Avoiding Slow Links in Distributed Constraint Satisfaction

Muhammed Basharu[1]*, Kenneth N. Brown[1], and Youssef Hamadi[2]

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

Abstract. Distributed Constraint Satisfaction (DisCSP) algorithms assume an underlying communication network. We show that the presence of slow links in that network can have an adverse effect on algorithm performance. We propose two *delegation* methods for bypassing slow links, which (i) respect and (ii) ignore the existing priority orderings. We demonstrate in empirical tests that, when added to the IDIBT/CBJ algorithm, both methods reduce elapsed time while simultaneously reducing the number of messages, and that the method which ignores the priority ordering can achieve up to a 75% reduction in elapsed time.

1 Introduction

Distributed Constraint Satisfaction Problems (DisCSP) [1] formalise naturally distributed decision problems, where autonomous agents make local decisions, but must collaborate with each other to ensure that their decisions are compatible. Examples include scheduling joint oil pipeline usage [2] and target tracking with sensor networks [3]. DisCSPs are typically solved with distributed tree-based search, where a partial order of agents is used to record the progress of exploration. In these algorithms, agents send local solutions to their children (the set of neighbours below them in the ordering). Children in turn solve their local problems to be consistent with 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.

Distributed tree-based search can either be synchronous [4] or asynchronous [1,5]. Synchronised search closely resembles standard non-distributed search processes. Using a tree ordering, agents pass control up and down the tree, and each agent only becomes active when it has control. In asynchronous search, all agents may operate simultaneously, computing their own local solutions based on their current knowledge of their parents' decisions. Each agent is activated to perform its computations whenever it receives messages from its neighbours. Asynchronous search tends to have shorter run-times, since much of the computation is done in parallel and dead-ends can be identified

* 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.

early, but at the expense of some redundant chains of computation and higher network load.

Agents are assumed to be autonomous processes residing on different processors connected by some communication network. Message passing between these processors plays a critical role in the performance of the solution process. Delays in receiving messages not only prolong overall runtime but, in the case of asynchronous algorithms, can increase the amount of redundant search. Previous studies on the impact of random delays for individual messages have shown that performance consistently degrades as the length of delays increase (these are discussed further in Section 3).

In this paper, we consider the presence of persistent delays on individual links between agents in a DisCSP. These delays appear in the physical network layers either in the form of long term congestion on an Internet path between two agents or line-of-sight interference in radio sensor networks. We show that performance of asynchronous search algorithms can be adversely affected even when only a small percentage of links have such persistent delays. To overcome these effects, we propose two methods to allow agents to build faster logical networks around slow links by delegating agents with faster links to act as intermediaries for relaying messages. The intermediary agents will also asynchronously collate decisions they receive so that larger and more coherent partial solutions are transmitted on the faster links - thus the intuition is that fewer chains of redundant search are invoked. We will show that the methods, which can be implemented in existing DisCSP algorithms, preserve privacy levels. We also show the implementation of delegation in existing asynchronous algorithms can result in up to 75% savings in runtime in adverse network conditions.

The rest of this paper is structured as follows. We start with some preliminaries in Section 2 followed with an overview of related work on the effect of message delays in distributed search (Section 3). In Sections 4 and 5, the two approaches to delegation for by-passing slow links in DisCSP networks are presented. The results from experimental evaluations of both approaches are presented in Section 7.

2 Preliminaries

A DisCSP is a 4-tuple (X,D,C,A) where X is a set of n decision variables $(X_1, X_2, ..., X_n)$ and D is a set of domains $(D_1, D_2, ..., D_n)$ of possible values for the variables in X respectively. C is the set of constraint on the values of the variables. The constraint $C_k(X_{k1}, ..., X_{kj})$ is a predicate defined on the Cartesian product $D_{k1} \times ... \times D_{kj}$. Constraint C_k is satisfied if the values assigned to the variables involved satisfy the predicate. The set A = $\{A_1, A_2, ..., A_p\}$ is the 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 $f : \mathsf{X} \rightarrow \mathsf{A}, s.t. f(X_i) = A_j$.

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. It is assumed that each agent controls its own variables, and, as a default, knows only the domains of its variables and the constraints defined on them. The agents collaborate to find a global solution through message passing. A basic method for finding a global solution uses the distributed backtracking paradigm [1,5]. 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. 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 Message Delays in DisCSP Tree Search

There is some consensus from previous investigations into the effects of delays in message passing on distributed tree search performance. Results reported in [6,7] show considerable performance degradation of tree search algorithms as the delay on inter-agent links increase. The results reported by Bejar et al [3] found the same degradation in performance for uniform increases in delays on all links in their study with the Asynchronous Backtracking (ABT) [1] and Asynchronous Weak Commitment (AWC) [1] search algorithms. The study into the effect of non-uniform delays on links showed a performance degradation in ABT, however, the non-uniform delays appeared to benefit AWC. This prompted the authors to propose artificial delays as a source of randomisation to improve AWC's performance when link delays are uniform.

