Delegation in Tree-search for Distributed Constraint Satisfaction

Muhammed Basharu[1]*, Ken Brown[1], and Youssef Hamadi[2]

[1] Cork Constraint Computation Centre, University College Cork, Ireland.
mb@4c.ucc.ie, k.brown@cs.ucc.ie
[2] Microsoft Research, 7 J J Thomson Avenue, Cambridge, United Kingdom.
youssefh@microsoft.com

Abstract. We introduce the idea of delegation in distributed tree-search, as a method to reduce the communication overhead when solving Distributed Constraint Satisfaction Problems (DisCSPs). With delegation, an agent can eliminate some direct forward links to child neighbours and choose intermediaries for communicating with such children. We present an algorithm which constructs long delegation paths automatically, and we prove that given certain assumptions it does not decrease privacy. We show experimentally that delegation can reduce messages by 50% for hard problems, although at the expense of more constraint checks.

1 Introduction

Distributed Constraint Satisfaction Problems (DisCSP) [10] are a generalisation of CSPs for tackling decision problems where the processing power and autonomy are naturally distributed - for example, meeting scheduling or sensor networks. Agents maintain local CSPs, which are linked through inter-agent constraints. DisCSPs are generally solved by distributed tree-based search, where a partial order of the agents is used to record the progress of the exploration. In most of these algorithms, agents send local solutions to their children (the set of neighbours lower than them in the ordering). Children in turn solve their local problems to be consistent with the incoming partial solutions. When an agent cannot find a local solution, a distributed backtracking step is started and addressed to a subset of the agent's parents. Within this broad framework, many different approaches are possible, balancing the issues of total run-time, network transmission costs, fair use of resources, and maintenance of agent privacy.

The main decision is whether the search should be synchronous or asynchronous. Synchronised search closely resembles standard non-distributed search processes. Using the tree-ordering, agents pass control up and down the tree, and each agent only operates when it has control. Typically, an agent receives a partial solution for all its ancestor agents, computes its own local extension, and passes the new partial solution onto its children. Backtracking is synchronised similarly. In asynchronous search, all agents

* This work is supported by grants from Microsoft Research, Science Foundation Ireland, and the Embark Initiative of the Irish Research Council of Science Engineering and Technology.

may operate simultaneously, computing their own local solution based on whatever current knowledge they have of the other agents' decisions, and updating those solutions when that knowledge is updated. Asynchronous search tends to have a smaller total runtime, since much computation is done in parallel and dead-ends can be identified early, but at the expense of more network traffic, and possibly redundant chains of computation. Synchronous search reduces the network traffic, but typically has a longer runtime. In addition, privacy can be compromised, since larger partial solutions are passed up and down the tree. The consensus view is that if message passing is relatively more expensive than computation, and privacy is not important, then synchronised search is better; on the other hand, if runtime is important, or privacy is important, then an asynchronous search is better.

Here, we consider the case where message passing is slow, unreliable or expensive, but where privacy is also important. We present a concept called delegation, where some agents may decide to transmit their local solutions through intermediate agents. Specifically, an agent may appoint one of its neighbours to relay messages to a second neighbour. This can be viewed as an implicit form of local synchronisation, although each agent is still free to act asynchronously, and indeed backtracking messages continue to be asynchronous. The intuition is that the second neighbour should receive larger and more coherent partial solutions from the intermediary, and thus should invoke fewer redundant chains of decisions, at the expense of a small delay in receiving the original message. We will show a simple delegation strategy which preserves the privacy level of existing algorithms. We also show that the delegation strategy can reduce the number of messages by approximately 50% for hard problems, but similarly increases the number of constraint checks, and thus is effective in scenarios where the cost of each message is high.

In the following, we start with an overview of the DisCSP formalism. Section 3 defines delegation, and in Section 4, we present an algorithm for performing delegation in advance of a search where links between unconnected agents are added prior to a search. In Section 5, we consider algorithms where new links are created as a search progresses and present a technique for performing delegation for such algorithms. Both Sections 4 and 5 also include results of evaluations of the respective delegation strategies.

2 Background

A DisCSP is a 4-tuple (X,D,C,A) where:

1. X is a set of n variables $X_1, X_2, \ldots, X_n$.
2. D is a set of domains $D_1, D_2, \ldots, D_n$ of possible values for the variables X_1, X_2, ..., X_n respectively.
3. C is a set of constraints on the values of the variables. The constraint C_k (X_{k1}, ..., X_{kj}) is a predicate defined on the Cartesian product $D_{k1} \times \ldots \times D_{kj}$. A constraint is satisfied if the value assignment of these variables satisfies the predicate.
4. A $= \{A_1, A_2, \ldots, A_p\}$is a partition of X among p autonomous processes or agents where each agent A_k "owns" a subset of the variables in X with respect to some mapping function $f : \mathsf{X} \rightarrow \mathsf{A}, s.t. f(X_i) = A_j$.

