Efficient, Superstabilizing Decentralised Optimisation for Dynamic Task Allocation Environments

Kathryn S. Macarthur, Alessandro Farinelli*
Sarvapali D. Ramchurn and Nicholas R. Jennings
School of Electronics and Computer Science, University of Southampton, SO17 1BJ, UK
*Department of Computer Science, University of Verona, 15 I-37134, Italy
{ksm08r, sdr, nrj}@ecs.soton.ac.uk, *alessandro.farinelli@univr.it

ABSTRACT
Decentralised optimisation is a key issue for multi-agent systems, and while many solution techniques have been developed, few provide support for dynamic environments, which change over time, such as disaster management. Given this, in this paper, we present Bounded Fast Max Sum (BFMS): a novel, dynamic, superstabilizing algorithm which provides a bounded approximate solution to certain classes of distributed constraint optimisation problems. We achieve this by eliminating dependencies in the constraint functions, according to how much impact they have on the overall solution value. In more detail, we propose iGHS, which computes a maximum spanning tree on subsections of the constraint graph, in order to reduce communication and computation overheads. Given this, we empirically evaluate BFMS, which shows that BFMS reduces communication and computation done by Bounded Max Sum by up to 99%, while obtaining 60–88% of the optimal utility.

Categories and Subject Descriptors
I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence—*Multiagent Systems*; G.2.2 [**Discrete Mathematics**]: Graph Theory

General Terms
Algorithms, Experimentation, Theory

Keywords
Distributed constraint optimization, disaster management

1. INTRODUCTION
Multi-agent systems have been advocated as a key solution technology for coordinating the activities of rescue forces for disaster management. Specifically, coordination in such domains can be conveniently framed as a distributed constraint optimisation problem [13]. Many solution techniques, such as ADOPT (Asynchronous Distributed OPTimisation) [10], DPOP (Distributed Pseudotree Optimization Procedure) [11] and Bounded Max Sum (BMS) [6] have been proposed to solve such optimisation problems. Now, while complete algorithms such as ADOPT and DPOP guarantee optimal solutions, they use a lot of communication and computation. On the other hand, approximate algorithms like BMS tend to incur a lower coordination overhead, and can provide bounds on the quality of the approximation they give [6].

However, very few algorithms exist that reduce redundant communication and computation when used in dynamic environments. Nonetheless, dynamism is a key issue for search and rescue in disaster management: the environment evolves over time, and new information becomes available as time progresses. For example, information about civilians to rescue from buildings may change, and new rescue agencies can arrive, at any time, to help with the rescue effort. In such applications, optimal solutions are most desirable, but may not be achievable within the available time. Thus, good quality approximate algorithms can be used, due to their smaller overheads, but only if they are able to produce good quality solutions. Now, it is also important that an algorithm can always recover from environmental change gracefully, so as not to enter an unsafe state during recovery. This is formally known as *superstabilization* [4]. In more detail, in order for an algorithm to be superstabilizing, it must be super-stabilizing [3], and a pre-defined *passage predicate* must hold at all times. More specifically, a self-stabilizing algorithm must be distributed across a number of agents, and must be able to return those agents to some legitimate state after a change in the environment. This legitimate state is defined with a *legitimacy predicate*, which, when invalidated, forces the algorithm to return the agents to a state where the predicate holds in a finite amount of time. More generally speaking, in a superstabilizing algorithm, the passage predicate must hold at all times, including whenever the legitimacy predicate does not.

To date, few algorithms are suited for use in dynamic environments, and instead would need to be re-run, incurring unneeded overheads without superstabilization guarantees. Exceptions to this include SDPOP (Superstabilizing DPOP) [12], which is superstabilizing, but requires a prohibitive amount of communication and computation on large scale scenarios. Another algorithm that partially fits our requirements is the Fast Max Sum (FMS) algorithm [13], which reduces the communicational and computational impact of a change in the environment. However, FMS is only proven to converge to an optimal solution on specific[1] problem instances, and is therefore not general enough to be applied to more realistic arbitrary disaster management environments.

Against this background, in this paper, we propose a novel algorithm for performing superstabilizing distributed constraint optimisation in dynamic environments. In more detail, our algorithm provides approximate solutions with quality guarantees for constraint graphs with arbitrary topologies[2], while reducing communication and computation. In particular, this paper advances the state of the art in the

[1]Where the underlying constraint graph contains no cycles.
[2]Even those in which the underlying graph contains cycles

following ways: first, we present iGHS, an iterative variant of the GHS algorithm (named after the authors Gallager, Humblet and Spira) [7], which computes a maximum spanning tree of the constraint graph in BMS. Second, we present Bounded Fast Max Sum (BFMS): an efficient, superstabilizing constraint optimisation algorithm, which combines iGHS with principles from FMS and BMS.

The rest of this paper is structured as follows. In Section 2 we discuss the relevant background for our work. Next, we formulate our problem in Section 3, and present our iGHS algorithm in Section 4. We present BFMS in Section 5, and empirically evaluate it in Section 6. We discuss work related to iGHS in Section 7, and, in Section 8, we conclude.