The study by Zivan and Meisels [7] also looked at the effects of fixed uniform message delays and random delays for individual messages on distributed tree search performance. The work investigated the the impact of the different delays in ABT, single and concurrent multi-context synchronous search algorithms. The results on the impact of fixed uniform delays agree with the findings from [3], showing the considerable degradation in performance as the size of delays increases and that the synchronous single context search was the worst affected. For ABT, message passing doubled and runtime increased almost three-fold when the range of delays was increased to the equivalent of 50-100 computation steps. They also found that the multi-context synchronous search algorithm was the most robust to the effects of the increasing delays, its message passing was unchanged and runtime increased by just 40%.

The effect of slow links on performance of distributed applications have also been studied in the distributed and grid computing communities. Similar findings of the sensitivity to the presence of slow links have been observed. For example, Plaat et al [8] found that applications requiring frequent synchronisation are particularly sensitive to slow links in networks. They considered application specific methods for dealing with slow links like combining messages to reduce the traffic on slow links or reducing the number of synchronisation points. In related work, Koenig and Kale [9] use dynamic load balancing to overcome the effects of slow links i.e. by partitioning load for processors by exploiting the knowledge of link delays so that communications on slow links are minimised. The idea of routing messages to avoid slow links is the obvious method for dealing which such links. Routing algorithms, applicable to networking in general, are used to find the shortest (or fastest) paths between any two nodes in a network (e.g. [10]). However, privacy (in DisCSP terms) is not a major issue in network routing as such there are fewer restrictions on the paths which messages are sent between nodes.

In a preliminary step for our study, we investigated the effects of persistent delays on a random subset of the links in the network. We used the Interleaved Distributed Intelligent Backtracking Algorithm (IDIBT/CBJ)[5] to solve random DisCSPs, first without delays, and then on the same problems with delays equivalent to 50 constraint checks on 5% of the links. The results, displayed in Table 1, show a decrease in performance for all metrics measured. However, what stands out from the results is the fact that the number of backtracks and obsolete backtrack messages increase. This indicates that slow links do not just lengthen the time between decisions made by agents but they also cause agents to make more decisions with out-dated information. Since we are seeing such a degradation for just a few slow links, and that slow links may be a feature of whatever communication network the DisCSP algorithms are using, we now turn our attention to methods for avoiding the use of such links.

metric	without delays	with delays	change
Runtime	18,223	20,849	+14%
Message count	181,207	195,183	+7%
Backtracks	32,521	37,572	+15%
Obsolete Backtracks	26,528	31,261	+17%

Table 1. Effect of delays on 5% of the links on performance of IDIBT/CBJ (all metrics averaged over 20 problems $\langle n = 20, d = 10, p_1 = 0.5, p_2 = 0.4\rangle$).

4 Selective Directed Delegation

The idea of *delegation*, presented in [11], describes a process in which a parent sends its messages to only one of a pair of children, and delegates that child to forward those messages to the other when it is ready. If the third agent is also a child of the parent, as in [11], then this process does not introduce any new links or agents into the DisCSP, and should not involve a violation of privacy. Delegation in [11] was implemented by local algorithms which chained together overlapping 3-cliques of agents, producing maximal delegation paths while maintaining privacy. The algorithms were applied to networks with no delays, and it was demonstrated that delegation was a hybrid of synchronous and asynchronous search - some partial local synchronisation is introduced, but agents are still free to act autonomously - and was midway between the two in performance: it reduced the message count of asynchronous search at the expense of non-concurrent constraint checks (NCCCs) [12], while reducing the NCCCs of synchronous search at the expense of messages.

We now show how to apply the idea of delegation for bypassing slow links in a network. Children will select intermediaries to relay messages to them, but only if the paths via the intermediaries are faster then the direct link from the parent. Intermediaries have the advantage of not only providing faster routes for messages, but also of sending larger coherent solutions to the children, since they can asynchronously collate decisions they are meant to relay. More coherent solutions should reduce the amount of redundant search.

We introduce a new algorithm for performing delegation amongst agents in DisCSPs - Selective Directed Delegation (SDD). Unlike the local concurrent algorithm from [11], in SDD agents select intermediaries to relay parent decisions to them rather than selecting children to relay decisions to lower children. As with delegation, SDD is intended to be implemented in distributed tree search algorithms after an ordering of agents has been established (for example, by DisAO [5]) and the ordering remains static during search. We also assume that during the ordering phase, agents use the opportunity to get estimates of delays on links with their neighbours.

