SBDO: A New Robust Approach to Dynamic Distributed Constraint Optimisation

Graham Billiau, Chee Fon Chang, and Aditya Ghose

Decision Systems Lab
School of Computer Science and Software Engg
University of Wollongong, NSW, Australia
{gdb339,c03,aditya}@uow.edu.au

Abstract. Dynamic distributed constraint optimisation problems are a very effective tool for solving multi-agent problems. However they require protocols for agents to collaborate in optimising shared objectives in a decentralised manner without necessarily revealing all of their private constraints. In this paper, we present the details of the Support-Based Distributed Optimisation (SBDO) algorithm for solving dynamic distributed constraint optimisation problems. This algorithm is complete wrt hard constraints but not wrt objectives. Furthermore, we show that SBDO is completely asynchronous, sound and fault tolerant. Finally, we evaluate the performance of SDBO with respect to DynCOAA for Dyn-DCOP and ADOPT, DPOP for DCOP. The results highlight that in general, SBDO out performs these algorithms on criteria such as time, solution quality, number of messages, non-concurrent constraint checks and memory usage.

1 Introduction

Dynamic Distributed Constraint Optimisation Problems (DynDCOP) are a problem domain that has not been well explored. DynDCOPs allow us to model problems that can not be assumed to be static, that is they change so frequently that by the time a DCOP solver has found a solution it is already obsolete. DynDCOPs are very useful for modelling and solving multi-agent coordination and planning problems. These problems appear in many areas such as scheduling patient treatment in a hospital or managing the airspace above an airport. As DynDCOP is an extension of the well explored Distributed Constraint Optimisation Problem (DCOP), techniques utilised to solve DCOP present a good foundation.

Very few of the DCOP algorithms consider what happens when agents fail. The max-sum algorithms [11] have been shown to be robust even when 90% of messages are not delivered. While none of the others consider what happens when agents fail. There are many reasons, such as hardware failures or malicious attack that may cause an agent to fail. It is particularly important to be able to continue solving even when agents fail in dynamic solvers, as they are often expected to run continuously for a long duration.

N. Desai, A. Liu, and M. Winikoff (Eds.): PRIMA 2010, LNAI 7057, pp. 11–26, 2012.
© Springer-Verlag Berlin Heidelberg 2012

1.1 Related Work

At this time there are only two other algorithms that can solve DynDCOPs, Dynamic Constraint Optimisation Ant Algorithm (DynCOAA)[6] and Self-Stabilising Distributed Psuedo-tree Optimisation Procedure (S-DPOP)[8]. Of these two DynCOAA is incomplete and S-DPOP is complete. Neither of these two algorithms consider the possibility of agent failure, so are unable to recover from failures.

As DynDCOP is an extension of the well explored Distributed Constraint Optimisation Problem (DCOP), techniques utilised to solve DCOP present a good foundation. There are a large number of DCOP algorithms, such as ADOPT [7], NCBB [1], DALO [3] and Divide-and-Coordinate [12]. As none of these algorithms are currently capable of solving dynamic problems we do not consider them further.

In section 2, we will present the Support Based Distributed Optimisation algorithm (SBDO) which improves on the existing DynDCOP solvers by being completely asynchronous, fault tolerant and having no hierarchy among agents. Section 3 describes the performance results comparison from the dynamic problems, the fault tolerance and static problem dimension. In section 4, we present the conclusions.

2 Support Based Distributed Optimisation

SBDO is an extension of the SBDS algorithm[2]. SBDS is a complete Distributed Constraint Satisfaction Problem solver. SBDO extends it by adding a local search mechanism for optimising the solution found while maintaining the completeness wrt hard constraints. SBDO also adds support for solving dynamic problems.

We define DynDCOPs as follows. Our definitions differ to that in the literature as we treat hard constraints and soft constraints/objectives differently.

