Support-based Distributed Search

Peter Harvey, Chee Fon Chang, and Aditya Ghose

Decision Systems Laboratory
University of Wollongong
NSW, Australia

Abstract. 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 variables for completeness. Distributed variants of local-search algorithms (such as distributed breakout) inherit the incompleteness properties of their predecessors.
We will provide an example of a naturally-occurring DisCSP where a global ordering is difficult to maintain and creates undesirable behaviours. We will present an algorithm which blends local and global search in such a way that a global ordering is not required. It demonstrates a new direction for DisCSP research, avoiding any notion of 'authority' between variables while appearing to achieve completeness.

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 other domains such as distributed scheduling and resource contention. In the usual case 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 many classes of problems. For example, an agent which has a 'higher' rank in the ordering has more 'authority' and is less likely to change value than a 'lower' ranking agent. Thus higher-ranked agents are granted more stable answers for anytime queries.

We also argue that it is difficult to add constraints between two previously independent DisCSPs when using a total order. To do so would require a re-computation of the variable ordering and/or an arbitrary decision that the first

DisCSP ranks higher than the second DisCSP. If a problem is frequently altered by 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 seen in a practical problem, described below.

Example 1. The universities of Pluto and Saturn each use an automated system for scheduling meetings amongst their own staff. Staff input constraints of the form 'Alice needs to meet with Bob for 2 hours this Wednesday or Thursday' into their own computers. Individual universities contain a large number of staff with generally sparse connections. Therefore a distributed algorithm is used in which computers communicate directly with each other. Computers 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. For example, 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 (Alice's schedules are more stable than Bob's).

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

We have highlighted problems that arise from using a total order to establish 'authority' or 'importance' between variables, and maintaining a total order subject to merging of previously independent DisCSPs. These difficulties motivate 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.

Section 2 will present *arguments* which form the basis of communication in our algorithm. The use of arguments is inspired by work within the agents and logic communities [4]. They allows us to avoid a central authority as agents must provide convincing reasons for assignment changes. Section 3 will present the algorithm and describe the internal decision processes of each agent. Section 4 will present analysis and worked examples.

2 Using Arguments

We begin by outlining the distributed setting of our algorithm. Agents have control over the assignment of value to their own variable and are connected to neighbours by binary constraints. Communication with neighbouring agents is only by way of *arguments*. Mutually acceptable assignments are established by agreeing to or countering arguments, where any counter-argument must be 'stronger' (according to a measure described later) than the argument which it is countering. To this end, arguments received from neighbours can be used as *support* to defeat the arguments of other neighbours. Behaviour approximating this description can be seen in human negotiations, where arguments, counter-arguments, and support relations are common.

In our algorithm arguments are either:

– a provably infeasible assignment to a set of variables, or;
– a locally feasible assignment to a sequence of variables (where each variable is connected by a constraint to its immediate predecessor).

The first is known within constraint satisfaction literature as a *nogood*. This is equivalent to 'flat refusal', and is always (and only) generated if an agent finds that there is no possible value it can take while certain other assignments are in effect. The only option when a nogood has been received by an agent is to change its current assignment, or to generate a nogood for another agent.

The second type of argument we term an *isgood*. This is equivalent to 'a proposal', and is generated by an agent in an attempt to convince other agents to take upon a feasible assignment. A neighbour will accept an isgood if it does not conflict with their current assignment, or is 'strong' enough to convince them to change. A neighbour will reject an isgood if it is able to counter with a nogood or a stronger isgood.

At this point we can provide a simplified overview of the algorithm, as executed by each agent:

1. Receive and store new isgoods and nogoods from neighbours.
2. If a received isgood is found inconsistent (due to our constraints and learned nogoods) respond with a nogood and delete the isgood.
3. Choose a strong isgood (received from a neighbour) to use as our support.
4. Extend the chosen isgood to build a 'rationale' describing a choice of value.
5. Send new isgoods to neighbours (as appropriate) derived from the rationale.
6. Goto 1.

2.1 Building isgoods

An isgood is written as a non-empty sequence of triples (v, d, n), where $v \in \mathcal{V}$ is a variable, $d \in \mathcal{D}_v$ is the value assigned to that variable, and $n \in \mathbb{Z}^+$ is the number of values in $\mathcal{D}_v$ which were eliminated by preceding assignments to other variables. Each variable is represented at most once in an isgood, and the assignments described must satisfy all constraints defined over those variables.