Algorithm 1: SDD: Main loop for each agent A

```
/* n parents $p_1, ..., p_n$, m children $c_1, ..., c_m$ */
Data: d: array of n ints, d[i] set to delay on link to $p_i$
Data: pl: array of n booleans, initially true
Data: cl: array of m booleans, initially true
Data: a: array of $n \times n$ booleans, initially false
Data: ld: array of $n \times n$ ints
Data: r: array of n×n booleans, initially false
Data: f: array of n×m booleans, initially false
1  foreach child $c_i$ do
2      send message to $c_i$ containing { $(p_1, d[1]), ..., (p_n, d[n])$ }
3  while true do
4      message ← getMsg()
5      if message sender is a parent then
6          update a
7          update ld
8          findIntermediaries()
9      if message is $cutLink(A, c_i)$ then
10         cl[i] = false
11     if message is $relayDecision(p_i, c_j)$ then
12         f[i][j] = true
13     if message is $cancelRelayDecision(p_i, c_j)$ then
14         f[i][j] = false
```

Algorithms 1 and 2 outline the processes each agent undertakes to perform delegation with SDD. The local data structures can be interpreted as follows: **d**[i] is the delay on the link between an agent A and a parent p_i; **pl**[i] indicates that the link between A and p_i is active; **cl**[i] indicates that the link between A and the child c_i is active; **a** is the adjacency matrix of connectivity between parents such that **a**[i][j] is true if there is a link between p_i and p_j; **ld**[i][j] is the delay on the link between p_i and p_j; **r**[i][j] indicates that p_j relays p_i's decisions to A; and **f**[i][j] states that A relays decisions from p_i to c_j.

The algorithms describe the process where after establishing an ordering each agent sends information on link delays with its parents to all its children (Alg. 1, lines 1-2).

Algorithm 2: SDD: Selecting intermediaries to relay decisions from parents.

```
Data: data structures from Algorithm 1
foreach parent $p_i$ in order and $p_i$ is not relaying any decisions to A do
    choose $p_j$ s.t. $p_i <_o p_j$, pl[j] == true, a[i][j] == true, and ld[i][j] is fastest link
    if ld[i][j] < d[i] then
        newDelay ← ld[i][j] + d[j]
        if pl[i] == true then
            send $cutLink(p_i, A)$ to $p_i$
            pl[i] ← false
        else
            send $cancelRelayDecision(p_i, A)$ to $p_k$ (where r[i][k] == true)
            r[i][k] ← false;
        send $relayDecision(p_i, A)$ to $p_j$
        d[i] ← newDelay
        r[i][j] ← true
if new intermediary found for any $p_i$ then
    foreach child $c_i$ do
        send message to $c_i$ containing { $(p_1, d[1]), ..., (p_n, d[n])$ }
```

Each agent receives the link delay information from its parents and uses it to update the adjacency matrices to establish connectivity (and delays) amongst parents, as well as to detect all 3-cliques including 2 parents (6-8). For each parent p_i involved in at least one 3-clique (that has not already been itself chosen as an intermediary), the agent selects the intermediate parent p_j, with whom it has an active link, to relay decisions from p_i if the cummulative delay via p_j is faster than its direct link with p_i[3] (Alg. 2). After selecting p_j as an intermediary, a message is sent to p_i to stop p_i sending decisions to A (7) and another message is sent to p_j requesting it to relay all decisions received from p_i (11). The new delay for receiving messages from p_i via p_j is saved and then sent to all A's children. As information on link delays are updated, agents can find new (and faster) intermediaries for receiving decisions from some parents. As such, previous relay requests to intermediate parents can be revoked (9-10). Agents continue making changes to the network until no further improvements can be found. The process terminates when no delegation related messages are exchanged[4] and a search for a solution can proceed.

During a search, agents implement delegation using Algorithm 3. When an agent receives the decision of a parent p_i, it updates its *AgentView* and finds a compatible value (Alg. 3, lines 1-2). The agent then initiates a message for each of its children with active links and places its value in the message if it has one (5-6). And it also places p_i's value in the message if c_j uses it as an intermediary for relaying decisions from p_i (7-8). Agents also use intermediaries for sending backtrack decisions to their parents. Therefore, when an agent receives a $Back$ message meant for a parent it relays

[3] Ties are broken in favour of the nearest parent.

[4] We assume that there a separate process to detect this.

Algorithm 3: SDD: Responding to a decision from Parent $P = x_p$ by A during search.

```
Data: data structures from Algorithm 1
1 update AgentView($P, x_P$)
2 choose $A$'s decision $A$ = $x_a$
3 foreach child $c_i$ s.t. cl[i] == true do
4     message = new InfoVal(from: $A$ to $c_i$)
5     if $x_a \neq$ null then
6         message.contents ∪ $(A = x_a)$
7     if $f[indexOf(P)][c_i]$ == true then
8         message.contents ∪ $(P = x_P)$
9     send message
```

the message to parent if there is an active link between them or to the intermediary it receives the parent's decisions from.