Definition 1. *A Constraint Optimisation Problem (COP) is a tuple* $\langle \mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{R} \rangle$ *where* $\mathcal{X}$ *is a set* $\{x_1, \ldots, x_n\}$ *of variables,* $\mathcal{D}$ *is a set* $\{d_1, \ldots, d_n\}$ *of variable domains,* $\mathcal{C}$ *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 utility functions defined over* $\mathcal{X}$*.*

Definition 2. *A Distributed Constraint Optimisation Problem (DCOP) is a tuple* $\langle \mathcal{A}, COP, \mathcal{C}, \mathcal{R} \rangle$ *where* $\mathcal{A}$ *is a set* $\{a_1, \ldots, a_p\}$ *of agents, COP is a set* $\{COP_1, \ldots, COP_p\}$ *of disjoint COPs,* $\mathcal{C}$ *is a set* $\{c_1, \ldots, c_m\}$ *of shared constraints and* $\mathcal{R}$ *is a set* $\{r_1, \ldots, r_q\}$ *of shared utility functions.*

The shared constraints and utility functions are defined over variables from several different COPs.

Definition 3. *A Dynamic Distributed Constraint Optimisation Problem (DynDCOP) is a sequence* $\langle DCOP_1, \ldots, DCOP_n \rangle$ *where each DCOP differs from the previous one by an added or removed constraint/objective/agent. The goal is to find and maintain a solution where all the constraints are satisfied and the objective function optimised.*

We assume the existence of a global objective function that the collection of agents seeks to optimise, but we require that it must be possible to decompose this function into agent-specific objective functions such that the optimal assignment of variables for the decomposed set of objective functions corresponds to the optimal assignment for the global objective.

Note that to maintain generality of this discussion, we leave the details for the decomposition up to the designer. However, each objective function must return a value proportionate to how good the partial solution is, a utility value, such that a better solution returns a higher utility value. The utility values returned by all of the objective functions must be comparable and can be aggregated.

To further increase the generality of the algorithm, shared objectives can be used as well as local objectives. Shared objectives are used when a (sub)objective can not be decomposed to include only the variables of one agent. In this case the objective can be shared between the agents that together control the variables used in the objective. The objective is evaluated by any of the agents that share it as soon as that agent knows an assignment to all the variables in the objective. The utility returned by the shared objective is added to the utility of the agent's local objective. If the agent does not have enough information to evaluate the objective it is ignored and only the agent's local objective is used.

2.1 Communication

The physical communication channels that agents must use to communicate are never perfect, so it is desirable for algorithms to be able to tolerate messages arriving in random order. That is the messages sent between two agents may not arrive in the same order they were sent, or they may never arrive at all. The proposed algorithm is robust against messages arriving in random order but not robust against message loss.

The most common message used for communication in SBDO is an 'isgood', which is very similar to a partial assignment and is in part inspired by techniques used in formal argumentation, where the notion of an argument is used to encode alternate points of view.

Definition 4. *The neighbour graph is an undirected graph $\langle N, E\rangle$. N is the set of agents and $E \subseteq N \times N$ such that there is an edge $\{A_i, A_j\}$ iff there exists a shared constraint or a shared objective defined over both A_i and A_j.*

Definition 5. *Given a $DCOP = \langle \mathcal{A}, COP, \mathcal{C}, \mathcal{R}\rangle$. An isgood is a sequence $\langle A_1, \ldots, A_n\rangle$ of assignments such that the sequence is a simple path through the neighbour graph. Each assignment is a triple $\langle a, \{\langle x_1, \mathcal{D}_{1i}\rangle, \ldots, \langle x_n, \mathcal{D}_{nj}\rangle\}, utility\rangle$ such that none of the constraints in the DCOP are violated. The total utility of an isgood is the aggregation of the utilities of all the assignments within it. As such an isgood encodes a partial solution to the problem as well as the relative utility of the partial solution.*

An isgood can be considered as an argument, in which case the first $n-1$ assignments form the justification and the last assignment is the conclusion. As

in formal argumentation theories, an argument may attack/defeat other arguments and the agent receiving these potentially competing arguments must pick the winning argument. Because of this, each agent attempts to send stronger arguments over time to influence their neighbours.