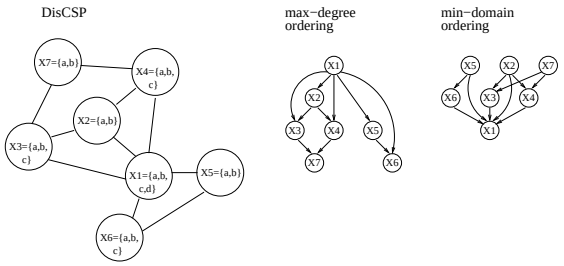

Fig. 1. A DisCSP and two agent orderings.

A solution to a DisCSP is, as for standard CSPs, an assignment to each variable of value from its domain, such that all constraints are satisfied.

In the solving process, we assume that each agent controls its own variables, and, as a default, knows only its own domains and the constraints defined on its variables. The agents must cooperate to find a global solution through message passing. A basic method for finding a global solution uses the distributed backtracking paradigm [8,3]. The agents are prioritized into a partial order $<_o$ such that any two agents are connected if there is at least one constraint between them. The ordering is determined by user-defined heuristics and classical CSP heuristics can be used as presented in Figure 1. Solution synthesis uses the partial ordering to perform an exhaustive search with backtracking. An agent instantiates its local problem w.r.t. higher priority agents and sends its local solution to lower priority neighbours, while backtracking messages are passed back up the ordering. This process computes a global solution by distributed aggregation of local solutions.

3 Delegation in DisCSP

Consider the situation shown in Figure 2. A_i has to share its partial solution with at least two connected children, A_j and A_k. On receiving this solution, both A_j and A_k make choices of their own, and transmit those to their chosen neighbours, invoking further search. Suppose A_k then receives A_j's choice, and discovers it is incompatible with its own choice. It must then find a new consistent choice and transmit that to its neighbours, overriding the previous message. This will invoke a new search, whose messages may take some time to catch up and override the previous one, and thus two searches, each requiring messages and computation, are spreading across the network at the same time, even though one of them is redundant. Alternatively, A_k may not be able to find a consistent value, and so must transmit a backtrack message to A_j; meanwhile, the previous redundant search continues in the rest of the network without being cancelled.

The question we ask here is whether a more selective procedure for transmitting partial solutions can improve efficiency by reducing the number and size of redundant searches. In particular, we consider whether an agent should reduce the number of forward messages it sends, by delegating some children to relay the messages to

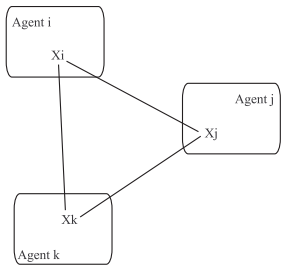

Fig. 2. A DisCSP agent 3-clique

other children. In our example, A_i might choose to delegate through A_j, and so A_k would receive messages from A_j only, where each message contains consistent value assignments for A_i and A_j. Initially, this reduces the number of messages (from 3 to 2 between the agents shown), and may stop a possibly redundant search initiated before A_j's value is transmitted. However, (i) the details of the search algorithm can interact in different ways to cause different searches, and (ii) the inherent unpredictability of message timings can have significant effects on the efficiency of an algorithm.

We define delegation in DisCSPs as follows:

Definition 1 *A DisCSP P=(X,D,C,A) can implement delegation iff,* $\exists A_i,\ A_j,\ A_k,\ X_i,\ X_j,\ X_k$ *s.t.* $f(X_i) = A_i, f(X_j) = A_j, f(X_k) = A_k$ *and* $\exists C_{ij}, C_{ik}, C_{jk} \in C$. A_i *can delegate its messages for* A_k *via* A_j*, denoted as* $(A_i, (A_j, A_k))$*, s.t.* A_i *does not send any messages directly to* A_k*, and* A_j *relays* A_i*'s decisions to* A_k *instead.*

Note that the definition ensures that the agents form a *3-clique*, and hence delegation does not add any extra links in the graph. Furthermore, delegation preserves the privacy of existing algorithms since although partial solutions are collated within agents, no agent will receive any values it would not have received without delegation.

3.1 Delegation vs. Synchronisation

The collation and transmission of partial solutions in delegation appears to be similar to synchronised search [2,9,11]. However, there are significant differences. In the following, we highlight some key features of synchronous search and we use these to show how it differs from delegation and why asynchrony is still retained with delegation.