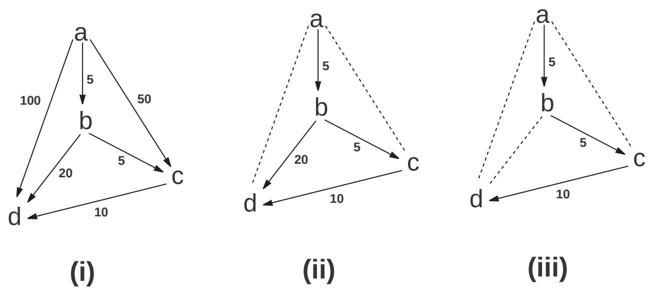

Fig. 1. Establishing directed delegation paths.

The example in Figure 1 is used to illustrate the process of establishing directed delegation paths in an ordered DisCSP. Figure 1(i) is a DisCSP with delays on the individual links between agents shown. To initiate the process, each agent sends details of its connectivity with its parents to each of its children e.g, A_b sends out the message $\{A_b \rightarrow (A_a, 5)\}$. A_c and A_d use this message to detect the cliques with A_a. A_c selects A_b to relay messages from A_a and deactivates the affected link. A_d also selects A_b as an intermediary and as a result A_d has to keep the link with A_b active ignoring the faster path via A_c (ii). After choosing its intermediary, A_c sends an update to A_d with its new delay to A_a. With this update, A_d revises its decision to use A_b as a relay for A_a selecting A_c instead. Because A_b no longer serves as a relay, A_d then selects A_c to also relay decisions from A_b. The process then terminates with the network configuration shown in (iii).

As a result of the delegations created, during a search whenever A_b receives a decision from A_a it evaluates the decision and relays A_a's value along with its own value to A_c and so on. The same delegations are also used during backtracking. So a *Back* message from A_d to A_a will be sent on multiple hops passing through A_c and A_b.

5 Selective Undirected Delegation

Restricting the set of possible intermediaries to parents may be a limitation. The implication is that, without re-ordering agents, some significantly slow links can be left active in a network with adverse effects on search performance. For example, in the DisCSP shown in Figure 2(i), SDD is forced to retain the slowest link in the network. This is the motivation for our second extension to delegation in the Selective Undirected Delegation (SUD) method, which allows agents choose shared children as relay nodes for receiving values from parents (Fig. 2(ii)). The key decision in delegation is that the agent chosen to relay messages from a parent must normally receive the parent's decisions during a search. This rules out the possibility of selecting a parent of a parent as a relay node in undirected delegation as it results in some loss of privacy.

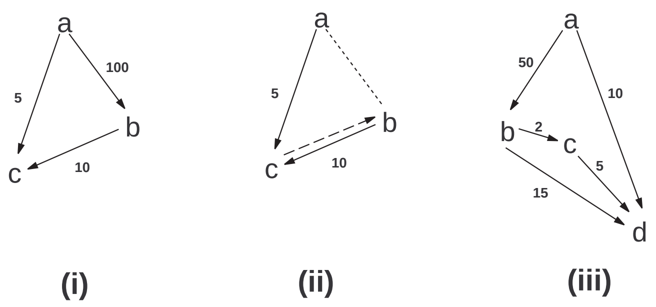

Fig. 2. Limitations of Selective Directed Delegation.

In SUD, agents run similar processes to SDD with the exception that agents can choose shared children as relays for receiving parents' values. However, since agents only make decisions about receiving decisions from parents, delegation with SUD allows agents to keep some links with children partially active. The example in Figure 2(iii) is one such case, where A_b chooses A_d to relay decisions from A_a and A_d chooses A_c to relay decisions from A_b. Note that we assume A_b cannot request A_a's messages to be routed through A_d then A_c, since A_c would then receive message it would not be entitled to see.

Like SDD, agents make local decisions process to reduce a DisCSP network. The steps are outlined in Algorithms 4 and 5, which differ from Algorithms 1 and 2 with the inclusion of children as possible intermediaries. The process initiates with agents informing all their neighbours about connectivity with all other neighbours, as well as providing information about delays on links to the neighbours. As usual 3-cliques are detected (but this time including children) and intermediaries for relaying parent decisions are selected. Agents may also revise chosen intermediaries as they receive updates from their neighbours. Again the process terminates when no further changes to the network can be made.