Definition 6. *The ordering over isgoods is: First the total utility of the isgoods is compared, with higher being better. If they are equal then the number of assignments in each isgood is compared, with more being better. Finally if they are equal then one is picked randomly but consistently.*[1] *That is, if an agent picks isgood A over isgood B then in all future comparisons it will choose A over B.*

Instead of using an ordering over the variables, which causes problems in dynamic environments, we use a total ordering over the partial solutions, or isgoods. This ordering is needed so that the solution can be optimised as well as to prevent cyclic behaviour. Whenever we refer to one isgood being better than another in this paper it is with respect to this ordering

To avoid cycles of oscillating values which might occur because there is no variable ordering we increase the length of successive isgoods that are sent. This is achieved by recording the last isgood sent and attempting to send a longer one. As any cycle must be finite eventually the isgoods being sent will contain the cycle itself. If the cycle is made up of inconsistent values then a nogood will be generated, breaking the cycle. Else the cycle breaking mechanism of `update_view()` (alg. 2) will break the cycle. The proofs of soundness and termination from SBDS[10] still hold. Due to space limitation, readers are directed to [2] for details.

Rather than using all the information contained in all the isgoods that an agent has received, which is often inconsistent. Each agent picks a single isgood to use as the justification for the assignments to its own variables. The agent who has sent the best isgood is chosen as the support for the agent. The isgood that agent sent is used as the basis for the agents view.

Definition 7. *Given a DCOP* $\langle \mathcal{A}, COP, \mathcal{C}, \mathcal{R} \rangle$*. A nogood is a pair* $\langle P, C \rangle$ *where* P *is a set of variable value pairs* $\{\langle x_1, \mathcal{D}_{1i} \rangle, \ldots, \langle x_n, \mathcal{D}_{nj} \rangle\}$ *forming a partial assignment and* $C \subset \mathcal{C}$*, is the justification such that* P *violates at least one constraint in* C*. As such a nogood represents a partial solution that is proven to not be part of any global solution.*

Hard constraints are handled differently to objectives in order to guarantee that any solution found will satisfy all of the hard constraints. Nogoods with justifications [10] are used as these allow us to guarantee that all the hard constraints are satisfied (as shown in [2]) as well as allowing obsolete nogoods to be identified after hard constraints are removed from the problem.

Due to the dynamic nature of the input problem the algorithm never terminates (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, since new inputs from the environment, in the form of added or deleted variables/constraints/objectives might invalidate them).

[1] Cryptographic hash functions can provide a suitable comparison.

2.2 Dynamic Problems

Unlike other dynamic algorithms we do not explicitly model the concept of solution stability. Instead we assume that if there is a cost associated with changing the value of a variable the agent takes it into account in its local objective function(s).

Most of the changes to the problem that can occur in a dynamic system are straightforward to implement, except for removing hard constraints (which we discuss later). Several messages are required to communicate any changes to the problem to the agents: add constraint, pre-remove constraint, post-remove constraint, 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 post-remove constraint they are assumed to be sent by the environment. Only the agents that control the variables involved in the objective or hard 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 that agent.

When a hard 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. 3) and `post-remove constraint()` (alg. 5) are required to ensure correctness.

To catch any nogoods that arrive after the constraint removed message that makes them obsolete, the agent must also check the removed constraints it knows of when it receives a new nogood. If the nogood is obsolete then it is discarded and the associated counter decremented.

2.3 Fault Tolerance

Due to the nature of the algorithm, when an agent fails it has a minimal impact on the other agents. Unlike algorithms that impose a hierarchy, agents do not require a message from the failed agent(s) before they can continue processing. Instead all agents just continue oblivious to the fact that an agent has failed. The only limitation is that the value assigned to affected variables can not change, other agents must continue using the last known value for the variables.

When an agent fails all its knowledge regarding sent and received isgoods is lost. This effectively means that messages have been lost, which this algorithm

can not account for. So when the failed agent restarts it must request that its neighbours send it the last isgood and all nogoods that they sent to this agent, as well as the last isgood and all nogoods they have received from this agent. This prevents most knowledge loss and allows the failed agent to resume solving faster. But if two neighbouring agents both fail at the same time then some information is irretrievably lost.There is a simple extension to the algorithm that will ensure that will ensure it functions correctly wrt agent failure and random message order. Unfortunately there is not enough space to present it here. Only accounting for agent failure is affected by this issue.

Theorem 1. *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.

When two or more 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. Because of this each agents set of sent and received messages are still consistent. Therefore the procedure for removing obsolete nogoods is still correct.