1. **Privilege passing:** At the core of synchronous distributed search is the concept of *privilege passing* [2,11], where agents are in turn given a privilege to extend a partial solution or to revise earlier decisions as a search progresses. Generally, in synchronous search one agent is active at a time while all the other agents remain in a wait state, although in some versions [2] a DFS-tree ordering allows agents in unconnected branches of a search tree to be active simultaneously. Privilege passing ensures that agents have up-to-date information on the state of a search and as such minimises useless processing. In delegation there is no concept of privilege

passing. Agents still retain their autonomy and they carry out an asynchronous search where each agent is triggered into action whenever it receives messages irrespective of the state of its ancestors or successors. Therefore, an agent may be active simultaneously with other agents that are constrained with it. However, there is clearly some form of local synchronisation in delegation, since some agents only receive a parent's decision after some intermediary has processed it.

2. **Backtracking:** because of privilege passing, the processing of backtrack messages tends to be in reverse order of the search, and an agent must wait until all its children complete their actions before processing a backtrack from one of them. Using delegation, an agent responds immediately to a backtrack message, and can initiate new searches as a result, so there can be multiple searches proceeding simultaneously in the same sub-tree. The basic principle of delegation also makes no commitment to what should happen when a dead-end is discovered - depending on the details of the algorithm, an agent may decline to forward infeasible partial solutions, or may forward some sub-solution, allowing child agents to continue with a search or learn nogoods.
3. **Privacy:** In the synchronous algorithms of [9,11], the current partial solution for all ancestors is passed from one agent to the next, and thus agents will receive values for variables to which they are not connected. Using delegation, an agent should only receive values that it would also have received in the original algorithm. An additional consequence of this is that message packets should be smaller.

The idea of deputing agents was also explored in the Asynchronous Partial Overlay (APO) algorithm [5]. APO involves a resolution process that requires conflicting agents to centralise information about related parts of a problem within a mediator to resolve conflicts. There are significant privacy implications from mediation as agents have to reveal complete information about their domains and constraints violations for mediation to take place. In contrast, delegation requires agents simply to detect cliques, select intermediaries, and to route only the information that intermediaries are expected to see through them. Previous research on the performance trade-offs between synchronous and asynchronous backtracking have shown that message passing is reduced with synchronisation, but these savings come with the cost of an increase in run time (e.g. in [9]). However, later results reported in [11] suggest that synchronous algorithms may perform equally as well as asynchronous algorithms in runtime although idle time is much higher in synchronous search. Other results reported in [1] also show that the inclusion of some partial synchronisation improves efficiency of asynchronous backtracking - improving both the message count and the runtime.

4 Performing static delegation

We first consider delegation inside IDIBT/CBJ [4]. In the preprocessing phase of IDIBT /CBJ, agents are ordered with the Distributed Agent Ordering (DisAO) algorithm [4], part of which involves an extension of DisCSP graphs with the addition of tautological constraints between unconnected agents along different sub-trees. The extensions ensure the correctness of backtracking steps. The algorithm and its proof of complete-

ness have been described in [4]. In this section, we show how to plan delegation after ordering but before search, and evaluate its effect experimentally.

4.1 Establishing Delegation Paths

Algorithm 1 is presented for establishing the delegation paths below an agent A. This algorithm is independent of the tree search and it is run after agents construct an ordering with DisAO. Therefore each agent knows its parents, its children, and their positions in the ordering. The local data structures can be interpreted as follows: d[i] states whether A talks directly to c_i; l[i] is the length of the delegation path to c_i; m[i][j] indicates whether c_i is a parent of c_j; r[i][j] states whether c_i will relay messages to c_j; and f[i][j] states whether A must forward p_i's messages to c_j.

First, an agent must detect all ordered 3-cliques involving itself and two children. Each agent sends the full list of its children to each of its parents (line 2); the receiving agent can then populate its parenthood matrix m (3-7). The agent then processes each child in order of priority; if the child has no intermediate parents, it remains directly connected; otherwise, the intermediate parent that is furthest away from the agent (10) is selected to relay messages (11), the child's path length is updated (12), and it is marked as no longer directly connected (13). Once all delegations have been selected, each child is told to whom it must relay A's messages (15). Finally, when an agent receives those messages from its parents, it records the relay instructions (18). This algorithm selects the longest path for each delegation by chaining together overlapping 3-cliques. We aim for long delegation paths in order to remove many forward links, and so that the final messages aggregate as many local solutions as possible.

Figure 3(a) presents an example of this algorithm in use. We assume the 5 agents have been ordered using DisAO with the max-deg heuristic, as shown in Figure 3(b). First, X_5 will send an empty list to both X_4 and X_1, X_4 sends $\{X_5\}$ to X_3 and X_1, and X_3 sends $\{X_2, X_4\}$ to X_1. On receipt of these messages, X_1 keeps X_3 on a direct link, and then decides to delegate X_3 to relay messages to X_4. Similarly, X_1 eliminates the forward links with X_2, selecting X_3 as the intermediary, and eliminates the forward link to X_5, with X_4 as the intermediary. Figure 3(c) shows the DisCSP with the active forward links (solid links) after delegation paths have been established. Note that only 1 out of 4 links from X_1 is active. Therefore, during a tree search X_1 will only have to communicate with X_3 whenever it revises its value, but it knows the updates will reach all its children. Note that messages from X_1 to X_5 are relayed twice: X_3 will relay X_1's decision to X_4, and X_4 knows that if it receives a decision forX_1, it must extract it and relay it to X_5.

