

Support-based Distributed Search

A new approach for multiagent constraint processing

Peter Harvey
Decision Systems Laboratory
University of Wollongong
NSW, Australia
pah06@uow.edu.au

Chee Fon Chang
Decision Systems Laboratory
University of Wollongong
NSW, Australia
c03@uow.edu.au

Aditya Ghose
Decision Systems Laboratory
University of Wollongong
NSW, Australia
aditya@uow.edu.au

ABSTRACT

Distributed Constraint Satisfaction Problems provide a natural mechanism for multiagent coordination and agreement. To date, algorithms for Distributed Constraint Satisfaction Problems have tended to mirror existing non-distributed global-search or local-search algorithms. Unfortunately, existing distributed global-search algorithms derive from classical backtracking search methods and require a total ordering over agents for completeness. Distributed variants of local-search algorithms (such as distributed breakout) inherit the incompleteness properties of their predecessors, or depend on the creation of new communication links between agents. This paper presents an algorithm in which a global ordering is not required, while avoiding the problems of existing local-search algorithms.

1. INTRODUCTION

Constraint Satisfaction Problems (CSPs) have proven applicable in a wide variety of domains. A CSP is classically defined by a set of variables $\mathcal{V}$, a domain for each variable $\mathcal{D}_v$, and a set of constraints $\mathcal{C}$. A *solution* to a CSP is a complete assignment of values to variables which satisfies every constraint.

A Distributed CSP (DisCSP) is formed when the description and solution procedure of a CSP are separated amongst multiple agents. The distributed environment extends the applicability of CSPs to domains such as distributed scheduling and resource contention. Of particular interest is the use of DisCSPs as models for solving other multiagent problems, with DisCSP algorithms defining a protocol for agent communication. Common examples of such problems include scheduling, task assignment, and limited forms of negotiation where simple decision(s) must be made per agent. A DisCSP can be constructed by representing each of the agents decisions as a variable, with constraints describing any inter-agent relationships.

A DisCSP can be solved by distributed variants of existing global-search or local-search algorithms. However, local-

search algorithms [7] are incomplete in both the distributed and non-distributed case. Distributed variants of global-search [1, 3, 5, 6] presented to date make use of a total order over variables.

We argue that any total order impacts the characteristics of backtracking-style search in undesirable ways for use in many multiagent problems. For example, an agent which has a 'higher' rank in the ordering has more 'authority' and is therefore less likely to change its value than a 'lower' ranking agent. In an anytime environment this results in higher-ranked agents being granted more stable answers. While some problems may desire such behaviour our concern lies with those problems which do not.

We also argue that, when using a total order, it is difficult to add constraints between two previously independent DisCSPs. To do so would require a re-computation of the variable ordering and/or an arbitrary decision that one DisCSP ranks higher than the other. If a problem is frequently altered by the addition of groups of variables, as is likely to occur in large DisCSP networks, global re-computation will become increasingly difficult. If variable ordering is instead made arbitrarily (for example, ordering by variable identifier) the problem of stability is exacerbated.

These arguments can be demonstrated in an example large-scale meeting scheduling problem described below.

Example. The universities of Pluto and Saturn each use an automated system for scheduling meetings amongst their own staff. The staff give constraints of the form 'Alice needs to meet with Bob for 2 hours this Wednesday or Thursday' to agents on their own computers. Individual universities contain a large number of staff with generally sparse connections, so a distributed algorithm is used in which agents communicate directly with each other. Agents are assigned comparable identifiers using finely-tuned schemes specific to each university. These identifiers are chosen to permit backtracking in a distributed global-search algorithm within the university.

Despite best efforts at fairness, a static ordering creates problems between research peers. If the ordering states 'Alice ranks higher than Bob' then any trivial change in Alice's meeting times must always be accepted by Bob. Inversely, Bob may request a change to Alice's meetings only after exhausting all possible meeting schedules and detecting infeasibility. This problem is distinct from that of preference orderings, and instead relates to stability (ie. Alice's schedules are more stable than Bob's).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

AAMAS'06 May 8–12 2006, Hakodate, Hokkaido, Japan.
Copyright 2006 ACM 1-59593-303-4/06/0005 ...\$5.00.

To worsen matters, Bob from Pluto wishes to arrange a meeting with Carla from Saturn. Their identifiers, while still possibly unique, are not meaningfully comparable for the purpose of a backtracking search. To continue using any existing distributed algorithm, we must be able to compare identifiers between agents operating at Pluto's and Saturn's universities. An example solution is to decide that all Saturn's identifiers are 'greater' than Pluto's identifiers. Unfortunately this would have the same impact on the behaviour of the algorithm as outlined above - meeting schedules for researchers from Pluto would become subservient to those from Saturn. Any changes in meeting times for Saturn's researchers, no matter how trivial, must be accepted by Pluto's researchers.