2. BACKGROUND

Here, we present the necessary background for our work. We begin with a discussion of Max Sum and two particular variants that are useful in applying Max Sum to arbitrary dynamic environments: first, Fast Max Sum, used to reduce overheads caused by recomputation in dynamic environments, and second, Bounded Max Sum, which uses approximation to allow application of Max Sum to arbitrary environments. Finally, we discuss GHS, which is a key element of Bounded Max Sum, used to preprocess the constraint graph.

2.1 Max Sum

The Max Sum algorithm belongs to the GDL (Generalised Distributive Law) framework [1], and has been shown to be a very useful technique for distributed constraint optimisation [5]. In more detail, Max Sum provides good quality approximate solutions to DCOPs (Distributed Constraint Optimisation Problems) requiring very low computation and communication. This is achieved by using message passing over a factor graph representation (see [8]) of the dependencies between agents' utilities in the global utility function. More specifically, a factor graph is a bipartite, undirected graph consisting of variable nodes and function nodes, where function nodes are connected to variable nodes they depend on.

However, while Max Sum has been proven to converge to an optimal solution on tree-structured factor graphs, it uses redundant computation in dynamic environments, and lacks optimality guarantees on cyclic graphs [14]. In the former case, Max Sum would have to be re-run after every change in the environment, and in the latter, very limited theoretical results relative to convergence and solution quality exist. As such, Fast Max Sum (FMS) [13] and Bounded Max Sum (BMS) [6] were presented in order to combat these problems: FMS reduces overheads incurred after a change in the environment, and BMS gives solutions with quality guarantees on cyclic graphs. Given this, we elaborate further on FMS and BMS in the rest of this section.

2.1.1 Fast Max Sum

Fast Max Sum (FMS) [13] provides two main improvements to Max Sum: (i) a reduction of message size, and (ii) a reduction of redundant communication and computation after a change in the graph. These improvements can be gained in certain scenarios, such as rescue scenarios, where each variable represents an agent, and each factor a target to rescue. In such cases, a variable's domain is the set of tasks that it must choose between, and each task's dependencies are the variables whose domain contains them.

Next, we explain how FMS reduces unneeded communication and computation after a change in the underlying graph: be it addition or removal of a function, or a variable. Put simply, a variable in FMS will only send a message in response to a received message if the values given in the received message are different to those it previously received from that factor. In order to do this, FMS adds storage requirements to Max Sum, requiring each variable to store their previous value, as well as the last message they received on each of their edges, compare to it, and update it, when messages are received. Then, a variable will only send a new message to a function if and only if its previous values for that node have changed. For a more detailed example of the execution of FMS, we refer the reader to [13].

Nevertheless, FMS can only produce provably optimal solutions in certain cases,[3] and is therefore not general enough for use in all environments. As such, we look at Bounded Max Sum, which provides approximate solutions but guarantees convergence and solution quality.

2.1.2 Bounded Max Sum

Bounded Max Sum (BMS) [6] produces bounded approximate solutions by eliminating cycles in the factor graph (see Section 3). More specifically, low-impact dependencies are found and removed by constructing a maximum spanning tree of the factor graph using the GHS algorithm [7] and then Max Sum is run on the resulting tree. Each node in the tree keeps track of which dependencies have been removed and, by using this information, can compute an approximation ratio of the optimal solution.

While BMS is not explicitly superstabilizing, we could make it so by introducing storage at each factor, in order to maintain information on the system state during recovery from a change in the environment. Despite this, however, BMS could still incur redundant computation and communication after such a change in the environment, as the GHS algorithm and Max Sum algorithms would need to be re-run. Now, as explained above, FMS would reduce overheads in the latter part of the algorithm, but not the former. As such, next, we detail the GHS algorithm.

2.2 GHS

The GHS algorithm [7] is a distributed technique to find the minimum spanning tree (MST) of any given weighted undirected graph, using only local communication and low computation at each node.

The basic premise of the GHS algorithm is that the nodes of a graph are formed into a number of graph fragments, which gradually join on their minimum-weight edges, in order to eventually form one large graph fragment, containing the MST of the graph. This is done through localised message passing. For a more detailed discussion of the operation of the algorithm, please refer to [7].

While the GHS algorithm is adequate for static problems, such as those BMS was designed for, when a change is made to the graph, the algorithm must be completely re-run. To avoid this, it is important to operate such an algorithm over a defined subset of the graph whenever a change occurs. To this end, we have developed a novel algorithm which we

[3] As with Max Sum, where the constraint graph contains no cycles.

present in Section 4. In order to do this, we formulate our problem in the next section.

3. PROBLEM FORMULATION

In this section, we formally describe the decentralised coordination problem that we address in this paper. We focus on a task allocation problem: specifically, an environment containing a number of agents, $\mathbf{A} = \{a_1, \ldots, a_{|A|}\}$, who must complete a number of tasks, $\mathbf{T} = \{t_1, \ldots, t_{|T|}\}$. The set of tasks which an agent $a \in \mathbf{A}$ can potentially complete is denoted $T_a \subseteq \mathbf{T}$. Similarly, the set of agents which can complete a task $t \in \mathbf{T}$ is denoted $A_t \subseteq \mathbf{A}$.