Algorithm 4: SUD: Main loop for each agent A

```
/* q neighbours $n_1, ..., n_q$, g parents $p_1, ..., p_g$, m children $c_1, ..., c_m$ */
Data: d: array of g ints, d[i] is delay on link with $n_i$
Data: nl: array of q booleans, initially true /* nl[i]==true if link with $n_i$ is active*/
Data: a: array of $q \times q$ booleans, initially false
Data: ld: array of $q \times q$ ints
Data: r: array of q×q booleans, initially false
Data: f: array of q×q booleans, initially false
1  foreach neighbour $n_i$ do
2      send linkInfo message to $n_i$ containing { $(n_1, d[1]), ..., (n_q, d[q])$ }
3  while true do
4      message ← getMsg()
5      if message is linkInfo then
6          update a
7          update ld
8          findIntermediaries()
9      if message is $cutLink(A, c_i)$ then
10         nl[i] = false
11     if message is $relayDecision(p_i, c_j)$ then
12         f[i][j] = true
13     if message is $cancelRelayDecision(p_i, c_j)$ then
14         f[i][j] = false
```

We use the example in Figure 3 to illustrate the process of establishing undirected delegation paths with SUD. The figure shows an ordered DisCSP network with delays on individual links between agents. At the start, each agent informs its neighbours of the delays between itself and other neighbours. For example, A_d will send the message $\{A_d \rightarrow ((A_a, 100), (A_b, 5), (A_c, 10))\}$ to all its neighbours. Agents receive these messages and use them to build a profile of connectivity and delays in the network. In the next step, both A_d and A_c will detect the opportunity to use A_b as the relay for receiving messages from A_a and take action accordingly (Fig. 3(ii)).

Once A_b is selected as an intermediary, A_c has to maintain its direct link with it therefore A_c will not evaluate the path $A_b \rightarrow A_d \rightarrow A_c$. After updating the link delays, both A_c and A_d send updates to all their neighbours. In this case, the update from A_d to A_c will indicate that the cummulative delay to A_a is now 7 time units. Using this update, A_c cancels the relay request with A_b, cuts the direct link with A_b, and then requests that A_d relay the decisions for A_a and A_b to it (Fig. 3(iii)). After A_c sends the updates of its delays, the agents can no longer find any improvements to the message passing delay and the process terminates. As with SDD, during a search A_a will only

Algorithm 5: SUD: Selecting intermediaries to relay decisions from parents.

```
Data: data structures from Algorithm 4
foreach parent $p_i$ in order and $p_i$ is not relaying any decisions to A do
    choose $n_j$ s.t. $p_i <_o n_j$, nl[j] == true, a[i][j] == true, and ld[i][j] is fastest link
    if ld[i][j] < d[i] then
        newDelay ← ld[i][j] + d[j]
        if nl[i] == true then
            send $cutLink(p_i, A)$ to $p_i$
            nl[i] ← false
        else
            send $cancelRelayDecision(p_i, A)$ to $p_k$ (where r[i][k] == true)
            r[i][k] ← false;
        send $relayDecision(p_i, A)$ to $p_j$
        d[i] ← newDelay
        r[i][j] ← true;
if new intermediary found for any $p_i$ then
    foreach neighbour $n_i$ do
        send message to $n_i$ containing { $(n_1, d[1]), ..., (n_q, d[q])$ }
```

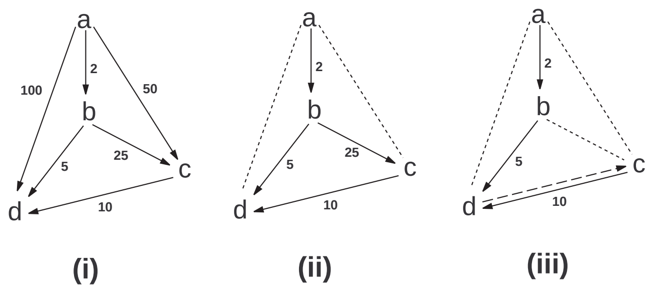

Fig. 3. Establishing undirected delegation paths with SUD.

send its decisions to A_b and A_d relays all decisions from A_a and A_b to A_c. All *Back* messages from A_c will be relayed through A_d to the target culprits.

6 Analysis

Termination The SDD and SUD processes are guaranteed to terminate after a finite number of delegation decisions have been made. First of all, delegations are acyclic - agents do not choose intermediaries for communicating with their children. So a pair of connected agents can not simultaneously make decisions about the connections between them. Secondly, there are a finite number of possible improvements that can be made by agents. Therefore at some point agents will stop sending new information to their neighbours and the processes will settle on current configurations. And finally, agents

Algorithm 6: SUD: Responding to a parent's ($P = x_P$) decision by A during search.

```
Data: data structures from Algorithm 4
1  update AgentView ($P, x_P$)
2  foreach $p_i$ s.t. $P <_o p_i$ and r[i][j] == true do
3      send InfoVal($P$, $x_P$) to $p_j$ /* relay decision immediately */
4  choose A's decision
5  foreach child $c_k$ s.t. cl[k] == true do
6      message = new InfoVal(from: $A$ to $c_i$)
7      if $x_a \neq$ null then
8          message.contents ∪ $(A = x_a)$
9      if f[$indexOf(P)$][$c_i$] == true then
10         message.contents ∪ $(P = x_P)$
11     send message
```

only revise decisions in favour of faster paths i.e. an agent will not restore a link after it has been cut, therefore agents can not oscillate between different network states.