Furthermore, any decision for resolving the variable order would require intervention by authorities at each university or the use of a heuristic method such as DisAO [1, 3]. While these decisions can be made for pairs or sets of universities, it does not scale well computationally. For example, if Dennis was an independent researcher he must establish 'comparability' with each university and all other researchers. The addition of new researchers frequently raises the possibility of frequent re-computation of variable ordering.

The above example highlights problems that arise from using a total order to establish 'authority' or 'importance' between agents, and maintaining a total order subject to merging of previously independent DisCSPs. The computational disadvantages of a total order were also noted and addressed within the development of Asynchronous Weak-Commitment Search (AWCS) [5, 6]. However, AWCS creates additional links between variables, which we believe is undesirable in large-scale meeting scheduling. The specific difficulties of large-scale distributed meeting scheduling motivated us to develop an algorithm which:

- has no need for 'authority' between variables, effectively avoiding the need for a total order on variables.
- provides fairness in the level of stability for variables.
- does not add links between variables, avoiding the eventual need for 'broadcasting' assignments.
- addresses the risk of cyclic behaviour exhibited by local search algorithms.

Section 2 will present a model of a simple meeting scheduling problem as a DisCSP, and present how *arguments* can form the basis of communication between agents. Section 3 will describe the internal decision processes of each agent to handle arguments in an appropriate manner. Section 4 will present analysis of the algorithm.

2. MODELLING ARGUMENTS

We begin with a simple example demonstrating how a distributed constraint satisfaction problem can be solved through arguments. We will then describe a formal model of this problem with corresponding notation able to represent the dialogue. This transformation serves as inspiration for a new distributed constraint satisfaction algorithm.

Example. Alice, Bob, Carla and Dennis are attending a conference and must organise meeting times for themselves:

- Bob must meet with Alice before meeting with Carla.

- Bob must meet with Alice before meeting with Carla.
- Dennis must meet with Alice.
- Bob, Carla and Dennis must have a separate group meeting.
- Available times are 1pm, 2pm and 3pm.
- Double-booking for meeting time-slots is not allowed.
- Each person knows only those meetings that they need to attend.
- No person has any 'authority' over any other.
- Communication can only occur between people who already know each other.

To find the solution they state arguments (proposals and rejections) in turn, providing further detail if two arguments are contradictory or if they alter a previous argument:

Alice to Dennis $\Rightarrow$ I propose a 1pm meeting
Dennis to Carla $\Rightarrow$ I propose a 2pm group meeting
Dennis to Bob $\Rightarrow$ I propose a 2pm group meeting
Carla to Bob $\Rightarrow$ I propose a 1pm meeting
Alice to Bob $\Rightarrow$ I propose a 2pm meeting
Bob to Alice $\Rightarrow$ I have a group meeting at 2pm,
so I propose a 1pm meeting instead
Bob to Carla $\Rightarrow$ I completely reject your proposal,
so I propose a 3pm meeting instead
Alice to Dennis $\Rightarrow$ I now have another meeting at 1pm,
so I propose a 3pm meeting instead

2.1 Constraint Model

To construct a distributed constraint satisfaction problem we translate the time of attending a meeting for each person into a variable. Equality constraints are used to ensure meeting times are agreed to by all users. For example, a pair of variables a and b may represent the scheduled time of the meeting between Alice and Dennis. The constraint $a = b$ is interpreted as 'the time Alice decides to meet with Dennis must be the same as the time that Dennis decides to meet with Alice'. Inequality constraints ensure that meetings occur at distinct times.

Note that there is significant redundancy in the constraints and variables as presented in Figure 1. This occurs as the constraints upon one person are not automatically known to others. Relaxing this requirement would generate a simpler constraint graph, but would conflict with our aim to solve in a distributed manner. Using this constraint model as an example, we will now define suitable notation for representing the dialogue.

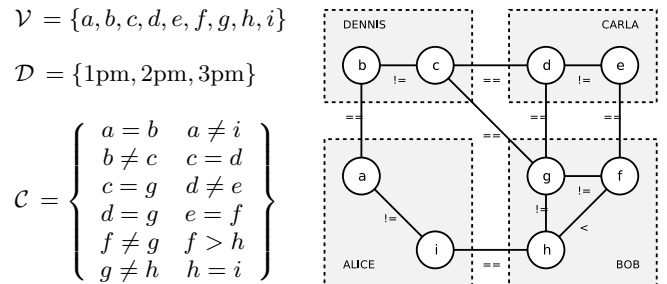


Figure 1: Example of a constraint model and graph

2.2 Argument Model

To represent proposals and rejections as outlined above, we require two distinct message types; isgoods and nogoods. These will be the only message types required to construct our algorithm.