Example 2. Take the problem of $x \neq y$ and $x = z$, where $\mathcal{D}_x = \mathcal{D}_y = \mathcal{D}_z = \{a, b, c\}$. A possible assignment of $x = a$ and $y = b$ (in that order) would be represented as an isgood $\langle (x, a, 0), (y, b, 1) \rangle$ which indicates that:

- x took value a, and neither of the alternatives (b or c) had been eliminated.
- y took value b, and the alternative a had already been eliminated (by the preceding assignment of a to x).

Similarly, a possible assignment of $x = a$ and $z = a$ (in that order) would be represented as an isgood $\langle (x, a, 0), (z, a, 2) \rangle$, as all alternative values for z were eliminated by the preceding assignment of x. Finally, the assignment $y = a$, $x = b$ and $z = b$ is represented as $\langle (y, a, 0), (x, b, 1), (z, b, 2) \rangle$.

A sub-isgood is any connected subsequence of an isgood that includes the tail, with elimination counts adjusted accordingly. For example, $\langle (z, b, 0) \rangle$ is a sub-isgood of $\langle (x, b, 0), (z, b, 2) \rangle$, and both are a sub-isgood of $\langle (y, a, 0), (x, b, 1), (z, b, 2) \rangle$. An isgood is also a sub-isgood of itself.

Note that any isgood must be ordered in such a way that there must exist a constraint between each assigned variable and its immediate predecessor. Within the given example, $\langle (x, b, 0), (y, a, 1), (z, b, 2) \rangle$ is not an isgood as there is no constraint between y and z.

2.2 Rating isgoods

It is often the case that an agent receives isgoods from multiple neighbours and is unable to choose a value consistent with all of them. In such a case the agent must choose one neighbour as 'support', and a value consistent with their most recent isgood. An agent builds upon its support by appending a value assignment, and is able to use this new isgood (called the *rationale*) to communicate with conflicting neighbours. The support should be selected to provide the strongest rationale, and so we require a scheme to allow all agents to determine the strength of an isgood.

We can define a function *strength* which calculates the strength of an isgood in a recursive fashion. We will use $\langle I, (v, d, n) \rangle$ to denote the isgood formed by taking an existing isgood I and appending the tuple (v, d, n).

$$\begin{aligned} \mathit{strength}(\langle\,\rangle) &= 0 \\ \mathit{strength}(\langle I, (v, d, n) \rangle) &= \mathit{strength}(I) \times |\mathcal{D}_v| + n + 1 \end{aligned}$$

This function has been chosen as it simulates the increasing strength of deduction demonstrated in backtracking search and ensures that increasing the size of an isgood increases its strength. This property guarantees that an agent is always able to select a support to successfully build on and use to defeat isgoods presented by neighbouring agents.

2.3 Building nogoods

A nogood is written as a set of pairs (v, d) where $v \in \mathcal{V}$ is a variable and $d \in \mathcal{D}_v$. For example, the nogood $\{(x, a), (y, b)\}$ indicates that the simultaneous assignment of $x = a$ and $y = b$ is inconsistent. A nogood may only be used to counter an isgood and must only contain a subset of those assignments described in the isgood. As a result, nogoods only involve variables which are connected by a chain; this has interesting implications for independence of subproblems.

Example 3. Take the problem of $x < z$, $x < y$ and $y \neq z$, where $\mathcal{D}_x = \mathcal{D}_y = \mathcal{D}_z = \{a, b, c\}$ and $a < b < c$. The assignment $x = b$ and $z = c$ would be written as an isgood $\langle (x, b, 0), (z, c, 2) \rangle$. However this does not permit an assignment to y. Thus y may send a response nogood to z (the last assigned variable) of $\{(x, b), (z, c)\}$.

In this case z would discover that it cannot take a value compatible with its own constraints, the assignment to x, and the new nogood. In turn, it would generate a nogood $\{(x, b)\}$ which is sent to x.

The introduction of a nogood to an agent may impact upon the number of assumptions recorded for a particular assignment. In the above example the nogood $\{(x, b)\}$ was introduced, in which case $\langle (x, a, 1) \rangle$ is then a possible isgood. An informal example demonstrating the construction and use of nogoods is presented below.