Correctness In order to prove the correctness of both delegation algorithms, we will show that: (1) any chain of delegations between any two agents A_i and A_j is faster if the direct link between them is cut, (2) undirected delegation paths are acyclic, (3) no agent A_j receives the same decision $A_i \leftarrow v$ from more than one source, (4) privacy is preserved in the delegation paths created. For these proofs, we assume that agents implement the DisAO algorithm to establish an ordering prior to a search and that tautological links between unconnected parents are created in the process as well.

Theorem 1 *If the direct link from A_i to A_j is cut, the cummulative delay on the chain of intermediate relays $R_1, R_2, ...R_n$ is lower than the delay on the link $A_i \rightarrow A_j$.*

Proof. From the algorithms, an agent A_j selects the intermediary A_k to relay messages from A_i when the cummulative delay (D) via the relay is faster than the direct link with A_i i.e. $(D_{i,k} + D_{j,k}) < D_{i,j}$. A_k, in turn, will only select A_m to relay messages from A_i if the condition $(D_{k,m} + D_{i,m}) < D_{i,k}$ holds. And as a result, $(D_{i,m} + D_{k,m} + D_{j,k}) < D_{i,j}$ which by extension holds true for any number of intermediate relays on the delegation path between A_i and A_j.

Theorem 2 *Delegation paths including partially active links in SUD are acyclic.*

Proof. For SUD, a link is kept partially active if the lower agent on the link A_k relays messages to its parent A_j from a higher parent A_i and A_k in turn selects A_m to relay messages from A_j to it. Thus creating two bidirectional paths, $A_i \rightarrow A_k \rightarrow A_j$ and $A_j \rightarrow A_m \rightarrow A_k$. A_j will only use A_k to communicate with A_i, therefore no messages meant for A_i will pass through A_m. From Theorem 1, A_k can not in turn use the path $A_m \rightarrow A_j$ to send messages to A_i since its direct link with A_i is the fastest available link. Therefore, there can be no cycles in message passing amongst the agents.

Theorem 3 *No agent A_j receives the same decision $A_i \leftarrow v$ from two separate agents.*

Proof. (Omitted) by inspection of the algorithms.

Theorem 4 *An agent A_j can receive a decision A_i by delegation if and only if it can receive it without delegation, given the ordering induced by DisAO (and so privacy of messages are not violated).*

Proof. (Omitted) by inspection of the algorithms.

7 Evaluations

We carried out evaluations of SDD and SUD by implementing them primarily in IDIBT / CBJ and with additional tests in ABT-Hyb [13]. We studied performance of the algorithms as the quality of network links degraded and their performance with different distributions of link delays. In the experiments, we varied the number of slow links in the DisCSP networks; the delays on these links were normally distributed over the equivalent of 10 to 100 constraint checks[5]. We also tested the methods on DisCSPs where the delays on links were exponentially distributed.

All algorithms were implemented in an environment simulating asynchronous parallel activity by agents. A shared clock was used to measure elapsed time where each tick on the clock is equivalent to a constraint check. Each agent in the simulator is triggered into action when it receives messages. The agent reads all its messages and performs the necessary computation. The number of constraint checks performed is counted and used to simulate the time which the agent is busy and unable to process any other messages (i.e. the agent is blocked). Messages from each agent are time-stamped for delivery at the end of its computation time plus any delays on links between it and the recipients. The minimum message delay (for good links) is one tick of the clock. As in [7], a separate mailer agent is used to handle message passing between agents - messages are sent when delivery time is reached or held back until the recipient is unblocked.

In IDIBT/CBJ and the extensions, a single search context is used (i.e. NC=1) and the algorithms were modified slightly to allow agents to process messages in packets. A *max-deg* heuristic was used for ordering agents with DisAO prior to each run. The message count for the extended versions include the count for establishing the orderings as well as the messages exchanged in performing delegations. The results reported here are from three experiments on 20 random DisCSPS $\langle n = 20, d = 10\rangle$ with sparse, medium, and dense graphs - at the complexity peak for each level of connectivity. We recorded the runtime (i.e. NCCCs plus NCCC-equivalent delays), message count, and the number of obsolete backtracks for the evaluations.