Theorem 2. *SBDO can continue solving when one or more agents fail.*

Proof. Because of the flat communication model no agent is ever waiting for a message from a specific agent before they can continue solving (they may be waiting for a message from any agent, but if this is the case they currently have no work to do). Because of this the other agents can always continue to solve the problem, using the last known value for the agents variables. When the agent restarts it is immediately informed of the current state of the problem so that it can resume solving.

2.4 Algorithm

Each agent must store the following information:

- view. This is an isgood consisting of the isgood received from support + an assignment to all this agents variables.
- recv(A). This is a mapping from an agent A to the last isgood received from that agent.
- nogoods. This is an unbounded store of all nogoods received.
- sent(A). This is a mapping from an agent A to the last isgood sent to that agent.

Algorithm 1. main()

```
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
      else
        Add N to nogoods
        for All neighbours A do
          if There is no valid assignment to myself wrt recv(A) then
            send_nogood(A)
    for All received environment messages do
      Process message
    for All received isgoods I do
      Let A be the agent who sent I
      set recv(A) to I
      if There is no valid assignment to myself wrt I then
        send_nogood(A)
    update_view()
    Let I be the best isgood in recv(A)
    if I is better than view then
      Set support to the agent that sent I
      Let view be recv(support) extended by a valid assignment to all local,
      public variables, chosen greedily
    for All neighbours A do
      if self and A are the first two variables in view then
        if view ≮ sent(A) or sent(A) is not consistent then
          Send view to A
          Set sent(A) to view
      else
        Let length be the longest sub-isgood that can be sent to A
        Let preferred be 0
        if sent(A) is ⟨⟩ or sent(A) ⋢ view then
          Set preferred to |sent(A)| + 1
        if sent(A) < recv(A) and view is inconsistent with recv(A) then
          Set preferred to max(preferred, |recv(A)| + 1)
        if preferred > 0 then
          Let I be an isgood such that I ⊑ view and |I| =
          min(length, preferred)
          Let U be the utility of I as returned by the local objective function
          + any shared objectives
          Set the utility of the assignment to self in I to U
          Send I to A
          Set sent(A) to I
    Wait until at least one message has been received
end
```

Algorithm 2. update_view()

```
begin
    Let view' be recv(support) extended by a valid assignment to all local, public
    variables, chosen greedily
    Let V be the first variable assigned in view'
    if scope(view') = scope(view) or view is better than view' or the assignment
    to V is the same in view' and recv(A) or the assignment to V is unequal in
    view and recv(A) then
        Set view to view'
end
```

Algorithm 3. pre-remove_constraint (M)

```
begin
    Let C be the removed constraint
    for Each neighbour A do
        Let counter be 0
        for Each nogood N sent to A do
            if N contains C as part of its justification then
                Increment counter by 1
                Delete N from sent nogoods
        if counter > 0 then
            Let M be a new constraint removed message with C and counter
            Send M to A
end
```

– support. The agent that this agent is using as its support.
– sent-nogoods. This is an unbounded store of all nogoods sent.
– removed-constraints. An unbounded store of received remove constraint messages.

We use the notation $A \sqsubseteq B$ to say that A is a sub-isgood of B. By sub-isgood we mean that A is the tail (or entirety) of B, $|A|$ to denote the number of assignments in A and scope(A) is the set of variables that are assigned in A.

Each agent greedily chooses what agent to use as its support and the values to assign to its own variables. As each agent may control many variables, each agent requires its own centralised Dynamic COP solver. Because of the way the support is selected a collection of agents can combine to cause an agent that has chosen sub-optimal assignments to change its assignments.

The basic steps each agent takes are quite simple. First it processes all the messages in its message queue. Then it decides what values to assign to its own variables. Last it sends all of its neighbours a message telling them what values it has chosen for its variables.

