instead of T:

$$\delta(d_i) = \sum_{(v_j,d_j)\in T} h_{ij}(d_i,d_j),$$

becomes:

$$\delta(d_i) = \sum_{(v_j,d_j)\in T^m} h_{ij}(d_i,d_j),$$

Contingency Edges Furthermore, in the case of unobservability, it will often benefit the MiniMax type of algorithm to have the values of the other neighbors of the failed variable available in their context when evaluating their own values. To use this paper's example, it would be helpful for A, B, and C to know each others' values when assigning their own values, because all of them are constrained with the failed variable. If A, B, and C all choose red and D chooses red (the worst case), the cost would be four. If A, B, and C instead choose red, green and blue respectively, the cost would be one in the worst case because D could only conflict with one of them. The actual action they take will depend on the scenario, and whether it makes sense to minimize the worst case, the expected value, the best case, and so on.

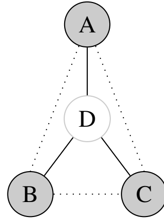

Fig. 2. After agent D fails, agents A, B, and C add contingency edges between each other. Solid lines indicate edges in the constraint graph and dotted lines indicate contingency edges.

In order to know that they must exchange contexts, Agents A, B, and C must know that they are connected to D. We propose doing this by adding *contingency edges*, which can help contain failures to a localized portion of the agent topology. We acknowledge that this will not work some cases, such as sensor networks that do not have direct communications links between all nodes and do not route messages.

Complete DCOP algorithms generally have three phases: first a pseudotree is constructed, then agents exchange messages with one another, and finally the agents pick values to assign their variables. Adding contingency edges requires another phase between pseudotree construction and message exchange. In this *exchange phase*, each agent constructs a list of their neighbors, puts the list into a message, and sends the message to all of their neighbors.

Agents with an edge connected to a failed agent then add *contingency edges* between themselves immediately after the agent fails, as depicted in Figure 4.2. Two agents connected by a contingency edge are called *contingency agents*. The contingency agents also exchange context information whenever they exchange context information with any other agents in the constraint graph.

This scheme works so long as two adjacent agents do not fail concurrently (or at least, before the neighbors of the first to fail have a chance to become contingency). This could be prevented by having neighbors of neighbors exchange neighbor lists. During the exchange phase, after having received a list of all their neighbors' neighbors, each agent distributes this set of lists to their neighbors. We call this *2-contingency*. If the neighbor exchange continues such that each agent has a list of all the agents within distance p in the constraint graph, we call it p-contingency. The number of rounds required for this algorithm is exactly p rounds, where $p \leq |A|$.

4.3 A Hybrid Approach

Realistically, many scenarios may have cases where failed agents' variables are observable, and others may have cases where they are not observable. Both of these could happen at the same time. Therefore, a hybrid approach combining the unary and Mini-Max modified contexts is needed. In terms of Adopt, we would need to replace T with $T^u \cup T^m$ to take into account the observable and unobservable variable assignments. The δ function would now look like this:

$$\delta(d_i) = \sum_{(v_j, d_j) \in (T^u \cup T^m)} h_{ij}(d_i, d_j),$$

5 Analysis

As this is a new method for DCR, there are no accepted techniques for evaluating RDCR algorithms. In this section, these techniques are evaluated using two different methods. The first is to examine the properties of the algorithm. In other words, what do they give you? The second is comparing the techniques to existing alternatives.

We propose three techniques for evaluating RDCR algorithms and apply them to the modified forms of Adopt described in Section 4.

5.1 Properties of the Technique

This subsection discusses properties of the technique. How robust it is to agent failure is discussed, and bounds are given on the solution quality.

Robustness to Failure How Robust are these techniques to failure? In the case of unary modification, the system can continue to function regardless of the number of agents that fail, although solution quality obviously degrades as more agents fail (see the next section for a precise description of this phenomena). In cases of MiniMax modification, the system can function well only so long as the neighbors of failed variables are able to create contingency edges and exchange contexts. So the extent of the system's robustness depends on the level of contingency employed.