Figures 4 and 5 plot the savings with delegation and the number of obsolete $Back$ messages, respectively, from attempts to solve DisCSPs with medium density $\langle p_1 = 0.5, p_2 = 0.4\rangle$. Although it is not shown, performance of the basic IDIBT/CBJ does

[5] The delays were sampled from a truncated normal distribution with $\mu = 0, \sigma = 1$ returning values between [-3.6,+3.6], which were scaled to the range [10,100].

degrade as expected as the percentage of slow links in the network increases. Figure 4 shows that there are savings with both delegation strategies even when the majority of the links are slow. First, we note that both delegation methods show a decrease in the number of messages, even though each time we introduce an intermediary we are requiring extra messages - the savings in redundant search clearly compensates for this overhead in most cases. Secondly, we note that the savings in NCCC, which includes the delay and so simulates total elapsed time, are more significant, and are greatest for SUD. We gain close to an order of magnitude speed up for most cases.

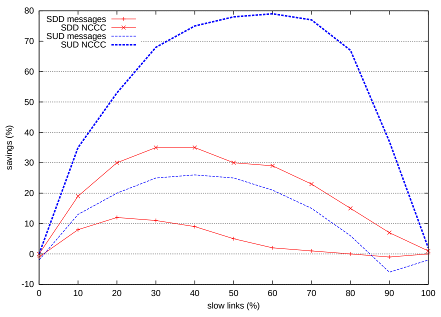

Fig. 4. Savings in runtime and messages with delegation as the number of slow links increase.

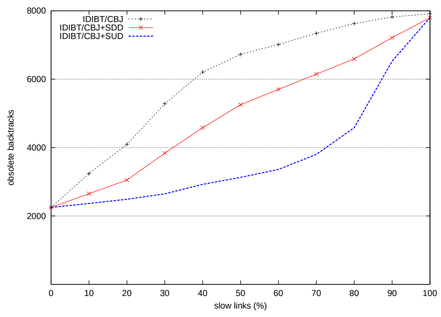

Fig. 5. Number of obsolete backtracks sent from runs plotted in Figure 4.

The same experiment was repeated for dense $\langle p_1 = 0.8, p_2 = 0.3\rangle$ and sparse $\langle p_1 = 0.3, p_2 = 0.5\rangle$ DisCSPs. The savings on the recorded metrics are plotted in Figures 6 and 7 respectively. These results show that the savings on sparse graphs, though considerable, were not as significant as the earlier findings. This is expected as there are fewer 3-cliques in the sparse networks and thus fewer opportunities to bypass the slow links.

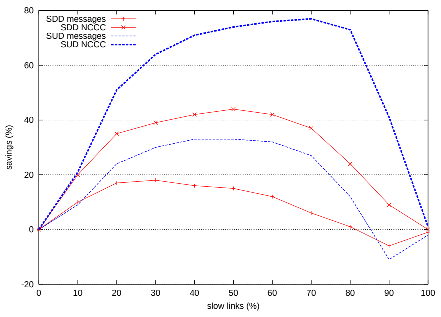

Fig. 6. Savings in runtime and message passing for SDD and SUD on dense DisCSPs.

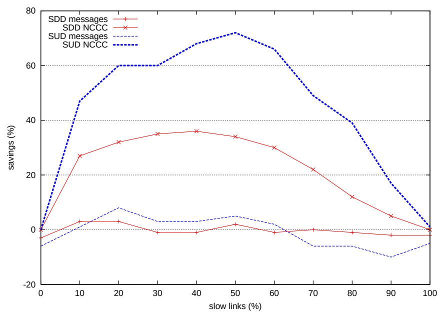

Fig. 7. Savings in runtime and message passing for SDD and SUD on sparse DisCSPs.

All the results presented so far show that there were little, if any, savings in runtime when all the links in the network were slow. We attribute this to the fact the delays on the slow links are distributed normally. Therefore majority of these delays were clustered around the mean values, which meant that there were few worse links to avoid and few faster links available to exploit. To confirm this, we ran new experiments with the same problems, but this time using an exponentially decaying distribution of link delays[6]. Performance of the delegation methods against the basic IDIBT/CBJ for the sparse problems are shown in Figure 8. Compared to Figure 7, there are consistent and significant savings in runtime for almost all cases with slow links. Critically, when all links have delays on them there are still savings of about 50% in runtime.

We ran additional tests for both delegation methods with ABT-Hyb and achieved similar results - assuming that DisAO is used to order agents prior to search and new links are created in that process as well. Results from runs on problems with $p_1 = 0.5$ and link delays distributed normally are shown in Figure 9.

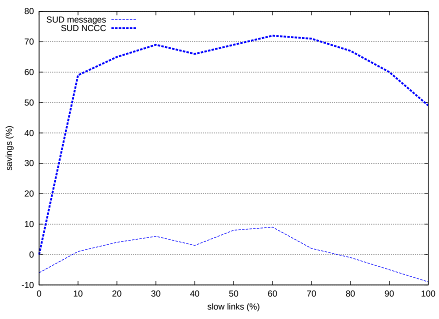

Fig. 8. Savings for SUD on sparse DisCSPs with exponential distribution of link delays.