Processing messages starts with all of the nogoods received. Nogoods are processed first in case they are later rendered obsolete by a message from the

Algorithm 4. send_nogood(A)

```
begin
    Let N be a nogood derived from recv(A)
    Send N to A
    Set recv(A) to none
    if support = A then
        Set support to self
end
```

Algorithm 5. post-remove_constraint(M)

```
begin
    Let C be the constraint referenced
    if removed-constraints already contains a message regarding C then
        Increment the counter of that message by the counter in M
    else
        for Each received nogood N do
            if N is justified by C then
                Delete N
                Decrement the counter by 1
        if counter ≠ 0 then
            Add M to removed-constraints
        pre-remove constraint(C)
end
```

environment and because one of them might invalidate one of the isgoods in the message queue. When a nogood is received it is added to the set of all known nogoods. Once all nogoods are processed the received isgoods must be rechecked to see if they are now inconsistent with this agent's assignment. If so, the isgood's sender must be informed by sending a nogood. This will force the sender to change their value in the next iteration. Next all environment messages are processed. The order within this group doesn't matter, but they may affect how the isgoods are processed. Finally, the received isgoods are processed. First, recv(A) is updated with this most recent isgood, then it checks if there is a valid assignment to its own variable. If there isn't, a nogood is created and sent back to the agent that sent the isgood. This will force the sender to change their value in the next iteration.

While the processing of most environment messages is straightforward, removing constraints requires special mention. When a constraint is removed from the problem all of the nogoods that were generated because of that constraint must also be removed. This is made more difficult because it is possible for the nogood message to arrive after the pre-remove constraint message that makes it obsolete. In order to ensure they are all deleted each agent must also maintain a store of all the nogoods it has sent and who it sent them to. When a pre-remove constraint message is received by an agent it checks its sent nogood store to

see if any of its neighbours must be notified. If any of the nogoods have the removed constraint as part of their justification, they are now obsolete and the agents neighbour must be notified. To notify the neighbour, this agent sends a post-remove constraint message with the constraint that has been removed and the total number of nogoods sent to that agent that are made obsolete.

Whenever an agent receives a post-remove constraint message it must go through its store of received nogoods and delete any that have this constraint as part of their justification. For each one that is deleted, the counter of total obsolete nogoods in the post-remove constraint message is decremented. When the counter reaches zero, all of the obsolete nogoods have been deleted and the post-remove constraint message 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 above. If an agent receives two or more post-remove constraint messages for the same constraint, the counters are simply added together.

Now that the agent has the most recent information about its environment, it can choose the best assignments for its own variables. This will normally require a centralised COP solver.

After the agent has updated its view it then checks to see if one of the other agents would make a better support than the current one. To do so, it picks the best isgood out of all of the isgoods it has received, then compares it with its view. If the isgood is better then it changes its support to the agent which sent the best isgood and then has to call `update_view()` (alg. 2) again to update its view. If its view is better then it keeps its current support.

Finally the agent must communicate changes to its local state to its neighbours. If it detects that it is part of a cycle with the agent it is currently sending an isgood to then it must send its entire view to that agent. Unless its view is worse than the last isgood sent to that agent. In which case it postpones sending a message to prevent cyclic behaviour. If it does not detect a cycle then it must decide how long an isgood to send. If the agent is updating obsolete information that it sent earlier then it attempts to send a longer isgood than sent previously. If the agent is in conflict with the agent than it also attempts to send a longer isgood than was received from the agent. However obviously it can't send an isgood longer than its view, but it also can not send an isgood that is self supporting i.e. if view is $\langle\langle B,\langle b,1\rangle,4\rangle,\langle C,\langle c,5\rangle,20\rangle,\langle A,\langle a,1\rangle,3\rangle,\langle D,\langle d,3\rangle,15\rangle\rangle$ and sending an isgood to A then the maximum length is 3.

2.5 Example

Example 1. Consider the following constraint optimisation 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 hard constraint, $\theta<\gamma$. The utility of the best assignment is 2, and the worst is 0.

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. 2 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. All agents then inform their neighbours of their decision by sending isgoods. Δ sends the isgood $\langle\langle \Delta, \{\langle \delta, 1\rangle\}, 0\rangle\rangle$ to Θ, Θ sends the isgood $\langle\langle \Theta, \{\langle \theta, -1\rangle\}, 2\rangle\rangle$ to Δ and Γ and Γ sends the isgood $\langle\langle \Gamma, \{\langle \gamma, -1\rangle\}, 2\rangle\rangle$ to Θ.