This problem can be conveniently represented with a factor graph (see [8]), which is a bipartite, undirected graph $\mathcal{FG} = \{\mathcal{N}, \mathcal{E}\}$, where $\mathcal{N}$ is the set of nodes, such that $\mathcal{N} = \mathcal{VN} \cup \mathcal{FN}$, where $\mathcal{VN}$ is a set of variable nodes, and $\mathcal{FN}$ is a set of function nodes. In addition, $\mathcal{E}$ is a set of edges, where each edge $e \in \mathcal{E}$ joins exactly one node in $\mathcal{VN}$ to exactly one node in $\mathcal{FN}$. An example factor graph formulation of our scenario is given in Figure 1.

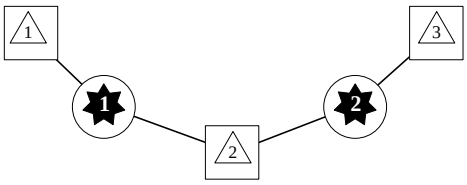


Figure 1: An example scenario containing 2 rescue agents (black stars) and 3 tasks (white triangles), formulated as a factor graph, with agents as variables (circles), and tasks as factors (squares).

In our problem, we have a set of variables, $\mathbf{x} = \{x_1, \ldots, x_m\}$, controlled by the set of agents $\mathbf{A}$. Each agent owns precisely one variable,[4] so we denote the variable belonging to agent $a \in \mathbf{A}$ as x_a. This variable represents the target of the agent: that is, the task that the agent must complete, and as such the domain of x_a is T_a.

Next, we define a set of functions, $\mathbf{F} = \{F_1, \ldots, F_n\}$, each representing a task $t \in \mathbf{T}$. A function $F_t(\mathbf{x}_t)$ is dependent on the set of variables which can potentially take the value t, or more formally, $\mathbf{x}_t = \{x_a | a \in A_t\}$. Thus, $F_t(\mathbf{x}_t)$ denotes the value for each possible assignment of the variables in $\mathbf{x}_t$.

Given this, we aim to find the state of each variable in $\mathbf{x}$ which maximises the sum of all functions in the environment (known as social welfare):

$$\mathbf{x}^* = \arg\max_{\mathbf{x}} \sum_{t \in \mathbf{T}} F_t(\mathbf{x}_t) \quad (1)$$

Specifically, the factor graph consists of a function node $FN \in \mathcal{FN}$ for each $F \in \mathbf{F}$ and a variable node $VN \in \mathcal{VN}$ for each $x \in \mathbf{x}$. We assume that each agent $a \in \mathbf{A}$ only has control over, and knowledge of, its own local variable x_a, and thus one variable node in the factor graph. The decision as to which agent computes for shared functions has no impact on the correctness of our approach, and as such any policy can be used to decide this (e.g. the agent with the lowest ID computes for shared functions).[5] Thus, we assume that each agent may compute any number of functions in $\mathbf{F}$, but that each function $F \in \mathbf{F}$ is computed by one agent $a \in \mathbf{A}$ only. Now, as mentioned earlier, each function $F \in \mathbf{F}$ represents one task in the task space, $t \in \mathbf{T}$. Hence, each function node in the factor graph represents one task, and so, each function node $F_t(\mathbf{x}_t)$ will be connected to the variable nodes representing the variables $\mathbf{x}_t$.

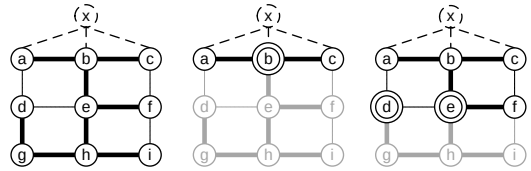
[4]This is not a limitation of our solution approach, but a feature of our reference domain.

4. IGHS

As previously mentioned, it is not ideal to completely recalculate the MST of an environment whenever the environment changes, as this is expensive in terms of communication and computation. Hence, to avoid such issues, upon certain types of graph change, we only find the spanning tree of a small part of the graph at a time. This allows us to reduce communication and computation overheads.

4.1 The Algorithm

The general idea of iGHS (iterative GHS) is to run GHS only on subgraphs of the whole problem. Specifically, given a spanning tree, the whole factor graph, and a node to add, we take a subgraph of the graph and run GHS on that in order to find a MST of this subgraph. We choose the subgraph by using a variable k, which defines the depth of nodes in the graph we consider, measured from the node to be added. This is illustrated in Figure 2, where part (a) shows the original spanning tree, with node x to be added, and parts (b) and (c) highlight the nodes which are included in subgraphs with $k = 1$, and $k = 2$, respectively.



(a) Original Spanning Tree. (b) $k = 1$ (c) $k = 2$

Figure 2: The effect of k on the size of the subgraph. Thick lines are spanning tree branches, thin lines are edges in the graph but not the spanning tree.