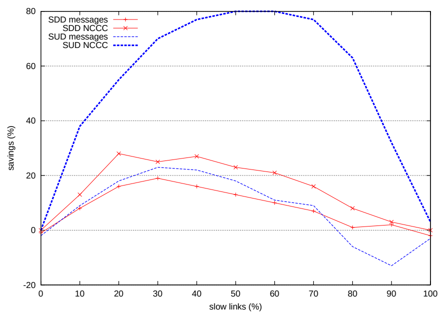

Fig. 9. Savings for SDD and SUD implemented in ABT-Hyb.

8 Summary

Asynchronous distributed tree search algorithms are sensitive to delay on links between agents. Performance of these algorithms, in terms of runtime and message passing, can degrade considerably even when only a small percentage of links in a network have persistent delays on them. The presence of such links also cause agents to invoke additional chains of redundant search which shows up as increases in the number of obsolete backtracks received.

We introduced the idea of delegation for DisCSP tree search to reduce the number of active links in DisCSP networks while preserving privacy, by having agents select intermediaries for relaying messages. In this paper, we extend the idea for bypassing slow links in networks in two ways. First, the Selective Directed Delegation method

[6] Sampling a distribution with $\lambda = 0.85$ and scaling the resulting values to the range [10,100].

allows agents to cut direct links with parents only if there are faster paths via some parent intermediaries. Secondly, the Selective Undirected Delegation method breaks the ordering of agents and allows selection of shared children as relay nodes. Algorithms for performing both forms of delegation were presented and are shown to be correct while also preserving privacy.

The two delegation methods were applied to two algorithms (IDIBT/CBJ and ABT-Hyb) and tested on random DisCSPs. The results of the evaluation showed that bypassing slow links with delegation improves runtime over the original algorithm, achieving savings of up to 75%. We also showed that delegation achieved considerable savings in runtime when all links have delays and these delays are distributed exponentially.

Our future work will focus on dynamic delegation in response to observed delays during search i.e. when links are intermittently slow or fast. We will also explore more selective delegation methods only bypassing really slow links. Finally, we also plan to study the new delegation methods in other algorithms.

References

1. M. Yokoo, *Distributed Constraint Satisfaction:Foundation of Cooperation in Multi-agent Systems*. Springer, 2001.
2. F. Marcellino, N. Omar, and A. V. Moura, "The planning of the oil derivatives transportation by pipelines as a distributed constraint optimisation problem," in *Workshop Distributed Constraint Reasoning (DCR'07)* (A. Petcu, ed.), pp. 1–15, January 2007.
3. R. Béjar, C. Domshlak, C. Fernández, C. Gomes, B. Krishnamachari, B. Selman, and M. Valls, "Sensor networks and Distributed CSP: communication, computation and complexity.," *Artificial Intelligence*, vol. 161, pp. 117–147, January 2005.
4. R. Zivan and A. Meisels, "Synchronous vs asynchronous search on DisCSPs," in *Proceedings of the First European Workshop on Multi-Agent Systems (EUMA)*, 2003.
5. Y. Hamadi, "Conflicting agents in distributed search.," *International Journal on Artificial Intelligence Tools*, vol. 14, no. 3, pp. 459–476, 2005.
6. M.-C. Silaghi and B. Faltings, "Asynchronous aggregation and consistency in distributed constraint satisfaction.," *Artificial Intelligence*, vol. 161, pp. 25–53, January 2005.
7. R. Zivan and A. Meisels, "Message delay and DisCSP search algorithms," *Annals of Mathematics and Artificial Intelligence*, vol. 46, pp. 415–439, April 2006.
8. A. Plaat, H. E. Bal, and R. Hofman, "Sensitivity of parallel applications to large differences in bandwidth and latency in two-layer interconnects," *Future Gener. Comput. Syst.*, vol. 17, no. 6, pp. 769–782, 2001.
9. G. A. Koenig and L. Kalé, "Optimizing distributed application performance using dynamic grid topology-aware load balancing.," in *Proc. 21st International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 1–10, IEEE, March 2007.
10. IEEE, "Routing information protocol."
11. M. Basharu, K. Brown, and Y. Hamadi, "Delegation in tree-search for distributed constraint satisfaction," in *Proc. Workshop on Distributed Constraint Reasoning (DCR-07)*, January 2007.
12. A. Meisels, I. Razgon, E. Kaplansky, and R. Zivan, "Comparing performance of distributed constraints processing algorithms.," in *Proc. Workshop on Distributed Constraint Reasoning (DCR'02)*, pp. 86–93, July 2002.
13. I. Brito and P. Meseguer, "Synchronous, asynchronous and hybrid algorithms for DisCSPs," in *Proc. Workshop on Distributed Constraint Reasoning (DCR'04)*, September 2004.