Solution Quality When using the unary modification technique, the cost of all the constraints involving failed agents can be calculated as:

$$f_u = f\left(\bigcup_{v_j \in \widetilde{V}} \{\langle v_j, g(v_j)\rangle\}\right).$$

Let T^* be an optimal variable assignment. The cost of the failed system, sc, is bounded by $sc \leq f_u + f(T^*)$. The failed agents have selected values that are beyond the control of the system. The functional agents have taken the cost of these values into account when selecting their own values, making this the best any system can do given the control constraints.

The cost of all the constraints involving failed agents can be calculated as:

$$f_m = f'\left(\bigcup_{v_j \in \widetilde{V}} \{\langle v_j, \lambda(v_j)\rangle\}\right).$$

The cost of the failed system in the MiniMax modification case, sc, can be bounded by $sc \leq f_m + f(T^*)$. The failed agents have selected values that are beyond the control of

the system. The functional agents have taken the cost of these values into account when selecting their own values, making their valuation the best any system could find given the control constraints.

Finally, consider a hybrid system where some of the variables are observable and some are not. We can represent the error of the observable variables by f_u as defined above, and the unobservable variables by f_m as defined above. The cost of the failed system, sc can then be bounded by $sc \leq f_m + f_u + f(T^*)$.

Having such bounds on the solution quality is a always a good property. What makes these bounds interesting is that they are the best an algorithm can do given the restriction of failed agents and a lack of global knowledge. In the unary case, the algorithm selects the best values that it can given the values chosen by the failed agents, which is no longer controllable. In the MiniMax case, the algorithm selects the values that minimize the worst case scenario. One could argue that the "best" values might be those with the lowest cost in the average case or even in the best case, it depends on whether one wants best average-case performance, best-case performance or worst-case performance which will probably be scenario dependent.

5.2 Comparison to Alternatives

Another way of evaluating these techniques is to compare them to existing alternatives. First, the techniques are compared to centralized constraint reasoning. The techniques are then compared to a standard DCR algorithm. Finally, the techniques are evaluated against a non-optimal local search algorithm.

Centralization In a centralized approach, the individual agents send their information to a central constraint solver. The central solver calculates an optimal set of values and sends these back out to the individual agents. In this model there are two cases of interest: central node failure and non-central node failure.

If the central node fails, then the entire system could be considered to have failed. Each agent will not have any information about the values to assign to its variables. Any solution this system could produce will be worse than any solution the modified Adopt variants could produce.

By this comparison, almost any algorithm that can tolerate agent failure will perform well.

Complete DCR Algorithms As for traditional DCR algorithms, Adopt [5], DPOP [6] and its variants (with the exception of S-DPOP, discussed below), NCBB [7], OptAPO [8] fail whenever a single agent fails. There is nothing specified in these papers about how to behave when an agent fails, so in the case of agent failure—if implemented faithfully to their original specifications—the algorithms will most likely hang while waiting for a message that will never come.

S-DPOP [4] is a variant of DPOP designed to be super-stabilizing. It is comprised of three self-stabilizing algorithms that run concurrently to maintain the system: depth-first search (DFS), utility propagation, and value propagation. DFS is used to maintain the

pseudotree, utility propagation to ensure agents know current utility values for their descendants, and value propagation to ensure they know current value assignments for the ancestors. As mentioned in Section 3.1, this algorithm is suitable for problems where the topology of the constraint network or the cost functions may change. It is not, however, meant to address problems where the agent fails but the variable remains relevant to the constraint network.

Local Search Algorithms There already exists another class of algorithms that are resilient to almost any type of failure. These are local search algorithms such as the Distributed Stochastic Search (DSA) family of algorithms [20]. These solutions are robust to failure in that if an agent stops communicating with its neighbors it will not cause the system to freeze or crash. However, these algorithms are not complete, regardless of whether any agents fail.

6 Conclusions

The guarantees of DCR fall apart under even the most modest failure scenarios. This must be addressed before DCR is ready to be deployed in the real world. Furthermore, that simply ignoring the failed agent is not the best course of action. This paper introduced the problem space for robust distributed constraint reasoning, two algorithms for solving a portion of this problem space, and an analysis of these algorithms. The problem space was divided based on the type of agent failure, whether or not message loss is considered, and whether or not the variables of the failed agents are observable. Existing techniques, plus the two techniques presented in this paper provide coverage of the entire problem space except for the case of Byzantine failure with message loss. The two techniques presented for modifying an existing DCOP algorithm, unary constraint modification and MiniMax modification, provide optimal degradation by yielding the best worst-case solutions.