An **isgood** is an ordered partial assignment for a sequence of connected variables, and so represents a ‘proposal’. Consider the argument in our example where Bob says to Alice: “I already have a group meeting at 2pm, so I propose a 1pm meeting for us instead”. This is a proposal, and so can be written as an ordered partial assignment or ‘isgood’:

$$\langle\langle g, 2pm \rangle, (h, 1pm)\rangle$$

This isgood is read as “variable g took on value 2pm, and so h took on value 1pm”. Note that variables in an isgood must be connected to their immediate predecessor, and therefore $\langle\langle d, 2pm \rangle, (h, 1pm)\rangle$ is *not* an isgood. Also note that we use the operator $+$ to represent the appending of a variable assignment to an isgood. For example, $\langle\langle g, 2pm \rangle, (h, 1pm)\rangle + (i, 1pm) = \langle\langle g, 2pm \rangle, (h, 1pm), (i, 1pm)\rangle$.

A **nogood** is an unordered partial assignment which is provably not part of a solution, and so represents a ‘rejection’. Consider the argument in our example where Bob says to Carla: “I reject your proposal, and I propose a 3pm meeting for us instead”. This is a rejection (he must meet Carla before Alice, so 1pm is not a possible meeting time) followed by a proposal, which written in sequence are:

$$\{(e, 1pm)\} \text{ and } \langle\langle f, 3pm \rangle\rangle$$

They are read as “variable e cannot take value 1pm” and “variable f took on value 3pm” respectively. As demonstrated in our example, a nogood is usually accompanied by an isgood.

We say that a constraint is **satisfied** by an isgood I if the constraint is explicitly not violated by the assignments in I . Testing whether a constraint is satisfied is therefore only possible if all variables appearing in the constraint also appear in I . Similarly, a nogood is satisfied if it is not a subset of the assignments in I . For example, given an isgood $I = \langle\langle g, 2pm \rangle, (h, 1pm)\rangle$, we know:

- $I + (i, 2pm)$ does not satisfy the constraint $h = i$
- $I + (i, 1pm)$ does satisfy the constraint $h = i$
- I does not satisfy the nogood $\{(h, 1pm)\}$
- I does satisfy the nogood $\{(h, 1pm), (i, 1pm)\}$

Thus, given a set of constraints and a set of nogoods, we say that an assignment (v, d) is **consistent** with respect to an isgood I iff each constraint on v and each nogood is satisfied by $I + (v, d)$.

By the notation $|I|$ we indicate the number of tuples in I . Finally, we will write $I \sqsubseteq I'$ to indicate that I is a **sub-isgood** of I' . A sub-isgood is the tail (or entirety) of another isgood. For example:

$$\begin{aligned} \langle\langle i, 1pm \rangle\rangle &\sqsubseteq \langle\langle h, 1pm \rangle, (i, 1pm)\rangle \\ &\sqsubseteq \langle\langle g, 2pm \rangle, (h, 1pm), (i, 1pm)\rangle \\ \langle\langle i, 2pm \rangle\rangle &\not\sqsubseteq \langle\langle h, 1pm \rangle, (i, 1pm)\rangle \\ \langle\langle h, 2pm \rangle\rangle &\not\sqsubseteq \langle\langle h, 1pm \rangle, (i, 1pm)\rangle \end{aligned}$$

3. SOLVING WITH ARGUMENTS

Using the above notation, and the dialogue of our example as a guide, it is possible to construct a distributed search algorithm in which agents will:

- send and receive proposals (isgoods) and rejections (nogoods)
- convince neighbours to change their value by sending progressively longer proposals
- reject a proposal from a neighbour if it is inconsistent
- justify their variable assignment by the proposal of just one neighbour
- communicate only with agents for which they share a constraint

To achieve this, each agent records the most recent proposals sent/received by neighbouring agents and an unbounded nogood store. Unlike other distributed algorithms, SBDS does not regard all information from neighbours as a consistent ‘agent view’. Instead, the isgood received from just one neighbour is chosen as justification for our current assignment and combines to form our ‘agent view’. Formally, the information stored by each agent is:

- $sent(v)$ - last isgood sent to each neighbour v
- $recv(v)$ - last isgood received from each neighbour v
- $nogoods$ - set of all nogoods ever received
- $support$ - the neighbour chosen for our ‘agent view’
- $view$ - current agent view ($recv(support)$) extended by an assignment to our own variable)

Procedure 1 *main* ()

```

1: while true do
2:   for all received nogoods  $N$  (in fifo order) do
3:     receive-nogood( $N$ )
4:   for all received isgoods  $I$  (in fifo order) do
5:     receive-isgood( $I$ )
6:     select-support()
7:   for all neighbours  $v$  do
8:     send-isgood( $v$ )
9:   wait until at least one message in the queue

```

Procedure 2 *receive-isgood* (I)

```

1: let  $v$  be the variable which sent  $I$ 
2: set  $recv(v)$  to  $I$ 
3: if no choice of value is consistent wrt  $recv(v)$  then
4:   send-nogood( $v$ )