Finding the MST of the subgraph in order for the rest of the graph to remain an MST is a challenge. In more detail, if two or more nodes in the subgraph have spanning tree branches to nodes not in the subgraph (we call these *frontier nodes*: they are shown in Figure 2 as nodes with double outlines), then joining these two frontier nodes in the MST could cause a cycle in the spanning tree as a whole, thus invalidating the tree. We can see how this would occur by considering nodes d and e in Figure 2(c): both nodes have a spanning tree branch in the remainder of the graph, so making a spanning tree branch either on edge (a, d) or edge (d, e) would connect d and e, thus creating a cycle in the rest of the graph. In order to combat this, we identify the frontier nodes, and ensure that we only connect to *one* of these nodes, on *one* edge. We can then guarantee we have not introduced a cycle.

iGHS operates in two phases: (1) a flooding phase which determines which nodes and edges are in the subgraph to be

[5]An allocation that balances the computational load among agents might be desirable, but is beyond the scope of the current paper.

Algorithm 1 Phase 1 of the iGHS algorithm, at a node n.

Require: $possEdges = adj(n), lastCount = -\infty,$
$inEdges, exEdges = \emptyset$
1: **At starting node, given** k**:**
2: $lastCount = k$;
3: Send $Flood(k-1)$ to all $n' \in possEdges$
4: **On receipt of** $Flood(d)$ **on edge** j**:**
5: **if** $d > lastCount$ **then**
6: $lastCount = d; inEdges = inEdges \cup j;$
7: $possEdges = possEdges \setminus j; exEdges = exEdges \setminus j;$
8: **if** $d < 0$ **then** // `Node is not in the subgraph`
9: Send $External$ on edge j, then **halt**.
10: **else** // `Node is in the subgraph`
11: Send $Flood(d-1)$ on all $e \in \{possEdges \setminus j\}$
12: **if** $possEdges = \emptyset$ **then** Send $FloodAck$ on edge j, then **halt**.
13: **else** Put message on end of queue.
14: **On receipt of** $External$ **on edge** j
15: $exEdges = exEdges \cup j; possEdges = possEdges \setminus j$
16: **On receipt of** $FloodAck$ **on edge** j
17: $inEdges = inEdges \cup j; exEdges = exEdges \setminus j; possEdges = possEdges \setminus j;$

considered, and (2) a phase based on GHS, which establishes the MST, and adds the minimum weight frontier edge (i.e., edge joining to a frontier node) to the MST.

4.2 Phase 1

Algorithm 1 gives pseudocode for the flooding phase of our algorithm. In this phase, each node identifies which of its adjacent edges in the graph are within the subgraph ($inEdges$), and which are not ($exEdges$). This is done by initially adding all of a node's adjacent edges, $adj(n)$, into $possEdges$, which denotes that they could belong to the subgraph.

In more detail, the flooding phase begins with the node to be added informing its neighbours of the value of k. More specifically, this node sends $Flood(k-1)$ messages to all of its neighbours (line 3). Now, when a node n receives a $Flood(d)$ message, it will propagate further $Flood(d-1)$ messages along edges it is unsure of the status of (line 11), unless it receives a $Flood$ message on edge j containing a value less than 0. If this happens, the node sends an $External$ message on edge j (line 9), informing the node at the other end that j is in $exEdges$ (line 15).

Now, in order for a node to classify an edge into $inEdges$, the node must have received either a $Flood$ message (line 6) or a $FloodAck$ message on that edge (line 16). $FloodAck$ messages are sent when a node has classified all of its edges into $inEdges$ or $exEdges$ (line 12), and so, we can see that they are sent from the nodes that are furthest away from the new node, back toward the node to be added. The algorithm stops when the node to be added has received a $FloodAck$ message from each of its neighbours. Thus, when the node to be added has received $FloodAck$ messages on each of its edges, we know that every node in the subgraph is aware of which of its edges are in the subgraph, and which are not, and phase 1 of the algorithm is complete.

4.3 Phase 2

We give the pseudocode for phase 2 of iGHS in Algorithm 2. For brevity, we omit the sections (lines 3, 9, 14, 20, 35) of the algorithm that are repeated from GHS and focus only on the areas in which we have made changes.

Now, we can classify nodes into frontier nodes and non-frontier nodes: if, for a node n, $exEdges \neq \emptyset$ and the previous status of at least one of the edges in $exEdges$ is $BRANCH$ (i.e., it was a branch in the spanning tree predating this computation), then n is a frontier node.

Algorithm 2 Phase 2 of the iGHS algorithm, at a node n.