From	**To**	**Argument**	
x	y, z	$\langle (x, b, 0) \rangle$	x proposes an assignment
z	y, x	$\langle (z, c, 0) \rangle$	z proposes an assignment compatible with x
y	x	$\langle (y, c, 0) \rangle$	y proposes an assignment compatible with x
y	z	$\langle (x, b, 0), (y, c, 2) \rangle$	y counter-proposes with a stronger isgood
z	y	$\{(x, b), (y, c)\}$	z rejects the the proposal from y
y	x	$\{(x, b)\}$	y rejects the original proposal from x
y	x, z	$\langle (y, b, 0) \rangle$	y proposes an assignment compatible with z
x	y, z	$\langle (x, a, 1) \rangle$	x proposes a new assignment
algorithm stops with $x = a, y = b, z = c$			

Note that the above example involved variables which are fully connected, making it impossible to demonstrate some aspects of our algorithm. For instance, a nogood is *not* counted as a constraint when determining possible sequencing of variables in an isgood. Also, the above example does not demonstrate the asynchronous aspects of our algorithm.

3 Procedure

The algorithm we have devised for constraint satisfaction consists of six sub-procedures and a main loop (not counting procedures for constraint tests, etc).

process-isgood(I)	handle received isgood I
process-nogood(N)	handle received nogood N
select-support()	choose a neighbour who will be our support
update-rationale()	update the rationale after a change in support
compute-nogood(v)	compute and send a nogood to neighbour v
compute-isgood(v)	compute and send an isgood to neighbour v

Each procedure also assumes the availability of certain data structures:

isgood(v)	last isgood received from neighbour v	initially $\langle\,\rangle$
last(v)	last isgood sent to neighbour v	initially $\langle\,\rangle$
nogoods(v)	set of nogoods applicable to *isgood*(v)	initially $\{\,\}$
rationale	current isgood including our assignment	initially $\langle\,\rangle$
support	current neighbour regarded as our support	initially *self*

Given an isgood I we also define utility functions:

scope(I)	the sequence of variables in the order presented in I
var(I)	the set of variables in I
ass(I)	the set of assignments in I

For $I = \langle (y, a, 0), (x, b, 1), (z, b, 2)\rangle$ we have $scope(I) = \langle y, x, z\rangle$, $var(I) = \{x, y, z\}$, and $ass(I) = \{(x, b), (y, a), (z, b)\}$.

We make use of constants *self* and $\mathcal{D}$ to refer to our own variable and domain. Correspondingly, *isgood*(*self*) is always the empty isgood $\langle\,\rangle$ as we never send an isgood to ourselves. The set *nogoods*(*self*) are those nogoods which mention only our own assignment and not those of neighbours.

We will present procedures for processing messages and maintaining structures first, then the procedures for sending messages. This corresponds approximately to the *main* loop of our algorithm (below) which processes all messages before choosing *support* and *rationale* and then sending new isgoods.

Procedure 1 *main* ()

```
1: while true do
2:    for each received nogood N (in fifo order) do process-nogood(N)
3:    for each received isgood I (in fifo order) do process-isgood(I)
4:    select-support()
5:    for each neighbour v do compute-isgood(v)
6:    wait until there is at least one message in the queue
7: end while
```

Procedure 2 *process-isgood* (I)

```
1: let v be the source of I
2: set isgood(v) := I
3: if there is no assignment consistent with isgood(v) and nogoods(v) then
4:    compute-nogood(v)
5: else
6:    remove from nogoods(v) those not applicable to isgood(v)
7: end if
```

The *process-isgood* procedure updates the value of $isgood(v)$ appropriately. If no assignment to our own variable is consistent with the new isgood (and current known nogoods) the procedure *compute-nogood* is called to generate and send a nogood. Otherwise, any newly-obsolete nogoods are removed and the new isgood is retained.

The *process-nogood* procedure handles the storage of an incoming nogood. The nogood is first tested against $isgood(self)$, the empty isgood, to check for non-contingent domain reductions. In the event that our domain is permanently wiped out we terminate the algorithm, with messages sent to neighbours instructing them to also terminate.

The nogood is then tested for 'applicability' with the isgood of each neighbour variable, and if found to be applicable is stored in the corresponding collection of nogoods. A nogood is 'applicable' to an isgood if the isgood assigns the same values to the same variables as found in the nogood (excluding *self*).