```

Procedure 3 *receive-nogood* (N)

```

1: if  $N$  in  $nogoods$  then
2:   break, as this nogood was already known
3: add  $N$  to  $nogoods$ 
4: if no value is consistent then
5:   terminate algorithm
6: for all neighbours  $v$  do
7:   if no choice of value is consistent wrt  $recv(v)$  then
8:     send-nogood( $v$ )

```

Procedure 4 *select-support* ()

```
1: update-view ()
2: if our current value is inconsistent wrt some recv(v)
   and  $|recv(v)| \geq |view|$  then
3:   set support to a neighbour u, maximising  $|recv(u)|$ 
4:   update-view ()
```

Procedure 5 *update-view* ()

```
1: let view' be recv(support) extended by a consistent
   assignment to self, and maximal with respect to  $\prec$ 
2: let v be the first variable assigned in view'
3: if  $scope(view) \neq scope(view')$  or  $view \prec view'$  or
   the assignment of v is equal in view' and recv(v) or
   the assignment of v is unequal in view and recv(v) then
4:   set view to view'
```

Procedure 6 *send-nogood* (*v*)

```
1: let N be an inconsistent subset of recv(v)
2: send N to v
3: set recv(v) to  $\langle \rangle$ 
4: if support = v then set support to self
```

Procedure 7 *send-isgood* (*v*)

```
1: if our current value is consistent wrt recv(v) and
    $sent(v) \subseteq view$  then
2:   break, as a new isgood is not necessary
3: lock communication channel with v
4: if there are no unprocessed isgoods from v then
5:   let L be  $\min(\max(|recv(v)|, |sent(v) + I|), |view|)$ 
6:   let I be an isgood such that  $I \subseteq view$  and  $|I| = L$ 
7:   send I to v
8:   set sent(v) to I
9: unlock communication channel with v
```

The *main* loop of our algorithm processes all messages before choosing *support* and *view* and sending new isgoods. Incoming isgoods are stored in *recv*(*v*) by the *receive-isgood* procedure. If no assignment to our own variable is consistent with respect to the new isgood and current known nogoods, the procedure *send-nogood* is called to derive and send a nogood. Similarly the *receive-nogood* procedure handles an incoming nogood; each *recv*(*v*) is re-tested for consistency, and *send-nogood* is called if appropriate.

The *select-support* procedure determines which neighbouring variable will be considered as our support for this iteration. A new support must be chosen if a received isgood from a neighbour is longer than our current *view* and conflicts with our current value.

The *update-view* procedure refreshes the current *view* according to the isgood *recv*(*support*). In most cases *update-view* will replace *view* by selecting and appending a consistent assignment for our variable to the tail of *recv*(*support*). As our algorithm is asynchronous, and agents can determine their assignments simultaneously, there is the possibility of cyclic behaviour. This is corrected in existing distributed search algorithms by the use of a total order on agents; a change by a lower ranked agent cannot induce a change in a

higher ranked agent. In place of a total order on agents, we use orderings for isgoods defined over the same set of variables. These orderings are only used in a situation where a cycle is deemed likely. Being independent they can contain no bias towards any agent or value.

Formally, let $scope(I)$ be the sequence of variables in the isgood *I*. For example, with $I = \langle (c, 1pm), (b, 2pm) \rangle$ we have $scope(I) = \langle c, b \rangle$. We assume an ordering $\prec$ is known to all agents and is total for isgoods of the same *unordered* scope.¹ We will not replace a *view* with *view'* if each of the following is true:

- *view* is not out-of-date (ie. the most recently received assignment for *v* is the same as that presented in *view*)
- *view'* is out-of-date (ie. the most recently received assignment for *v* is different from that presented in *view'*)
- *view'* may be part of a cycle (ie. the ordered scopes of *view* and *view'* are identical, and the first assignment is to a neighbour *v*)
- *view'* is lower in the ordering (ie. $view' \prec view$)

This scheme causes an agent to postpone changing its value if its new view would be out-of-date, would propagate a cycle, and the old view is regarded as ‘superior’ by the ordering. As the definition of the ordering is uniform across all agents, any cyclic behaviour will quickly resolve in favour of a single view. Theorem 1 contains a formal statement and proof of this result.

The *send-nogood* procedure generates and sends an appropriate nogood when a received isgood is found to be inconsistent. The nogood may be any inconsistent subset of the assignments in the isgood. An interesting effect of this procedure is that nogoods are formed only from variables in a sequence, rather than ‘all neighbours’ as occurs in AWCS.

The *send-isgood* procedure constructs the strongest possible isgood to send to agent *v* while satisfying certain ‘minimality’ requirements. If there is no need to send an isgood (ie. we have already established agreement with our neighbour) then we need not counter its argument. An isgood is not sent to a neighbour *v* if there are unprocessed isgoods in the communication channel.