Require: $inEdges, exEdges, frontier$
1: **Procedure** `wakeup()`
2: $bestFrontierEdge = nil; bestFrontierWeight = \infty$
3: **if** $frontier = false$ **then** Proceed as GHS `wakeup()`
4: **On receipt of** $Connect(L)$ **or** $Test(L, F)$ **on edge** j
5: **if** $frontier = true$ **then** // j `is a frontier edge.`
6: $SE(j) = REJECTED$
7: Send $Frontier$ on j
8: **else** // j `is an edge in the subgraph.`
9: Proceed as GHS $Connect(L)$ or $Test(L, F)$
10: **On receipt of** $InstConnect(L)$ **on edge** j
11: $SE(j) = BRANCH$
12: **Procedure** `test()`
13: **if** there are adjacent edges in state BASIC and not in $exEdges$ or $frontierEdges$ **then**
14: Proceed as GHS `test()` procedure.
15: **On receipt of** $Frontier$ **on edge** j
16: $SE(j) = REJECTED;$
17: $frontierEdges = frontierEdges \cup j$
18: $bestFrontierEdge = \min_{e \in frontierEdges} w(e)$
19: $bestFrontierWt =$ weight of $bestFrontierEdge$
20: **if** $FN =$ nil **then** Proceed as last received GHS $Connect$
21: **else** `test()`
22: **Procedure** `report()`
23: **if** $findCount = 0$ and $testEdge = nil$ **then**
24: $SN = FOUND$
25: Send $Report(bestWt, bestFrontierWt)$ on $inBranch$
26: **On receipt of** $Report(w, fw)$ **on edge** j
27: **if** $j \neq inBranch$ **then**
28: $findCount = findCount - 1$
29: **if** $w < bestWt$ **then** $bestWt = w; bestEdge = j;$
30: **if** $fw < bestFrontierWt$ **then** $bestFrontierWt = fw; bestFrontierEdge = j;$
31: `report()`
32: **else if** $fw > bestFrontierWt$ and $w = \infty$ **then**
33: `ChangeFrontierRoot()`
34: **else**
35: Proceed as GHS $Report(w)$
36: `ChangeFrontierRoot()`
37: **if** $SE(bestFrontierEdge) = BRANCH$ **then**
38: Send $ChangeFrontierRoot$ on $bestFrontierEdge$
39: **else**
40: $SE(bestFrontierEdge) = BRANCH$
41: Send $InstConnect(L)$ on $bestFrontierEdge$

Given this, we now go into more detail on Algorithm 2. When a node's status is $SLEEPING$, and it receives a message, it runs the `wakeup()` procedure (lines 1–3), initialising its variables, and tries to connect to one of its neighbours, if it is *not* a frontier node. When a node receives a connection attempt from its neighbour on edge j (line 4), the algorithm will proceed in one of two ways: if the node is a frontier node, then it marks j as not in the spanning tree and sends a $Frontier$ message along j (lines 5–7), and if the node is not a frontier node, then it proceeds as it would in GHS (line 8). When a node receives a $Frontier$ message along j (line 15), it marks j as not in the spanning tree (line 16), adds j to its list of frontier edges (line 17), and updates its best frontier edge and weight values (lines 18–19), before carrying on as if edge j did not exist (lines 20–21).

Now, when a node has connected on its minimum-weight edge, it tries to connect on each of its other edges, in order of weight (see lines 12–14). However, if a node has no more unclassified edges left, then it runs the procedure `report()` (line 22), and informs its parent in the tree of the lowest weight frontier edge it has (lines 23–25). When the parent receives this report (line 26), it decides which of its neighbours has the lowest-weight frontier edge, and informs its own parent (lines 27–31). However, if the report message is received by the root of the graph (line 32), then the receiving node can be sure that its best frontier edge leads to the spanning tree's best frontier edge. As such, the receiving node sends instruction to connect along the best frontier edge (line 33). Finally, if a node receives an instruction to

join on its best frontier edge (line 36), it determines whether its best frontier edge points further down the tree (line 37), or is the edge to connect on (line 39), and either passes the message on (line 38), or connects on the best frontier edge (lines 41, 10–11).

4.4 Properties of iGHS

Having presented our algorithm, we now show that it is correct, and superstabilizing.

Correct. In order to prove the correctness of iGHS, we must show first, that it cannot create a cycle in the overall spanning tree of the graph, and second, that it will make a minimum spanning tree in the subgraph.

First, given that GHS is correct [7], and cannot create cycles, we know that iGHS will not make a cycle in the subgraph it runs on. Thus, it is sufficient for us to show that, in joining our MST to the spanning tree of the rest of the graph, we cannot create a cycle. This is because if a node is connected to a tree-structured graph on a single edge, then it is impossible for that node to have created a cycle in the overall graph. Now, as we are sure that we make a spanning tree across all edges but those that connect to frontier nodes, and we connect that tree to the rest of the graph on a single edge, we can guarantee that our algorithm can, indeed, not make a cycle in the graph overall. Second, we can guarantee that the spanning tree produced by running iGHS on the subgraph is minimum due to the properties of the GHS algorithm.

Superstabilizing. In order to show that iGHS is superstabilizing, we define the following predicates: *legitimacy*: when the algorithm is complete, the spanning tree produced contains no cycles, *passage*: at no time during recovery from a perturbation, are any cycles inadvertently formed. We can say that iGHS is superstabilizing with respect to these predicates: first, because GHS does not form any cycles in its operation, and second, because iGHS is correct.