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

We define the function $strength_{self}(I) = strength(I) \times |\mathcal{D}| + e + 1$, where e is the number of values in $\mathcal{D}$ eliminated when subject to the assignments described

Procedure 3 *process-nogood* (N)

```
 1: if N is applicable to isgood(self) then
 2:    add N to nogoods(self)
 3:    if no value is consistent with nogoods(self) then terminate algorithm
 4: end if
 5: for all neighbours v do
 6:    if N is applicable to isgood(v) then
 7:       add N to nogoods(v)
 8:       if there is no assignment consistent with isgood(v) and nogoods(v) then
 9:          compute-nogood(v)
10:       end if
11:    end if
12: end for
```

Procedure 4 *select-support* ()

1: *update-rationale* ()
2: **if** for some neighbour v, $isgood(v)$ conflicts with our value in *rationale* and $strength(isgood(v)) \geq strength(rationale)$ **then**
3: **set** *support* to maximise $strength_{self}(isgood(support))$
4: *update-rationale* ()
5: **end if**

Procedure 5 *update-rationale* ()

1: **let** $r := isgood(support)$ extended by an assignment to self, maximising $hash(r)$
2: **set** $weak := (scope(rationale) = scope(r)$ **and** $hash(rationale) > hash(r))$
3: **set** $rationale := r$

in I. Using this function we can determine which neighbour to choose as our support to provide the strongest rationale.

We call upon *update-rationale* twice within *select-support*. Once to ensure the rationale is correct if $isgood(support)$ has changed before testing if a new support should be chosen, and once if the value of *support* has changed. In each case *update-rationale* updates the value of *rationale* by appending an assignment to our variable to the tail of $isgood(support)$.

In the situation where $isgood(support)$ has changed, but has not strengthened nor changed in scope, the algorithm may develop cyclic behaviour of the form demonstrated in Dynamic Backtracking. We address this problem by way of a global *hash* function to map isgoods to a totally ordered set. Given two isgoods I and J where $var(I) = var(J)$ we define *hash* such that $ass(I) = ass(J)$ iff $hash(I) = hash(J)$. By using this definition we provide a total order for isgoods defined over the same set of variables. In general the *hash* function should be constructed to have the chaotic properties of a cryptographic hash function.

We choose the rationale to maximise the value of the *hash* function. We mark the new rationale as *weak* if it has the same scope but worsens the value of *hash*. This information is used by the *compute-isgood* procedure which will refuse to use a weak rationale if it would perpetuate cyclic behaviour. As the *hash* function must have a maximum feasible rationale for any set of variables, and all variables use the same *hash*, any cycle will resolve in favour of the maximal rationale.

The *compute-nogood* procedure generates and sends an appropriate nogood when a domain wipeout is discovered. The nogood is formed from the values of all neighbouring variables listed in the isgood. An interesting effect of this

Procedure 6 *compute-nogood* (v)

1: **let** $N :=$ {a minimal inconsistent subset of $ass(isgood(v))$}
2: **send** N to v
3: **set** $isgood(v) := \langle\,\rangle$
4: **if** $support = v$ **then** $support := self$

Procedure 7 *compute-isgood* (v)

1: **if** *weak* = true **and** v is the first variable in *rationale* **and**
   isgood$(v) = \langle\,\rangle$ or the value of v is different in *rationale* and *isgood*(v) **then**
2: **break**, to avoid creating a cycle
3: **end if**
4: **if** *last*(v) contains only assignments that are also in *rationale* **and** *last*$(v) \neq \langle\,\rangle$
    and our own value in *rationale* is consistent with *isgood*(v) and *nogoods*(v) **then**
5: **break**, as a new isgood is not necessary
6: **end if**
7: **lock** communication channel with v
8: **if** there are no unprocessed isgoods from v **then**
9: **construct** an isgood I, maximising *strength*(I) while satisfying the following:
      I is a sub-isgood of *rationale* **and**
      I does not contain an assignment to v **and**
      for every J which is a strict sub-isgood of I either
        strength$(J) \leq$ *strength*(*last*(v)) **or**
        strength$(J) <$ *strength*(*isgood*(v))
10: **send** I to v
11: **set** *last*$(v) := I$
12: **end if**
13: **unlock** communication channel with v

procedure is that all nogoods are formed from variables in a sequence. This places an additional restriction on the formation of nogoods which has no impact on the completeness of the algorithm.