One piece of future work is obvious from Table 1: finding algorithms that can function under Byzantine failure in environments with message loss. Other work includes empirical analysis of these techniques and integration with other DCR problems, such as Dynamic DCR.

References

1. Yokoo, M., Ishida, T., Durfee, E.H., o Kuwabara, K.: Distributed constraint satisfaction for formalizing distributed problem solving. In: Proceedings of the 12th International Conference on Distributed Computing Systems. (1992) 614–621
2. Petcu, A.: A Class of Algorithms for Distributed Constraint Optimization. PhD thesis, Swiss Federal Institute of Technology (2007)
3. Jung, H., Tambe, M., Kulkarni, S.: Argumentation as distributed constraint satisfaction: applications and results. In: Proceedings of the Fifth International Conference on Autonomous Agents, ACM New York, NY, USA (May 2001) 324–331
4. Petcu, A., Faltings, B.: S-dpop: Superstabilizing, fault-containing multiagent combinatorial optimization. In: Proceedings of the National Conference on Artificial Intelligence, Pittsburgh, Pennsylvania, USA, AAAI (July 2005) 449–454

5. Modi, P.J., Tambe, M., Yokoo, M.: Adopt: Asynchronous distributed constraint optimization with quality guarantees. Artificial Intelligence **161** (2005) 149–180
6. Petcu, A., Faltings, B.V.: A scalable method for multiagent constraint optimization. In: Proceedings of the International Joint Conference on Artificial Intelligence. (2005) 266–271
7. Chechetka, A., Sycara, K.: No-commitment branch and bound search for distributed constraint optimization. In: Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems, Hakodate, Japan, ACM Press (2006) 1427–1429
8. Mailler, R., Lesser, V.: Solving distributed constraint optimization problems using cooperative mediation. In: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems. (2004) 438–445
9. Kopena, J.B., Sultanik, E.A., Lass, R.N., Nguyen, D.N., Dugan, C.J., Modi, P.J., Regli, W.C.: Coordination of first responders under communication and resource constraints. IEEE Internet Computing: Special Issue on Crisis Management (January/February 2008)
10. Maheswaran, R.T., Tambe, M., Bowring, E., Pearce, J.P., Varakantham, P.: Taking DCOP to the real world: Efficient complete solutions for distributed multi-event scheduling. In: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems-. Volume 1. (2004) 310–317
11. Greenstadt, R., Pearce, J.P., Tambe, M.: Analysis of privacy loss in distributed constraint optimization. In: Proceedings of the National Conference on Artificial Intelligence. (2006) 647
12. Pettersson, O.: Execution monitoring in robotics: A survey. Robotics and Autonomous Systems **53** (2005) 73–88
13. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann (1996)
14. Lamport, L., Shostak, R., Pease, M.: The byzantine generals problem. ACM Transactions on Programming Languages and Systems **4**(3) (1982) 382–401
15. Petcu, A., Faltings, B., Parkes, D.: M-DPOP: Faithful distributed implementation of efficient social choice problems. Journal of Artificial Intelligence Research (JAIR) **32** (2008) 705–755
16. Šišlák, D., Samek, J., Pěchouček, M.: Decentralized algorithms for collision avoidance in airspace. In Padgham, Parkes, Mueller, Parsons, eds.: Proceedings of Seventh International Joint Conference on Autonomous Agents and Multiagent Systems, Estoril, Portugal (May 2008) 543–550
17. Modi, P.J., Ali, S.M.: Distributed constraint reasoning under unreliable communication. In: Proceedings of the Distributed Constraint Reasoning Workshop. (2003)
18. Modi, P.J., Shen, W.M., Tambe, M., Yokoo, M.: ADOPT: asynchronous distributed constraint optimization with quality guarantees. Artificial Intelligence **161**(1-2) (2006) 149–180
19. von Neumann, J.: Zur theorie der gesellschaftsspiele. Mathematische Annalen **100** (1928) 295–320
20. Zhang, W., Wang, G., Xing, Z., Wittenburg, L.: Distributed stochastic search and distributed breakout: properties, comparison and applications to constraint optimization problems in sensor networks. Artificial Intelligence **161**(1–2) (2005) 55–87