Now that we have ascertained these properties, we present Bounded Fast Max Sum, which combines iGHS and Fast Max Sum in order to solve distributed constraint optimisation problems on arbitrary graphs.

5. BOUNDED FAST MAX SUM

Here, we introduce the Bounded Fast Max Sum algorithm, which consists of three main procedures running sequentially after a node is added to the graph:

1. **Efficient, superstabilizing spanning tree generation**: using iGHS (Section 4) to re-calculate the maximum spanning tree around the added node.

2. **Fast Max Sum**, to calculate the optimal assignment of variables in the spanning tree. Only the nodes who's utility changes as a result of the addition need resend their Max Sum messages through the tree: if their change in utility does not change, then the decision rules in Fast Max Sum will ensure messages are not transmitted unnecessarily.

3. **WSUM and SOLUTION propagation**, in order to calculate the quality of the solution found.

Now, while adding a node to the graph is always handled in the same manner, this is not the case for removing nodes. In more detail, we must consider both circumstances under which a (function or variable) node can be removed from the graph: first, physical disconnection, such as when a rescue agent is malfunctioning or disappears, and second, when a task has been completed. To support the first case, we introduce contingency plans. Each time a node n is certain of the status of each of its incident edges (i.e. when they have all been marked BRANCH or REJECTED), it chooses its lowest weighted, and hence, most important, BRANCH edge and informs the node at the other end (n_2) that n_2 is n's contingency. Then, if node n is removed from the graph, n_2 will instigate the iGHS algorithm, and maximise the spanning tree around itself. The second case (a task being completed) is slightly different, in that the factor node for that task will not be physically removed from the graph. Instead, the factor node acts as a 'handler' for iGHS, by setting the weight of each of its incident edges to 0 (as their utility would be 0 anyway), and instigating execution of iGHS.

In terms of how many nodes can be added and/or removed simultaneously, the iGHS algorithm imposes a restriction on both. More specifically, as iGHS has a set of values at each node which are used in the spanning tree formation, we must ensure that no one node is involved in more than one instance of iGHS at a time, so that these values are not being overwritten by multiple iGHS instances. Hence, we can say that if two or more nodes are to be added to and/or removed from the graph simultaneously, they must be of distance at least $2k+1$ away from each other, to avoid overlap in iGHS instances. We next detail the execution of BFMS.

5.1 The Algorithm

As in BMS, each dependency link e_{ij} in the factor graph $\mathcal{FG}$ is weighted as below:

$$w_{ij} = \max_{\mathbf{x}_i \setminus j} \left[\max_{x_j} F_i(\mathbf{x}_i) - \min_{x_j} F_i(\mathbf{x}_i) \right] \tag{2}$$

This weight represents the maximum impact that variable x_i can have over function F_j. Thus, by not including link e_{ij} in the spanning tree, we say that the distance between our solution and the optimal is at most w_{ij}.

Now, once each link has been weighted, our iGHS algorithm is started (see Section 4 for more details). The outcome of the convergence of this algorithm is, initially, a maximum spanning tree.[6] Following this, after every addition to the graph, the spanning tree around the change is maximised, thus iteratively improving a section of the graph.

Next, we run Fast Max Sum on the resulting spanning tree, beginning at the leaves, and propagating up the tree, with each node waiting to receive messages from all of its children before calculating and sending its next message. These messages then propagate back down the tree from the root to the leaves in a similar manner, at which point the algorithm converges, and a variable assignment $\tilde{\mathbf{x}}$ can be computed. Functions with removed dependencies are evaluated by minimising over the removed dependencies, as in BMS.

Finally, the algorithm enters the WSUM and SOLUTION propagation phase, which is the same as that of BMS. More

[6]The maximum spanning tree can be obtained by using iGHS and negating edge weights

specifically, once the leaves have received the Fast Max Sum messages, they can compose new WSUM and SOLUTION messages. If the leaf node is a function, WSUM is the sum of the weights of its removed dependencies, and SOLUTION is $F_t(\tilde{\mathbf{x}}_t)$. If the leaf is a variable, the WSUM and SOLUTION messages are empty. When a node receives WSUM and SOLUTION from all its children, it can process them according to whether it is a function node or a variable node. If the node is a variable node, these messages are the sum of messages from its children. If the node is a function node, the messages are the sum of messages from its children, plus its own values for the removed edge weights (for WSUM) and $F_i(\tilde{\mathbf{x}}_i)$ (for SOLUTION). Once these messages reach the root, the root propagates them back down, so every node is aware of the total weight removed, W, and the solution value, $\tilde{V} = \sum_{t \in T} F_t(\tilde{\mathbf{x}}_t)$.

Now the agents have all information to compute the approximation ratio, as follows: $\rho(FG) = 1+(\tilde{V}^m+W-\tilde{V})/\tilde{V}$ where FG is the factor graph the algorithm has been run on, $\tilde{V}^m$ is the approximate solution value, and $\tilde{V}$ is the actual solution value.Now, as the approximation ratio tends to 1, this indicates improvement in the solution quality guarantees, because this indicates that the total weight of removed edges is small. Thus, in order to help this, we wish to keep the value of W as low as possible, by only removing low-weight edges (i.e., edges that have low bearing on the overall solution). We can see that the value of k given to iGHS has a bearing on the approximation ratio, too — higher values of k optimise larger sections of the graph. This means iGHS is more likely to remove the lowest weight combination of edges.