A wide range of heuristics can be used to generate a 'minimal' nogood. Our experiments have shown that a nogood that consists mostly of recent assignments (those closest to the current assignment) leads to better performance.

The *compute-isgood* procedure constructs the weakest isgood possible to send to agent v while still satisfying certain requirements. It is called after all isgoods and nogoods have been processed and all new nogoods have been sent. If there is no need to send an isgood (we have already established agreement with our neighbour) then we need not counter its argument.

We ensure that individual pairs of agents do not communicate simultaneously by making using of a mutex on individual communication channels. If an argument from a neighbour v has arrived between the last call to *process-isgood* and the activation of the lock 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 must try to increase the strength of isgoods which are sent. This is achieved by recording the last isgood sent and attempting to increase the strength of subsequent isgoods. Increasing argument strengths can also be seen in humans where any change in decision must be backed by a rationale stronger than that given previously. As any cycle must be finite, eventually the arguments being transmitted

will contain the cycle itself and (if formed from inconsistent values) will generate a nogood, breaking the cycle.

As an isgood must not contain the description of the assignment to the target variable v, it is possible for the isgood to be cut short. This unfortunately allows for cyclic behaviour for *consistent* problems as observed in Distributed Breakout [7]. To prevent this behaviour we do not send an isgood to the first variable in the rationale if the rationale has been marked as *weak* and contradicts the current value of that variable. We can conclude that the rationale will eventually become redundant and should not be propagated in a cycle.

Note that in actual implementation, the elimination counts stored within an isgood are not as described in this paper. We use a data structure equivalent in size and information content, with the added feature that the construction of an isgood (line 9 of *compute-isgood*) and the evaluation of *strength* become trivial. We have omitted the details of this structure in favour of a more intuitive explanation of isgoods.

4 Results

An implementation of this algorithm was run against a variety of planar graph problems, n-queens problems, and random binary problems. Performance of this algorithm (measured in CPU time and number of messages) was fair, but not competitive with other distributed CSP algorithms. In part we attribute this to the time expended by the algorithm in the absence of a total order on variables.

We have described specific behavioural aspects that we wish to achieve:

- no total ordering over variables is required for backtracking.
- fairness in stability for variable assignments.
- completeness of the algorithm on all problems.

4.1 Variable ordering and backtracking

Each isgood establishes an order over variables within a local context in the form of a sequence in which assignments were made. For the purposes of nogoods and value selection an agent treats its support as a higher-ranked agent, though it is free to change support at any time. This is necessary for the introduction of nogoods in the style of Dynamic Backtracking [2, 1]. We argue that the combination of these local orders does not necessarily end in the construction of a total order over variables.

Figure 1 shows a constraint graph of a 100-variable problem with random binary constraints and domain size of 8. Figures 2 - 4 display directed graphs of the support relation (ie. which variables treated others as support) observed at algorithm termination for constraint tightness values of 0.3, 0.4 and 0.5.

We have observed that the formation of support connections will be avoided if:

- constraints are weak (Fig. 2), as conflicts can be avoided; or

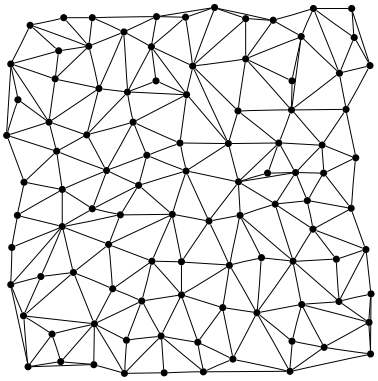

Fig. 1. 100-variable constraint graph

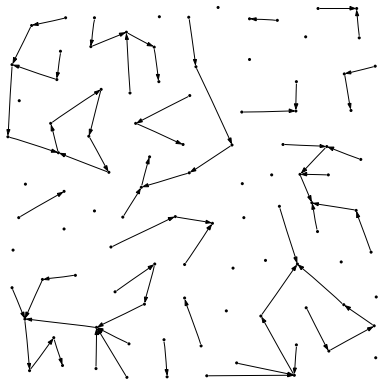

Fig. 2. Support relation at termination, with tightness 0.3

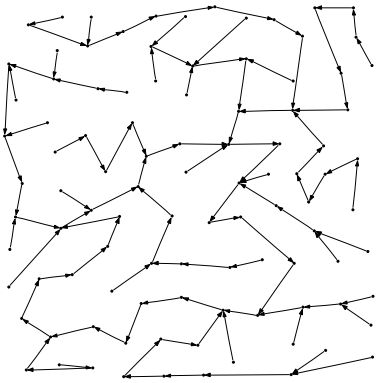

Fig. 3. Support relation at termination, with tightness 0.4

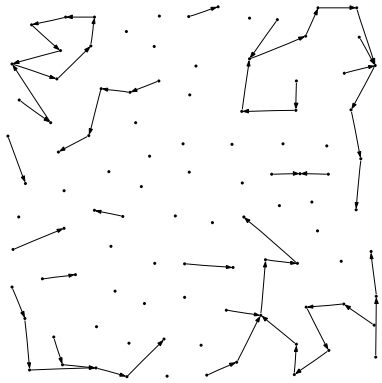

Fig. 4. Support relation at termination, with tightness 0.5

– constraints are very strong (Fig. 4), as variables have little need for the support of neighbours in developing a rationale.

In other cases a global 'authority' (one variable at the root of a large portion of support relation) may still not arise. Observe that in Fig. 3 no variable provides a root for the support relation.

Returning to our introductory motivating example, this is an important result. Backtracking via nogoods may only occur across support relations. It implies that the scheduling of meetings within Pluto and Saturn universities may remain independent, despite constraints between them. Backtracking search (generation of nogoods) initiated within Pluto's agents will not necessarily propagate to (and disrupt) Saturn's agents as support relations may not exist. We have verified this behaviour by constructing problems formed of two weakly connected CSPs and observing the development of the support relations over time.

4.2 Stability of assignments

Algorithms which use fixed variable orderings have a tendency for higher-ranked variables to change value less often. Conversely, lower-ranked variables change value frequently. Within classical backtracking this is an obvious behaviour, and its presence continues in distributed variants. We have provided reasons why such behaviours is undesirable.

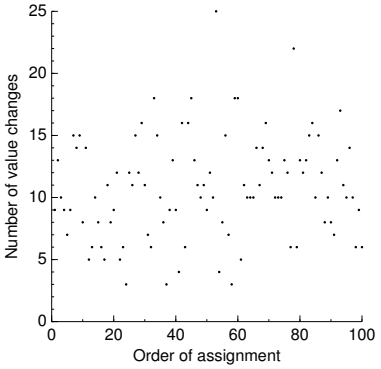

Fig. 5. Counts of value changes, with constraint tightness of 0.4

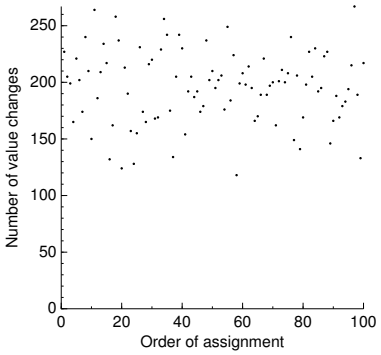

Fig. 6. Counts of value changes, with constraint tightness of 0.5

Figures 5 and 6 present the number of value changes per variable. We have sorted the variables according to the order in which they were first assigned. We see that the order in which the variables chose their initial assignment was irrelevant to the number of times their value changed. The frequency of value change is also evenly distributed.

4.3 Completeness

We are able to provide arguments for the completeness of our algorithm, though no formal proof has been established. First note that all steps of the algorithm are correct. Nogoods contain all relevant information, and are disregarded if no longer applicable. Isgoods are always consistent with the constraints across those variables. We now present an explanation and demonstration of our technique for avoiding cyclic behaviour and a corresponding argument for termination.

Distributed Breakout avoids the requirement of a global ordering and allows variables to increase the importance of their constraints. The ability for agents to increase their 'insistence' on a variable change provided inspiration for the generation of arguments. Unfortunately Distributed Breakout suffers from incompleteness [7] for many satisfiable CSPs. This is due to variables which are in conflict being able to simultaneously and independently change their value, retaining their conflicting status and leading to cyclic behaviour.