When Θ receives the isgood from Γ, it notices that the isgood is inconsistent with its knowledge, as there is no value in its domain less then -1. So Θ sends the nogood $\langle\{\langle \gamma, -1\rangle\}, \{\theta < \gamma\}\rangle$. After receiving the isgoods all the agents decide which agent to use as their support. Θ has to chose between itself and Δ. 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 don't send new isgoods, while Δ sends the isgood $\langle\langle \Delta, \{\langle \delta, 2\rangle\}, 2\rangle\rangle$ to Θ. Normally it would include the assignment to θ as well, but that would create a circular argument, so the assignment to Θ is trimmed. Next, Γ receives the nogood from Θ and so is forced to change its assignment to $\gamma = 0$ and sends another isgood to Θ with its new assignment. Simultaneously Θ receives the new isgood from Δ, but does not make any changes because of it, so does not send a new isgood.

Then the problem changes. The constraint $\theta < \gamma$ is removed from the problem. So the environment sends 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 constraint removed message $((\theta < \gamma), 1)$ 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 isgood to. As no agents have any messages to send the network has reached quiescence.

3 Results

To evaluate SBDO, we implemented it using Python and compared it with the two other DynDCOP algorithms, S-DPOP and DynCOAA. We used the reference implementation of S-DPOP[4], written in Java and we implemented DynCOAA and SBDO in python. We used the parameters for DynCOAA that are recommended by its authors [6], with 15 ants in each swarm. The different implementation languages mean that the memory and time used by each algorithm can't be compared directly. The Quality, Non-Concurrent Constraint Checks (NCCCs)[5], and messages required are independent of the implementation and so still directly comparable.

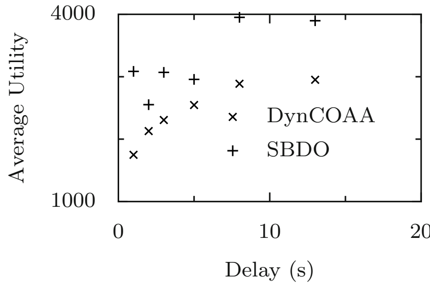

Fig. 1. Average solution quality per time step

The test platform was an AMD Athlon X2 6000+ processor with 4GB of RAM running OpenSolaris 10 release 06/09. Memory usage was measured by using DTrace to count all anonymous memory allocations and deallocations.

We used three sets of test problems: easy, moderate and hard. The easy set consists of the 120 handcrafted meeting scheduling problems provided in [9]. 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 12 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 16 randomly generated meeting scheduling problems. These problems have between 12 and 48 variables with a constraint density between 1.750 and 4.000. We ran each problem ten times to ensure the results represent the average performance of each algorithm.

3.1 Dynamic Problems

To evaluate SBDO's performance on dynamic problems we compared it against DynCOAA on the moderate and hard sets of 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, the utility of the current solution calculated, then two of the hard constraints were randomly replaced then the algorithm resumed. The problems objective function was left unchanged. This was repeated 25 times for each problem. By using the same random seed we guarantee that the dynamic problems are the same for all trials. We could not compare against S-DPOP as the provided implementation does not support terminating the current solving process after a period of time.

As fig 3.1 shows, SBDO always outperforms DynCOAA, however it is obvious that the solutions found by SBDO are not monotonically non-decreasing. This is because it does not have a global communication mechanism to coordinate value changes like DynCOAA does.