Algorithm 2 describes the process of relaying the appropriate decisions during search. A message to an agent from a parent will contain a decision for that parent, and may contain decisions for other parents as well. The agent first makes its own decision (line 1). The agent initialises a message for each of its children c_j with active links and places its value in the message if it has one (3). And then for each other parent decision (5), if there is another child c_t for which A uses c_j as an intermediary and to which A is expected to forward the other parent's decision (7), then A adds that decision to c_j's message (8). If however, the agent has no value, it holds on to the values for

Algorithm 1: choosing delegation paths for agent A

```
/* n children c_1, ... c_n, m parents p_1, ... p_m */
Data: d: array of n booleans, initially true
Data: l: array of n ints, initially 1
Data: m: array of n×n booleans, initially false
Data: r: array of n×n booleans, initially false
Data: f: array of m×n booleans, initially true
1  foreach parent p_i do
2      send message to p_i containing {c_1, ..., c_n}
3  foreach child c_i do
4      receive message from c_i containing child set C_i
5      for x ∈ C_i do
6          if x is a child of A (x = c_j) then
7              m[i][j] ← 1
8  foreach child c_j in order do
9      if ∃ k s.t. m[k][j] = 1 then
10         find i with highest l[i] s.t. m[i][j] = 1
11         r[i][j] ← true
12         l[j] ← l[i]+1
13         d[j] ← false
14 foreach child c_i do
15     send message to c_i with {c_j : r[i][j] = true}
16 foreach message with S_i received from a parent p_i do
17     foreach c_j ∈ S_i do
18         f[i][j] ← 1
```

each parent's decision that contributes to the domain wipe out (i.e. its conflict set) and it forwards other parents' decisions (10).

4.2 Theoretical properties

Here we proof that delegation paths created locally with local information are valid, showing that: (1) there is always a path of active forward links between each agent and each of its children, and (2) privacy is preserved in the paths generated. In particular, we will show that all intermediaries chosen to forward messages to child neighbours have active forward links with those children. We also give some other formal properties of Algorithm 1. For these results, we assume that the agent ordering algorithm, DisAO, produces a single highest priority agent, and recursively adds links to unconnected agents that share a child.

Theorem 1 *All agents in a delegation path from A_i to A_n are children of A_i*

Proof. From Algorithm 1, A_i only reasons and constructs paths with its children.

Algorithm 2: Agent A reacting to decision from p_i

Input: $M_i = \{(p_k, x_k) : p_k \text{ is parent of A}\}$

```
1  choose A's decision x_A
2  foreach c_j s.t. d[j] = true do
3      if x_A ≠ null then
4          message[j] ← {(A, x_A)}
5      foreach (k, x_k) ∈ M_i do
6          if x_A ≠ null then
7              if f[k][t] = true then
8                  message[j] ← message[j] ∪{(k, x_k)}
9          else
10             if (f[k][t] = true) ∧(k ∉ conflictSet_A) then
11                 message[j] ← message[j] ∪{(k, x_k)}
12     send message[j] to c_j
```

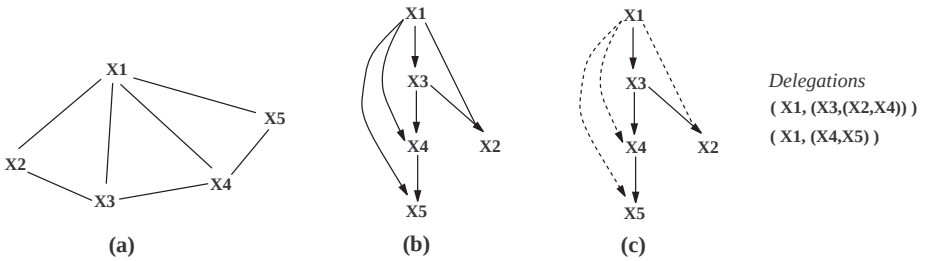

Fig. 3. An example DisCSP (a), a max-deg ordering established with DisAO (b), and active forward links (solid lines) after delegation (c).

Theorem 2 *The path length assigned to an agent A_j during the algorithm for agent A_i is the maximum path length from A_i to A_j using the sub-graph defined byA_i and A_i's children.*

Proof. (Omitted) by induction on the assigned path length

Corollary 3 *The delegation path from A_i to A_n is a maximum length path from A_i to A_n in the sub-graph defined by A_i and A_i's children.*

Proof. proof straight from Theorem 2.