5.2 Properties of BFMS

In order to verify that that BFMS is superstabilizing, we must first show that FMS is superstabilizing. We do this subject to the following predicates: *legitimacy*: $U(\mathbf{x}) = \max_{\mathbf{x}} \sum_{t \in \mathbf{T}} F_t(\mathbf{x}_t)$, where $U(\mathbf{x})$ is the total utility of assignment $\mathbf{x}$, and *passage*: the previous assignment of variables is maintained until a new solution has been computed.

PROPOSITION 1. *Fast Max Sum is superstabilizing on tree structured graphs, because it is self-stabilizing on tree structured graphs, and, during stabilization after a change in the graph, the previous assignment of variables is maintained until a new solution has been computed.*

PROOF. First, FMS is an extension to Max Sum, which is proven to converge to an optimal solution on tree structured graphs. Second, when a change occurs in the graph, FMS is run again, and therefore, provided the change did not introduce a cycle into the graph, FMS is guaranteed to converge to the optimal again, reaching a legitimate state within a finite amount of time. This is because FMS does not change the messages sent, it just stops duplicate messages being sent when values at some nodes have not changed as a result of the graph change. FMS is superstabilizing because it has storage at each variable node in order to maintain a previous assignment during recalculation, and so, the passage predicate always holds. □

Given this, BFMS is also superstabilizing, because FMS and iGHS (see Section 4) are. As BFMS combines these two algorithms, we can deduce that BFMS is superstabilizing.

Next, we empirically evaluate BFMS, and compare it to BMS, in order to show the improvements BFMS gives.

6. EMPIRICAL EVALUATION

In order to empirically evaluate Bounded Fast Max Sum, we ran two types of experiment, intended to measure our performance in terms of approximation quality, robustness and utility gained. We compare Bounded Fast Max Sum to Bounded Max Sum (BMS), and a greedy version of Bounded Fast Max Sum (G-BFMS), in which an added node will connect to the rest of the spanning tree on its best-weight edge.

The first experiment intends to show that our algorithm is robust to changes in the graph. In more detail, we compared BFMS with $k = 2$, $k = 3$ and $k = 4$ to BMS, and G-BFMS. More specifically, we ran these algorithms on a series of 50 randomly generated graphs, where, initially, $|A| = 5$ and $|T| = 5$. In addition, we compared results found on graphs with task nodes of degree $\delta_t = 3$, to those with agent nodes of degree $\delta_a = 3$. We used a random look-up table, drawn from a uniform distribution, as the utility function of each task in order to evaluate our algorithm in a general setting. Given this, we ran experiments where we added agents, high-weighted tasks (where, for all values, $u_t(a) \in [0.5, 1]$), and low-weighted tasks (where $u_t(a) \in [0, 0.5]$) in order to demonstrate the impact that k has on the quality of the approximation in these situations. Now, for each experiment, we first ran one of the algorithms, and alternated adding new agents and tasks, one at a time,[7] to the environment. We repeated this process 10 times, recording a number of values after each algorithm run, and calculating the mean of each value at each step, with 95% confidence intervals (which we have plotted on our graphs). In addition, we ran experiments where we added only agents to the environment, to see if adding only one type of node would show a different trend to alternating types. During these experiments, we recorded a number of values.[8] Firstly, we recorded two values from the preprocessing phase of the algorithms: namely, the mean total size of preprocessing messages sent (PTNS), and mean total preprocessing storage used (TSU). The smaller these values are, the better. In addition, we recorded some values from the message passing phase of the algorithms: mean computation units used at each node (MCU), mean total size of messages sent, in bytes (TSS), and the mean approximation ratio obtained (AR). The values of MCU and TSS should, preferably, be small, and the AR should be as close to 1 as is possible.

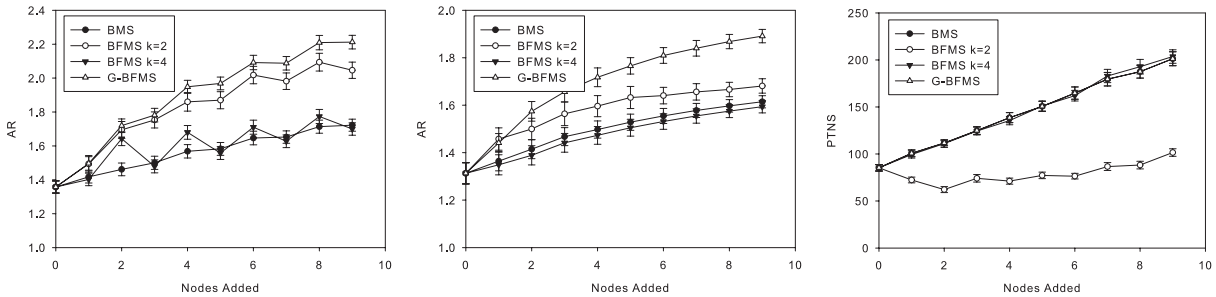
Given that FMS has been shown to outperform Max Sum in terms of MCU, TNS and TSS [13], we hypothesise:

Hypothesis: Bounded Fast Max Sum has lower MCU, and TSS than Bounded Max Sum. In addition, Bounded Fast Max Sum has lower PTNS than Bounded Max Sum.