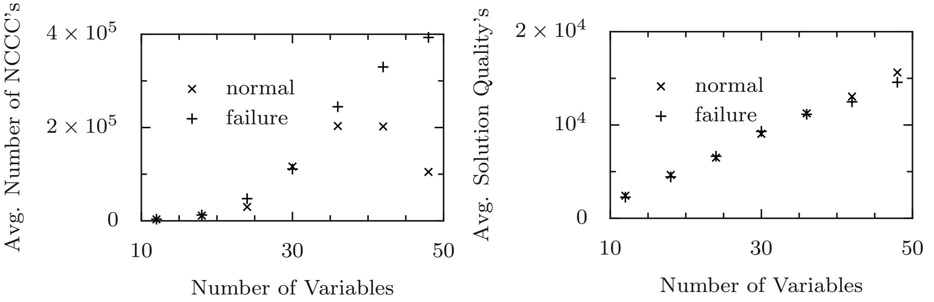

Fig. 2. Performance with unreliable agents

3.2 Fault Tolerance

To demonstrate the fault tolerance of SBDO it was run on the set of hard problems. Every 3 seconds a random agent was killed, then restarted between 1 and 3 seconds later. So at most one of the 12 to 48 agents was not operating at any time. We tried with other failure rates and got similar results. We choose to restart failed agents as our test problems are from the meeting scheduling domain. Where it is reasonable to expect that agents will be restarted when their failure is detected. As shown in figure 1 the algorithm requires more NCCCs, so therefore more time and messages to reach quiescence. Though as shown in figure 1 when it does terminate the solution is only slightly worse than when no agents fail.

3.3 Static Problems

To evaluate how SBDO performs on static problems we tested it against S-DPOP and DynCOAA. Table 1 describes the average and standard deviation for each of the metrics. Separated by easy, moderate and hard problems respectively. We were unable to measure NCCCs for DynCOAA, so they have not been reported. It also represents the average and standard deviation of the ratio of the 'utility' (or objective function value) computed over the optimal utility (represented as a percentage) for each of these algorithms. We note that SBDO generates near optimal, but not optimal, solutions in general. The SBDO algorithm performs very well, requiring slightly more messages and NCCCs, but less memory than S-DPOP. While producing slightly worse solutions that DynCOAA, but with much less time and messages.

3.4 Scalability

To evaluate how SBDO scales with different problem sizes we compared it against DPOP and DynCOAA on the moderate and hard sets of problems. Each instance was terminated after 10 minutes or if it used more than 3.5GB of memory. SBDO completed 98.6% of the moderate problems and 61.8% of the hard problems.

Table 1. Performance on static problems

Algorithm	Quality		Messages		NCCCs		Memory (MB)		Time (s)	
	avg	SD	avg	SD	avg	SD	avg	SD	avg	SD
SBDO	99.80%	0.87	70.48	25.87	643.45	451.73	0.49	0.04	0.16	0.06
DynCOAA	99.95%	2.4E-4	8715.35	2745.23	–	–	0.34	0.02	14.22	5.58
S-DPOP	100%	0.00	19.73	3.24	591.75	155.54	46.08	4.34	0.16	0.06

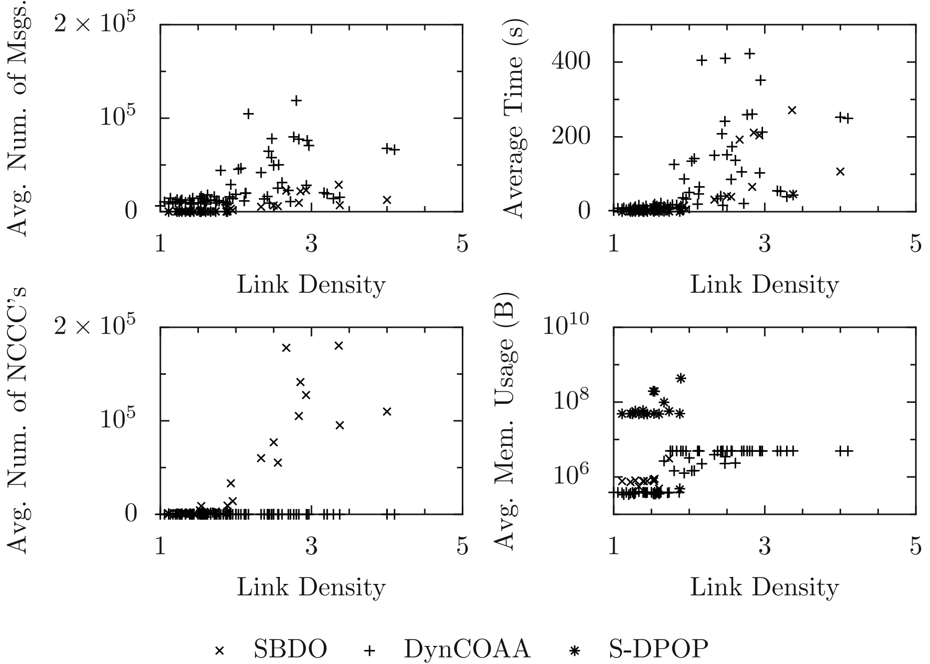

Fig. 3. Scalability of SBDO, DynCOAA and DPOP

DPOP completed all of the moderate problems and 17.5% of the hard problems. DynCOAA completed all of the moderate problems and 61.75% of the hard problems.