Theorem 4 *For any 3-clique (A_i, A_j, A_n), where $A_i \rightarrow A_j, A_i \rightarrow A_n, A_j \rightarrow A_n$ and there is a delegation $(A_i, (A_j, A_n))$, A_j will send its own decisions directly to A_n.*

Proof. To prove this, we will show that there can be no chain of agents $C_1, C_2, \ldots, c_t$ between A_j and A_n in the ordering, such that A_j delegates its message for A_n through

this chain. The extension phase of DisAO is a recursive process that add directed links (tautological constraints) between pairs of unrelated agents if they have at least one child in common. If there was such a chain, the procedure would have added links $A_i \rightarrow C_k$ for each C_k, working backwards from the child A_n. But by Corollary 4.3, the chain C_1 to C_t must be the longest path between A_j and A_n. Substituting this chain for the direct link $A_j \rightarrow A_n$ would then give a longer path in A_i's delegation chain. But by Corollary 4.3, A_i has chosen the longest path. Contradiction. Thus there can be no such intermediate chain, and so A_j must send its decision directly to A_n.

Theorem 5 *An agent A_j can receive a decision $A_i \leftarrow v$ by delegation if and only if it can receive it without delegation, given the ordering induced by DisAO (and so we don't violate the privacy of any message).*

Proof. (Omitted) by inspection of the algorithm

Theorem 6 *No agent A_j receives the same decision $A_i \leftarrow v$ from two separate agents.*

Proof. (Omitted) by inspection of the algorithm

4.3 Algorithm modifications

To implement delegation in IDIBT/CBJ, following modifications were made to the algorithm:

- Agents construct delegation structures after an ordering has been established with DisAO by running the processes in Algorithm 1. Once the delegations have been selected, each child is told to whom it must relay its parents messages.
- A search proceeds as normal with IDIBT/CBJ, except that each agent will only send ***InfoVal*** messages to those child neighbours with whom it has an active forward link. And when an agent receives values from a parent neighbour, for whom it acts as an intermediary, it relays that value to the respective child neighbour after it has processed the message and selected its own value (see Algorithm 2).
- All backward links remain active. Even if there is an intermediary between agents A_i and A_k, A_k will bypass the intermediary and send ***Back*** messages directly to agent A_i.
- For the sake of simplicity, all search in IDIBT/CBJ is executed in a single search context (i.e. assuming NC=1).

In Section 3.1, we highlighted backtracking as one of key differences of delegation and synchronous search. In synchronous search, when the agent holding the current partial solution backtracks it returns the privilege (and by extension the partial solution) back to a culprit variable. In delegation, however, there are a number of options for dealing with partial solutions by dead end agents. In our preliminary evaluations, we considered the following:

1. Backtracking agents still pass down collated partial solutions to child neighbours. The case for doing this is that while a search below the dead end agents continues with an incorrect search context, it gives opportunities for child neighbours to quickly detect other conflicts with subsets of the partial solutions that remain coherent after the resolution of the original conflict.

2. An agent could hold back its conflict set i.e. the subset of parents' values that cause a domain wipe out, and relay other values. Again, this allows the search to retain its asynchrony as well as allow child neighbours to quickly detect conflicts with the other ancestors in the solutions they receive.
3. There is the option of allowing agents temporarily hold up the search beneath them and not relay any parent decisions to child neighbours whenever they can not extend the partial solutions. With this, child neighbours will eventually receive more coherent solutions but this approach comes with a risk of these pauses cascading up a search tree and gradually introducing additional synchronisation into agents activations.

While each of the choices has its merits, we chose the option of holding back conflict sets in the implementation of delegation in IDIBT/CBJ. This allows us to preserve more asynchrony in a search (compared to holding all collated values) while reducing the amount of redundant work would be performed if agents still forward down all collated values when they perform backtracking.

4.4 Evaluations

IDIBT/CBJ with delegation was implemented in a discrete event simulation environment using a shared simulated clock for all agents. In the simulation, we assume that the time taken to perform each constraint check is equivalent to one simulated time step. And, we also assume that message passing delay is uniform for all agents.

The modified algorithm was tested on random DisCSPs $\langle n = 30, d = 10\rangle$ on problems with different constraint densities ($p_1 = \{0.3, 0.5\}$) and different percentages of forbidden tuples (p_2) in the constraints for each density. 30 problems where generated for each combination of constraint density and tightness. For comparison, similar runs were made on the same problems with IDIBT/CBJ and a synchronous backjumping algorithm (SCBJ) [11]. SCBJ is a distributed equivalent of a centralised backjumping algorithm, where a total ordering [3] is imposed on agents and agents are activated in turn (one at a time) to extend a partial solution. A partial solution is passed from one agent to the next during a search. When the partial solution can not be extended, the earliest assignments in the partial solution that contribute to a domain wipe-out are resolved into a conflict set and used to determine (and activate) the culprit agent in backjumping.