We ensure that individual pairs of agents do not communicate simultaneously by using a mutex on individual communication channels. If an argument from a neighbour *v* has arrived between the last call to *receive-isgood* and the activation of the lock then we skip sending an argument on this iteration. The result is half-duplex communication with all agents able to operate asynchronously.

To avoid cycles of oscillating agent values in inconsistent problems, we increase the length of successive arguments which are sent. This is achieved by recording the last isgood sent and attempting to increase the length of subsequent isgoods. As any cycle must be finite, eventually the arguments (isgoods) being transmitted will contain the cycle itself. If the cycle is formed from inconsistent values it will generate a nogood and break the cycle; otherwise the cycle-breaking mechanism of *update-view* will take effect.

¹Cryptographic hash functions can be used to provide a suitably unbiased ordering for sets of assignments written in a canonical form.

From	To	Argument
<i>a</i>	<i>b, i</i>	$\langle\langle a, 1\text{pm} \rangle\rangle$
<i>c</i>	<i>b, d, g</i>	$\langle\langle c, 1\text{pm} \rangle\rangle$
<i>e</i>	<i>d, f</i>	$\langle\langle e, 1\text{pm} \rangle\rangle$
<i>b</i>	<i>a</i>	$\langle\langle b, 1\text{pm} \rangle\rangle$
<i>b</i>	<i>c</i>	$\langle\langle a, 1\text{pm} \rangle, \langle b, 1\text{pm} \rangle\rangle$
<i>d</i>	<i>c, g</i>	$\langle\langle d, 1\text{pm} \rangle\rangle$
<i>d</i>	<i>e</i>	$\langle\langle c, 1\text{pm} \rangle, \langle d, 1\text{pm} \rangle\rangle$
<i>f</i>	<i>e</i>	$\{(e, 1\text{pm})\}$
<i>f</i>	<i>e, g, h</i>	$\langle\langle f, 3\text{pm} \rangle\rangle$
<i>e</i>	<i>f</i>	$\langle\langle e, 3\text{pm} \rangle\rangle$
<i>e</i>	<i>d</i>	$\langle\langle f, 3\text{pm} \rangle, \langle e, 3\text{pm} \rangle\rangle$
<i>c</i>	<i>b</i>	$\langle\langle c, 2\text{pm} \rangle\rangle$
<i>c</i>	<i>d, g</i>	$\langle\langle b, 1\text{pm} \rangle, \langle c, 2\text{pm} \rangle\rangle$
<i>g</i>	<i>c, d, f, h</i>	$\langle\langle g, 2\text{pm} \rangle\rangle$
<i>i</i>	<i>a, h</i>	$\langle\langle i, 2\text{pm} \rangle\rangle$
<i>d</i>	<i>c</i>	$\langle\langle d, 2\text{pm} \rangle\rangle$
<i>d</i>	<i>g</i>	$\langle\langle c, 2\text{pm} \rangle, \langle d, 2\text{pm} \rangle\rangle$
<i>d</i>	<i>e</i>	$\langle\langle b, 1\text{pm} \rangle, \langle c, 2\text{pm} \rangle, \langle d, 2\text{pm} \rangle\rangle$
<i>h</i>	<i>g</i>	$\langle\langle h, 1\text{pm} \rangle\rangle$
<i>h</i>	<i>i</i>	$\langle\langle g, 2\text{pm} \rangle, \langle h, 1\text{pm} \rangle\rangle$
<i>i</i>	<i>h</i>	$\langle\langle i, 1\text{pm} \rangle\rangle$
<i>i</i>	<i>a</i>	$\langle\langle h, 1\text{pm} \rangle, \langle i, 1\text{pm} \rangle\rangle$
<i>a</i>	<i>i</i>	$\langle\langle a, 3\text{pm} \rangle\rangle$
<i>a</i>	<i>b</i>	$\langle\langle i, 1\text{pm} \rangle, \langle a, 3\text{pm} \rangle\rangle$
<i>b</i>	<i>a</i>	$\langle\langle b, 3\text{pm} \rangle\rangle$
<i>b</i>	<i>c</i>	$\langle\langle a, 3\text{pm} \rangle, \langle b, 3\text{pm} \rangle\rangle$
<i>c</i>	<i>d, g</i>	$\langle\langle b, 3\text{pm} \rangle, \langle c, 2\text{pm} \rangle\rangle$

Solved: $a, b, e, f = 3\text{pm}$; $c, d, g = 2\text{pm}$; $h, i = 1\text{pm}$

In the first iteration, *a*, *c* and *e* announce their initial values (all chose 1pm). Note that other agents could communicate at the same time (eg. *g* and *h*) but to simplify the explanation we have limited the presentation of arguments.