The plots in figure 3 have been created by averaging the data collected from all the instances the algorithms were tested on. The plots show that SBDO scales well, though it does not scale as well as DPOP on most metrics, it scales much better on memory usage.

4 Conclusion

We have presented the Support Based Distributed Optimisation algorithm that can solve Dynamic Distributed Constraint Optimisation problems 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 'isgoods', which can be viewed as arguments. An isgood contains the assignment to the variables of an agent as well as the utility of the assignment and the context in which the decision was made. Each agent can choose one of the other agents as its support and in turn uses that agent's assignment 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 few resources, as shown in table 1. Also figure 3 shows that the resources required scale well with the size of the problem.

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 1. This flexibility, coupled with the knowledge redundancy in the network makes it fault tolerant. Other agents are able to continue solving unimpeded when one or even many agents fail. Error recovery is hastened by allowing an agent that has just restarted to recreate its previous state, as shown in figure 2.

The resulting algorithm is completely asynchronous, fault tolerant, complete with respect to hard constraints but incomplete with respect to soft constraints.

In future we plan to extend the concept of objectives to allow stability constraints to be expressed. We also intend to identify how to make the algorithm complete, or at least provide theoretical guarantees on solution quality.

References

1. Chechetka, A., Sycara, K.: No-commitment branch and bound search for distributed constraint optimization. In: AAMAS 2006, pp. 1427–1429. ACM (2006)
2. Harvey, P., Chang, C.F., Ghose, A.: Support-based distributed search: a new approach for multiagent constraint processing. In: AAMAS 2006, pp. 377–383. ACM (2006)
3. Kiekintveld, C., Yin, Z., Kumar, A., Tambe, M.: Asynchronous algorithms for approximate distributed constraint optimization with quality bounds. In: AAMAS, pp. 133–140 (2010)
4. Léauté, T., Ottens, B., Szymanek, R.: FRODO 2.0: An open-source framework for distributed constraint optimization. In: Proceedings of the IJCAI 2009 Distributed Constraint Reasoning Workshop (DCR 2009), Pasadena, California, USA, pp. 160–164 (July 13, 2009), http://liawww.epfl.ch/frodo/
5. Meisels, A., Kaplansky, E., Razgon, I., Zivan, R.: Comparing performance of distributed constraints processing algorithms. In: Proceedings of DCR Workshop, AAMAS 2002 (2002)
6. Mertens, K.: An Ant-Based Approach for Solving Dynamic Constraint Optimization Problems. PhD thesis, Katholieke Universiteit Leuven (December 2006)
7. Modi, P.J., Shen, W.-M., Tambe, M., Yokoo, M.: Adopt: asynchronous distributed constraint optimization with quality guarantees. Artificial Intelligence 161, 149–180 (2005)
8. Petcu, A., Faltings, B.: S-dpop: Superstabilizing, fault-containing multiagent combinatorial optimization. In: Proceedings of the National Conference on Artificial Intelligence, AAAI-2005, pp. 449–454. AAAI, Pittsburgh (July 2005)

9. Portway, C.P.: USC dcop repository (2008), http://teamcore.usc.edu/dcop
10. Schiex, T., Verfaillie, G.: Nogood recording for static and dynamic constraint satisfaction problems. In: TAI 1993, pp. 48–55 (1993)
11. Stranders, R., Farinelli, A., Rogers, A., Jennings, N.R.: Decentralised coordination of continuously valued control parameters using the max-sum algorithm. In: AAMAS, vol. (1), pp. 601–608 (2009)
12. Vinyals, M., Pujol, M., Rodríguez-Aguilar, J.A., Cerquides, J.: Divide-and-coordinate: Dcops by agreement. In: AAMAS, pp. 149–156 (2010)