In the charts plotted in Figure 4, we summarise the results from the experiments performed. The results show the average message count and the average Non-Concurrent Constraint Checks (NCCC) [6] from the runs. The average message count plotted in Figure 4 comprises the cost of running DisAO, messages exchanged in the course of the different searches, and where applicable, the messages exchanged in performing delegation with Algorithm 1. The results presented show with SCBJ, the average message count is lower than the asynchronous algorithms but its average NCCC is higher. With delegation in place, message count in the asynchronous search is reduced significantly especially on the most difficult problems.

[3] To keep comparisons fair, we use the same *max-deg* agent ordering from DisAO in SCBJ as well.

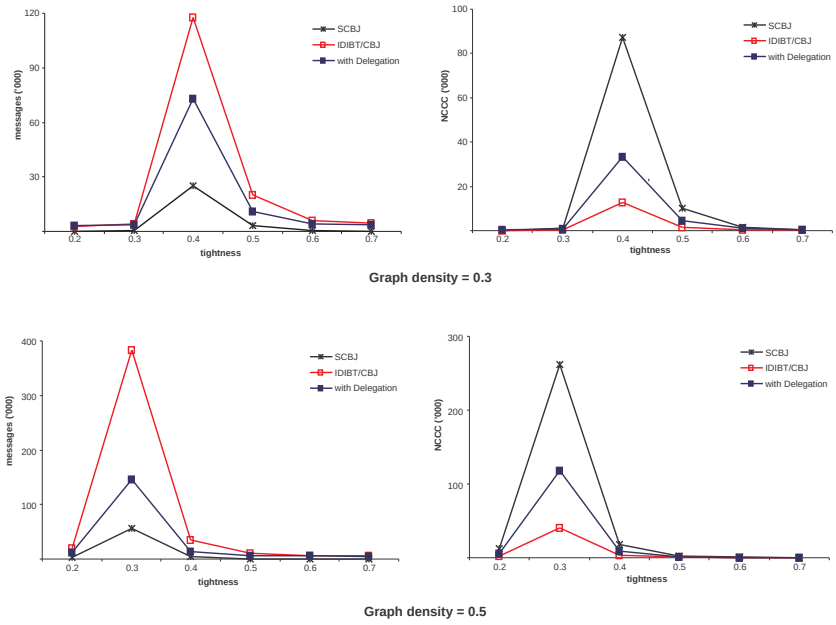

Fig. 4. Average message count and NCCC for solving DisCSPS with SCBJ, IDIBT/CBJ, and IDIBT/CBJ with delegation.

With respect to the NCCC, it appears that with delegation the average cost is higher than the corresponding cost for IDIBT/CBJ but lower than the averages for SCBJ. NCCC is a measure of the longest chain of sequential checks that can not be performed concurrently. Delegation appears to worsen performance on this metric because, firstly, it creates a physical chain of agents which can lengthen the sequential chain of constraint checks included in the final value of the metric. Secondly, delegation introduces some artificial delays in message passing, particularly for agents at the end of long message passing chains. It meant that in some cases the detection of conflicts with parents high up in a search tree was some times delayed. This resulted in intermediate agents having to abandon and reconstruct solutions, when such conflicts were discovered, with cost implications for the NCCC count.

Our motivation for delegation is to improve communication overhead of distributed backtracking in scenarios where the message passing is expensive relative to constraint checking but privacy is an issue. The results show that message passing is reduced with delegation compared to standard IDIBT/CBJ; however, this improvement comes with the cost of additional constraint checks. The results also show that by maintaining asynchrony in the search, constraint checking with delegation is lower than SCBJ. Message passing in SCBJ is lower but this is only achieved by violating privacy when partial solutions are passed from one agent to the next - and agents receive values they are not meant to see.

5 Dynamic delegation

Algorithms based on ABT [9,1] add links during search (as opposed to IDIBT/CBJ's preprocessing step). In such cases, precomputing the delegation paths before search is unlikely to be effective. Therefore, we now consider a dynamic delegation strategy, in which we allow agents to detect 3-cliques from the nogoods generated during a search, and then to chain together valid delegation paths. In this section, we describe our dynamic delegation method, implement it in a variant of ABT, and evaluate it experimentally.

5.1 Performing dynamic delegation

As mentioned earlier, in this form of delegation, coherent nogoods generated during a search are used to discover 3-cliques and to establish delegation paths. The key idea here is to allow an agent that receives nogoods to use the information to identify cliques and to determine if it can act as an intermediary for parent neighbours in the nogood.