In the second iteration, *b*, *d* and *f* respond to these proposals. In the first case, *b* accepts the proposed time of 1pm from *a* and communicates an agreeable value to *a*. As *c* proposed a contradictory value, *b* provides a longer isgood as a counter-proposal. At the same time, *d* accepts the proposed time of 1pm from *c* and communicates that to its neighbours. Finally, *f* rejects the proposal from *e* outright with a nogood, and proposes instead a time of 3pm.

In the third iteration, *e* and *c* respond to the counter-proposals. Given the longer proposal from *f*, *e* must change its current value. When updating its neighbour *d*, *e* must provide a longer argument. Similarly, *c* provides longer proposals to *d* and *g*.

In the fourth iteration *g* and *i* announce their values for the first time, choosing values supported by *c* and *a* respectively.

In the fifth iteration, *d* accepts the proposed time of 2pm from *c*. Note the increasing (and varying) lengths of isgoods as *d* communicates with different neighbours. Similarly, *h* accepts the proposed time of 2pm from *g* and chooses its first assignment accordingly.

In the sixth iteration, *i* changes its *support* from *a* to *h*, and changes its value accordingly. This change is then communicated to *a*. Note that this is the first communication from *i* to *a*.

In the seventh iteration, *a* is forced to choose between *b* and *i* as *support*. As *a* had no previous support it is free to choose either (if it previously had used *b* as *support* it would have retained it). In this instance, *a* chooses to use *i* as *support* and change value accordingly.

The final iterations consist of propagating this change to *b* (which changes its value), and then to *c*. Note that *c* does not change value, but still must communicate the change of *b*'s value to *d* and *g*.

Table 1: Example algorithm execution on the problem in Figure 1

4. RESULTS

We have described desirable properties of an algorithm for distributed constraint satisfaction. Specifically, we have attempted to construct an algorithm which:

- has no need for ‘authority’ between variables, effectively avoiding the need for a total order on variables.
- provides fairness in the level of stability for variables.
- does not add links between variables, avoiding the eventual need for ‘broadcasting’ assignments.
- addresses the risk of cyclic behaviour exhibited by local search algorithms.

The fact that we do not add links between variables is evident from the algorithm itself. Similarly, we note the absence of any total ordering over the variables, which avoids any notion of ‘authority’. Each isgood establishes an order over variables within a local context in the form of a sequence in which assignments were made. This is necessary for the introduction of nogoods in the style of Dynamic Backtracking [2, 1]. However, the combination of these local orders does not necessarily end in the construction of a total order over variables. We have also provided a novel method

to address cyclic behaviour which plagues distributed local search algorithms [7]. Each of these results can be observed in the detailed algorithm execution shown in Table 1.

Key to the success of SBDS is the elimination of cyclic behaviour in what is otherwise a local search algorithm. Below we will present formal proof that cyclic behaviour has been eliminated:

LEMMA 1. *Eventually no new nogoods will be generated, and the length of view will become stable (bounded above) for each agent.*

PROOF. Each agent maintains an unbounded store of nogoods, and each isgood is constructed in a way that is consistent with all nogoods known at the time. Therefore, if the algorithm were to never terminate, eventually no new nogoods will be generated.

If no new nogoods are generated, then the method of selection for *support* and *view* ensures that *support* will stabilise. As a result, the length of *view* will also stabilise as it is monotonic increasing over time, but bounded above by either the depth of the ‘support tree’ or the length of a cycle. $\square$

THEOREM 1. *The algorithm is sound, and will terminate.*

PROOF. The algorithm uses sound nogood derivation techniques and will terminate with ‘no solution’ only if the empty nogood is derived. Inversely, each agent ensures that its neighbours know its current value, and will continue to communicate if an inconsistency exists. The algorithm will not terminate unless it has a correct answer, and therefore is sound.

By Lemma 1 we know that eventually no new nogoods will be generated and the length of *view* will become stable for each agent. Therefore the *support* for each agent will also become stable, and so value selection for each variable will become dependant only upon information from its *support*. In such a situation the algorithm will only fail to terminate if there exists some directed tour of agents $v_1, \dots, v_n$ which are ‘supporting’ each other and oscillating between candidate solutions. However, each candidate solution has the same unordered scope, and so we can utilise the postponement scheme outlined in Section 3. By this scheme, solutions ranked lower by $\prec$ will be removed until the oscillating stops and the algorithm terminates. $\square$

Finally, we provide empirical evidence that no variables change value significantly more often than any other. Figure 2 presents the number of value changes per variable while using our algorithm to solve a randomly constructed feasible problem. Observe that the frequency of value change is fairly evenly distributed amongst variables, with none forced to change significantly more than another. Identical results are seen for other problem sizes and constraint tightness. This particular result also provides further evidence that no significant ‘authority’ is exerted by one variable over another. If authority was exerted we would expect an exponential (or at least polynomial) increasing curve, as some variables are forced to change by others.