By providing arguments with progressively increasing scope we have attempted to place a bound on the maximum level of 'insistence' a variable may provide. The use of increasing scope also provides us with a method to detect cycles of variables. It is tempting to provide additional connections between variables, providing synchronous behaviour for agents to avoid simultaneous change.

However, with the introduction of the *hash* function we have provided a total order (per set of variables) for assignments. When an agent observes successive rationales without change of scope the total order provides a common understanding of the 'better' rationale. Thus, without explicit synchronisation while executing, agents agree on a common assignment and prevent cyclic behaviour.

Take the CSP with $w \neq x, x = y, y \neq z, z = w$ and $\mathcal{D}_w = \mathcal{D}_x = \mathcal{D}_y = \mathcal{D}_z = \{T, F\}$. Also presume that *hash* rates $\{(w,T),(x,F),(y,F),(z,T)\}$ higher than $\{(w,F),(x,T),(y,T),(z,F)\}$. A possible execution of our algorithm demonstrating cycle-breaking behaviour is presented below. For the sake of clarity we have not included all notifications of value changes.

From	**To**	**Argument**	
x	w, y	$\langle(x,T,0)\rangle$	x proposes to w and y
z	w, y	$\langle(z,T,0)\rangle$	z proposes to w and y
w	z	$\langle(x,T,0),(w,F,1)\rangle$	w counter-proposes to z
y	x	$\langle(z,T,0),(y,F,1)\rangle$	y counter-proposes to x
x	w	$\langle(y,F,0),(x,F,1)\rangle$	x changes proposal to w
z	y	$\langle(w,F,0),(z,F,1)\rangle$	z changes proposal to y
w	z	$\langle(y,F,0),(x,F,1),(w,T,1)\rangle$	w changes proposal to z
y	x	$\langle(w,F,0),(z,F,1),(y,T,1)\rangle$	y changes proposal to x
x	w	$\langle(z,F,0),(y,T,1),(x,T,1)\rangle$	x changes proposal to w
z	y	$\langle(x,F,0),(w,T,1),(z,T,1)\rangle$	z changes proposal to y
w	z	$\langle(y,T,0),(x,T,1),(w,F,1)\rangle$	w changes proposal to z
y	x	$\langle(w,T,0),(z,T,1),(y,F,1)\rangle$	y changes proposal to x
x	w	$\langle(z,T,0),(y,F,1),(x,F,1)\rangle$	x changes proposal to w
z			z waits, rationale is weak
w	z	$\langle(y,F,0),(x,F,1),(w,T,1)\rangle$	w changes proposal to z

algorithm stops with $w = T,\ x = F,\ y = F,\ z = T$

Observe that z waited and did not propose $\langle(x,T,0),(w,F,1),(z,F,1)\rangle$ to y. This is due to the use of the *hash* function to detect weak rationales, and the observation that y currently had a value contradicting the proposal from w to z.

The algorithm will also detect unsatisfiable CSPs (due to the use of nogoods). Take the CSP with $w \neq x, x = y, y = z, z = w$ and $\mathcal{D}_w = \mathcal{D}_x = \mathcal{D}_y = \mathcal{D}_z = \{T, F\}$. A possible execution of our algorithm is presented below.

From	To	Argument	
x	w, y	$\langle (x, T, 0) \rangle$	x proposes to w and y
z	w, y	$\langle (z, T, 0) \rangle$	z proposes to w and y
w	z	$\langle (x, T, 0), (w, F, 1) \rangle$	w counter-proposes to z
z	y	$\langle (w, F, 0), (z, F, 1) \rangle$	z changes proposal to y
z	y	$\langle (z, F, 0) \rangle$	z changes proposal to w
y	x	$\langle (z, F, 0), (y, F, 1) \rangle$	y counter-proposes to x
x	w	$\langle (y, F, 0), (x, F, 1) \rangle$	x changes proposal to w
x	y	$\langle (x, F, 0) \rangle$	x changes proposal to y
w	z	$\langle (y, F, 0), (x, F, 1), (w, T, 1) \rangle$	w changes proposal to z
z	w	$\{(y, F), (w, T)\}$	z rejects proposal from w
z	w, y	$\langle (z, F, 0) \rangle$	z changes proposals to w and y
w	x	$\{(y, F)\}$	w rejects proposal from x
w	x, z	$\langle (w, F, 0) \rangle$	w proposes to x and z
x	y	$\{(y, F)\}$	x rejects proposal from y
x	y	$\langle (w, F, 0), (x, T, 1) \rangle$	x proposes to y
y	z	$\{(z, F)\}$	y rejects proposal from z
y	z	$\langle (y, T, 1) \rangle$	y proposes to z

eventual domain wipeout...