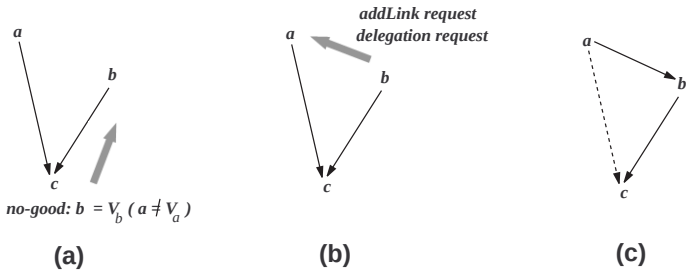

Fig. 5. Discovering cliques from nogoods.

The example in Figure 5 is used to illustrate the process. A_b receives the nogood ($b \neq V_b \Rightarrow (a = V_a)$) from which it can detect the 3-clique involving itself and both A_a and A_c. If the link from A_b to A_c is active, A_b can determine if it can perform delegation on behalf of the parent A_a to the child A_c. In this example, such an opportunity exists. Therefore, using Algorithm 3, A_b will send a delegation request to A_a for the delegation. After which the request is saved, to prevent A_b sending the same request repeatedly if there is thrashing along its path with both the parent and the child.

When A_a receives the delegation request, it processes the request with Algorithm 4[4]. A delegation request is rejected if either the forward link from the receiving agent to the target agent is inactive or the receiving agent is already delegating messages for a parent to the target agent. Otherwise, the request is accepted; which prompts the recipient to implement the delegation and to respond with an acceptance e.g. the acceptance from A_a to A_b, prompting agent A_b to relay A_a's decisions to A_c whenever ever it receives a value update from A_a; and agent A_a de-activates its forward link with A_c.

[4] Algorithm 4 uses the same data structures with Algorithm 1.

Algorithm 3: Agent A using nogoods to discover 3-cliques and delegations.

Input: $NG = \{(p_1, ...p_n) : p_i$ is var $\in rhs(NG)$; s is nogood sender$\}$
Data: Q: list of delegation requests sent by A

```
1 foreach p_i ∈ rhs(NG) do
2     if (p_i, (A, s)) ∉ Q then
3         send delRequest(p_i, A, s) to p_i;
4         Q ← Q ∪ (p_i, (A, s))
```

Algorithm 4: Agent A responding to delegation requests.

Input: $delRequest(A, (A_i, A_t)) : A_i$ is intermediate agent, A_t is delegation target

```
1 i ← index of A_i in children(A);
2 t ← index of A_t in children(A);
3 if (d[t] = false) ∨ (f[∗][t] = true) then
4     reject (A, (A_i, A_t));
5 /* accepting delegation */;
6 d[t] = false;
7 r[i][t] = true;
8 send message to A_i with {A_t : r[i][t] = true};
```

5.2 Algorithm modifications

For dynamic delegation, we modified ABT with partial synchronisation (ABTHyb) [1] as follows:

- Firstly, two new message types are introduced:
  - *delRequest*(A_i, A_j, A_k) - delegation request sent from an intermediate neighbour (A_j) to a parent (A_i) to inform the parent of the opportunity for delegating messages for A_k through A_j.
  - *delegation*(A_i, A_j, A_k) - an acceptance for a delegation request. This is sent from the parent A_i to the the intermediary A_j.
- Agents run the steps outlined in Algorithm 3 whenever nogoods are received from child neighbours. In the case that nogoods received contain values for unconnected agents, new links with these agents are first created before any delegation requests are sent.
- In response to delegation requests received, steps in Algorithm 4 are used to determine if such requests are accepted.
- Backward links between all agents remain active. Therefore nogoods are sent directly culprit parents irrespective of the delegation structures that exist at the time.
- When an agent sends a nogood, it moves into the synchronous phase of the search as described in [1]. With delegation, the agent will continue this partial synchronisation by holding on to any collated partial solutions until it returns to the asynchronous search.

5.3 Evaluation

The same problems from Section 4.4 were used to evaluate ABTHyb with dynamic evaluation. As previously, we also compared the modified algorithm with its ancestor and with SCBJ. However, for these evaluations we make SCBJ a nogood recording algorithm (and therefore it is referred to as SBT in the results for consistency), so that all three algorithms are compared on like terms. Furthermore, a total (*max-deg*) ordering is imposed on agents in all three algorithms.