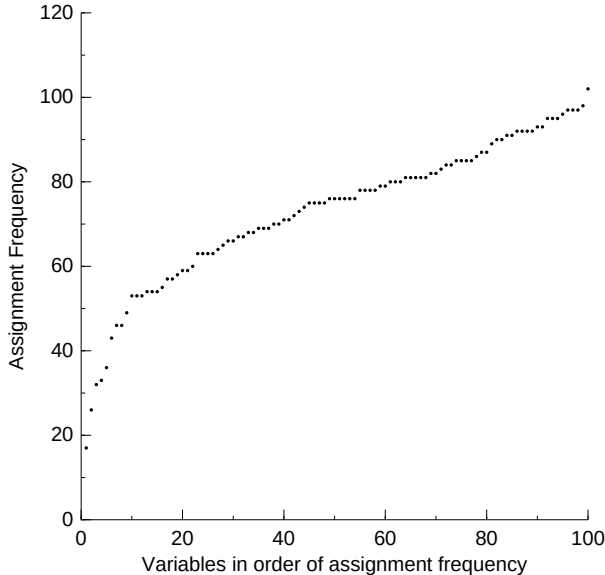


Figure 2: Frequency of assignment changes per variable for a random problem of 100 variables, 300 constraints, domain size of 5, and constraint tightness of 0.325

5. PERFORMANCE

In previous work [4] we used a complicated definition of is-good involving a measure of ‘strength’ that allowed SBDS to approximate chronological backtracking search. The aim of that measure of strength was to minimise the rate of growth of L in *send-isgood*, and so reduce the amount of information being transmitted in each iteration. However, this measure significantly complicated the proof of completeness. To minimise the rate of growth of L , and to increase performance, we replace this measure with a simple heuristic carefully constructed to maintain completeness.

First note that the completeness of SBDS depends upon lines 1 of *update-view* and line 5 of *send-isgood*. We risk cyclic behaviour if we do not choose a *view'* which is maximal with respect to $\prec$ or forcibly increase the length sent to neighbours. However, these items incur significant performance penalties. For example, by choosing a *view'* that is maximal with respect to $\prec$, we are unable to use min-conflict or similar value selection heuristics. Similarly, by consistently increasing the length of isgoods, we discourage local-search behaviour and impact performance accordingly.

However, we are not required to select the maximal *view'* or increase the length of isgoods with *every* iteration of the algorithm. We may instead perform either action after a finite number of iterations, by:

- forcing an increase in $|sent(v)|$ only after a bounded number of iterations of the algorithm;
- permitting heuristic selection of *view'* until $|sent(v)|$ cannot be increased further;
- permitting a decrease in $|sent(v)|$ when the ordered scope of *view* changes.

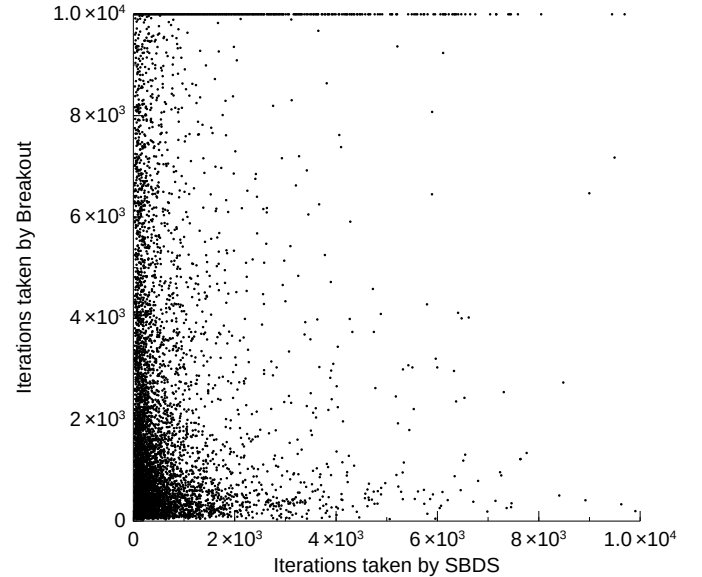


Figure 3: Comparison of SBDS and Distributed Breakout for feasible random problems with 200 variables and 400 constraints, domain size of 5, and constraint tightness of 0.4

These simple techniques significantly improve the performance of SBDS by permitting the use of min-conflict heuristics in value selection. Figures 3 and 4 present comparisons of SBDS and Distributed Breakout. We compare with Distributed Breakout as it is simple to implement, and is often very fast on feasible problem instances. We ran each on approximately 13000 feasible random binary problem instances, with an upper limit of 10000 iterations per instance. We also ran SBDS on approximately 350 infeasible random problem instances generated in the same way.