We will now argue that the strength of rationales is monotonic increasing (when conflicts between agents are encountered). As the strength of finite isgoods is bounded above, and we presume that only finite isgoods are constructed, this must lead to eventual termination or 'idling' of all agents.

1. An agent will only accept an isgood which conflicts with its current rationale if the strength of that isgood exceeds the strength of the rationale. If the rationale is weaker, the agent is guaranteed to establish a new, stronger rationale. If the rationale is stronger, the agent is able to respond with a stronger isgood and potentially provide a stronger rationale for neighbours.
2. An agent must always accept a nogood and this is likely to impact upon the rationale. If the nogood does not cause a domain wipeout the rationale will be strengthened by virtue of our chosen definition of *strength*. If a domain wipeout occurs a nogood will be issued to the current support, and our rationale is potentially weakened. In the absence of cycles of isgoods this 'nogood forwarding' behaviour will terminate with the rationale of an agent being increased. This strengthened rationale may then be propagated along the path the nogood used, providing new strengthened rationales *if required due to conflicts*.

From these observations we argue that the strength of rationales held by agents is, in a general sense, monotonic increasing. In the case where a rationale is not increased this is due to agreement amongst agents which, in turn, provides termination. We hope to establish a formal proof of the above arguments in future research.

5 Conclusion

A distributed environment extends the applicability of CSPs but provides new challenges to the development of algorithms. In the usual case a DisCSP can be solved by distributed variants of existing global-search or local-search algorithms. However, there exist natural examples of DisCSPs for which a total variable ordering is not desirable. If we are to solve such DisCSPs we must develop new algorithms designed specifically for distributed environments.

The algorithm described in this paper represents a realisation of new concepts for addressing these concerns. Key to this is the use of 'arguments', which provide:

- a more robust method of escalating an agent's 'insistence' on value selection, by incrementally increasing the strength of arguments.
- recording of a sequence of agent's assignments which led to a value selection, which provides opportunities for nogood construction.
- the placing of a total order over solutions for subsets of variables, resolving the cyclic behaviour exhibited by Distributed Breakout.
- avoiding fixed ranks for agents and the resultant 'subservient' behaviour which is undesirable in many instances.

Primarily this work is a proof-of-concept, providing a basis for the development of more efficient algorithms for solving DisCSPs without variable ordering. We have demonstrated our algorithm has desirable behavioural properties, and contains novel mechanisms for avoiding cyclic behaviour. Future work will focus on performance characteristics and formal completeness results. One promising performance improvement is the use of deltas when communicating with neighbours, as the majority of communication involves only minor changes to communicated isgoods.

References

1. Christian Bessière, Arnold Maestre, and Pedro Meseguer. Distributed dynamic backtracking. In Toby Walsh, editor, *CP*, volume 2239 of *Lecture Notes in Computer Science*, page 772. Springer, 2001.
2. Matthew L. Ginsberg. Dynamic backtracking. *J. Artif. Intell. Res. (JAIR)*, 1:25–46, 1993.
3. Youssef Hamadi. Interleaved backtracking in distributed constraint networks. *International Journal on Artificial Intelligence Tools*, 11(2):167–188, 2002.
4. Gerard Vreeswijk. Abstract argumentation systems. *Artif. Intell.*, 90(1-2):225–279, 1997.
5. Makoto Yokoo. Asynchronous weak-commitment search for solving distributed constraint satisfaction problems. In Ugo Montanari and Francesca Rossi, editors, *CP*, volume 976 of *Lecture Notes in Computer Science*, pages 88–102. Springer, 1995.
6. Makoto Yokoo and Katsutoshi Hirayama. Algorithms for distributed constraint satisfaction: A review. *Autonomous Agents and Multi-Agent Systems*, 3(2):185–207, 2000.
7. Weixiong Zhang and Lars Wittenburg. Distributed breakout revisited. In *AAAI/IAAI*, pages 352–, 2002.