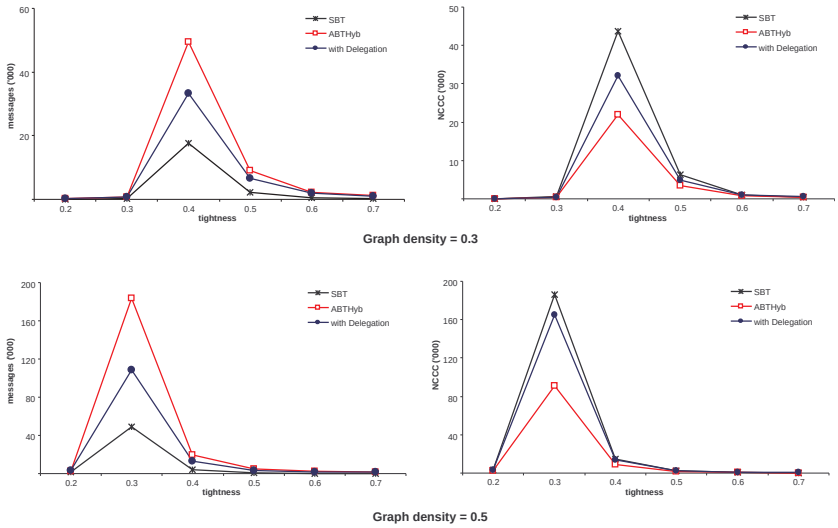

Fig. 6. Average message count and NCCC for solving DisCSPS with SBT, ABTHyb, and ABTHyb with delegation.

In Figure 6, we plot the averages of the evaluation metrics from the runs of the three algorithms. The results are consistent with the previous findings. They show that the synchronised algorithm still saves on message passing (although at the cost violating privacy) and that the NCCCs remain higher. Between the asynchronous algorithms, the effects of delegation is consistent with the earlier findings. Message passing cost is considerably reduced with delegation in place and there is the accompanying increase in the average NCCC count. Some reasons for the higher average NCCC count have been listed in Section 4.4, and these are still applicable.

6 Conclusion

We have introduced a new concept of delegation for improving the efficiency of message passing in distributed tree search. With delegation, agents can appoint intermediate neighbours for transmission of their local solutions to other child neighbours. The idea

reduces message passing by de-activating some forward links from agents with delegations, but it can enhance the search by improving the coherency of partial solutions received by agents while still preserving the privacy levels of an underlying algorithm.

We presented two forms of delegation for the prominent tree search strategies in DisCSP i.e. where links between unconnected agents are created prior to a search and where such links are created on the fly as required. We have shown, with empirical experiments, that both forms of delegation reduce message passing in asynchronous algorithms by as much as 50%. But the improvements come at the cost of additional non-concurrent checks, which we attribute to some implicit delays caused by the delegation of messages.

Overall, the results indicate that delegation is effective strategy for distributed tree search where the cost of message passing is significant relative to the cost of constraint checking and where privacy is an issue. For future work, we intend to extend delegation to other algorithms, such as ADOPT [7], and to improve the dynamic delegation algorithm. In particular, we are motivated by problems where the network connections are expensive, of different quality, and unreliable, and we are developing delegation heuristics which make use of this information.

References

1. I. Brito and P. Meseguer. Synchronous, asynchronous and hybrid algorithms for DisCSPs. In *Proc. of the 5th Int'l Workshop on Distributed Constraint Reasoning*, September 2004.
2. Z. Collin, R. Dechter, and S. Katz. On the feasibility of distributed constraint satisfaction. In *Proc. of the 12th Int'l Joint Conference on Artificial Intelligence, IJCAI*, pages 318–324, 1991.
3. Y. Hamadi, C. Bessière, and J. Quinqueton. Backtracking in distributed constraint networks. In *ECAI*, pages 219–223, Aug 1998.
4. Youssef Hamadi. Conflicting agents in distributed search. *Int'l Journal on Artificial Intelligence Tools*, 14(3):459–476, 2005.
5. R. Mailler and V. Lesser. Using cooperative mediation to solve distributed constraint satisfaction problems. In *Proc. of 3rd Int'l Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS 2004)*, 2004.
6. A Meisels, I. Razgon, E. Kaplansky, and R. Zivan. Comparing performance of distributed constraints processing algorithms., July 2002.
7. P. J. Modi, W. Shen, M. Tambe, and M. Yokoo. An asynchronous complete method for distributed constraint optimization. In *In Proc. The 2nd Int'l Joint Conference on Autonomous Agents & Multiagent Systems, AAMAS 2003*, pages 161–168. ACM, July 2003.
8. M. Yokoo. *Distributed Constraint Satisfaction:Foundation of Cooperation in Multi-agent Systems*. Springer, 2001.
9. M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara. Distributed constraint satisfaction for formalizing distributed problem solving. In *12th Int'l Conference on Distributed Computing Systems (ICDCS-92)*, pages 614–621, 1992.
10. M. Yokoo, T. Ishida, and K. Kubawara. Distributed constraint satisfaction for DAI problems. In M. N. Huhns, editor, *Proc. of the 10th International Workshop on Distributed Artificial Intelligence*, chapter 9. 1990.
11. R. Zivan and A. Meisels. Synchronous vs asynchronous search on DisCSPs. In *Proc. First European Workshop on Multi-Agent Systems (EUMAS)*, December 2003.