Figure 3 plots each random problem instance found feasible by SBDS, comparing the number of iterations taken for each algorithm. For the majority of instances where the algorithms completed, both took less than 2000 iterations. However, Distributed Breakout still had a large proportion of instances which took more than 2000 iterations, and a significant proportion were not completed. Particularly important is that there were few problems found ‘easy’ for Distributed Breakout but hard for SBDS.

Figure 4 graphs the percentage of completed problems for Distributed Breakout and SBDS. As Distributed Breakout is unable to solve infeasible problem instances we have not presented those. As can be seen, of the problem instances found feasible by SBDS, 98% were solved within a 4000-iteration limit. In contrast Distributed Breakout was able to complete only 50% of the same problems within 4000 iterations, and 55% within 10000 iterations. We also present the percentage of problem instances found to be infeasible by SBDS, and the iterations required to find them. While performance is worse on infeasible problems, it is still reasonable.

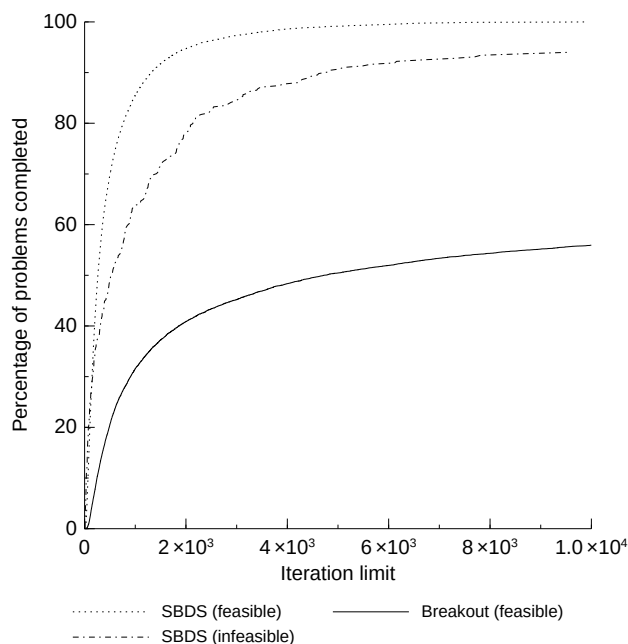


Figure 4: Comparison of SBDS on feasible and infeasible random problems with 200 variables and 400 constraints, domain size of 5, and constraint tightness of 0.4.

6. CONCLUSION

Distributed Constraint Satisfaction Problems provide a natural mechanism for multiagent coordination and agreement. To date, algorithms for Distributed Constraint Satisfaction Problems have tended to mirror existing non-distributed global-search or local-search algorithms. However, there exist natural examples of DisCSPs for which a total variable ordering and/or linking variables is not desired. If we are to solve such DisCSPs we must develop new algorithms designed specifically for distributed environments.

In this paper we have presented one such algorithm. Key to the success of the algorithm is the use of argumentation as a model for agent operation. This technique avoids fixed ranks for agents and the resultant behaviour which is undesirable in natural problems such as meeting scheduling. The placing of a total order over solutions for subsets of variables also provides a novel approach to solving the issue of cyclic behaviour in local search algorithms. This paper also represents a significant simplification, with formal proof results and better performance, of the algorithm presented in [4].

Note that SBDS has been developed with specific goals in mind; it is designed to solve problems for which existing algorithms are unsuitable. It is therefore difficult to compare performance characteristics with other algorithms, in much the same way as comparisons between centralised and distributed algorithms are difficult. The above performance results are promising, but further testing against other distributed search algorithms is required.

7. REFERENCES

- [1] C. Bessière, A. Maestre, and P. Meseguer. Distributed dynamic backtracking. In T. Walsh, editor, *CP*, volume 2239 of *Lecture Notes in Computer Science*, page 772. Springer, 2001.
- [2] M. L. Ginsberg. Dynamic backtracking. *J. Artif. Intell. Res. (JAIR)*, 1:25–46, 1993.
- [3] Y. Hamadi. Interleaved backtracking in distributed constraint networks. *International Journal on Artificial Intelligence Tools*, 11(2):167–188, 2002.
- [4] P. Harvey, C. F. Chang, and A. Ghose. Practical application of support-based distributed search. In *ICTAI*. IEEE Computer Society, 2005.
- [5] M. Yokoo. Asynchronous weak-commitment search for solving distributed constraint satisfaction problems. In U. Montanari and F. Rossi, editors, *CP*, volume 976 of *Lecture Notes in Computer Science*, pages 88–102. Springer, 1995.
- [6] M. Yokoo and K. Hirayama. Algorithms for distributed constraint satisfaction: A review. *Autonomous Agents and Multi-Agent Systems*, 3(2):185–207, 2000.
- [7] W. Zhang and L. Wittenburg. Distributed breakout revisited. In *AAAI/IAAI*, pages 352–, 2002.