We found from our experiments that varying the degree of tasks and agents had no real effect on the performance of our algorithm. In addition, we found that using high or low utility values of added tasks made little difference to the results either, and for this reason, we present here the results for high-valued utility functions, and $\delta_a = 3$. We can see from Figure 3(a) that alternating adding agents and tasks leads to faster deterioration in the approximation ratio when tasks

[7] As mentioned in Section 5, iGHS is only guaranteed to work if simultaneously added nodes are at least $2k+1$ nodes apart. The evaluation with multiple nodes is beyond the scope of this paper.

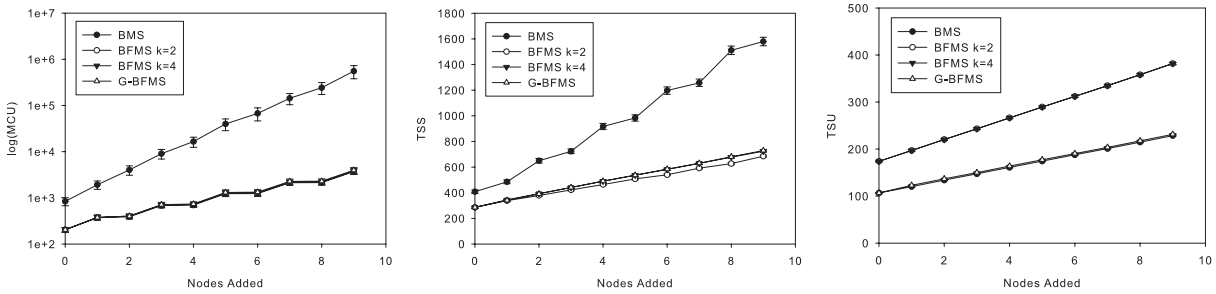
[8] Some of these are typical measures used in the DCOP community [10], approximation ratio comes from BMS [6].



(a) Mean Approximation Ratio: alternating.

(b) Mean Approximation Ratio: adding only agents.

(c) Mean preprocessing messages sent.



(d) Mean computation units used.

(e) Mean total size of messages sent.

(f) Mean total storage used.

Figure 3: Experiment 1 results.

are added. This is because functions have more impact on the approximation ratio than extra variables, and thus will degrade it further when added. Given this, when we only add agents (Figure 3(b)), we see a plateau in BFMS for both values of k, after adding around 5 tasks, where G-BFMS's approximation ratio continues to degrade. Figures 3(a), (b) and (c) show the effect that the value of k has on preprocessing messages sent and approximation quality: whilst $k = 2$ requires 28–53% fewer preprocessing messages than BMS, it produces approximation ratios within 81–94% of BMS. Increasing k to 3 gains approximation ratios within 88–100% of BMS, but reduces the saving in preprocessing messages sent to only 1–7% of BMS. Hence, it can be seen that higher values of k do achieve better approximation ratios, but at the expense of sending more preprocessing messages. Now, Figure 3(d) and (e) show the marked reduction in computation and message size gained by the use of FMS: up to a maximum of 99% over BMS. Finally, Figure 3(f) shows the extra storage at each node needed by iGHS, which, whilst linear in nature, is higher than that of BMS.

It can be seen in Figures 3(c) and (f) that G-BFMS uses very little additional storage and preprocessing messages after a change in the graph. This is because the decision made is entirely local to the node to be added, and, as such, no storage is needed, and only 1 message needs to be sent to confirm the chosen edge forming part of the spanning tree.

Next, our second experiment intends to evaluate the performance of our algorithm in a real scenario, and, as such, we used our own flooding simulator (based on that used in [13]) to compare utility gleaned by BFMS (using $k = 2$ and $k = 3$), G-BFMS and BMS in comparison to using BMS with complete information about all tasks to appear (denoted BMS-OPT). To do this, we set A such that $|A| = 10$ and varied the starting set of tasks to be completed, T, in increments of 5, such that $|T| \in \{0, 5 \cdots, 30\}$. Each task was given a deadline d_t, and a workload w_t, which indicate when the task will expire, and how long it will take to complete, respectively. We also generated a list of 10 tasks which were added to the environment, one at a time, every $|T|$ timesteps: thus, when we say BMS-OPT had full information, we mean that BMS-OPT was given the set T and the complete set of tasks that would be added over time at the start of the simulation. We randomly generated 50 instances of agent and task positions for each amount of tasks, drew the deadline of each task from a uniform distribution $d_t \in U(0, 10 \times |T|)$ and drew the workload from a uniform distribution as $w_t \in U(0, \frac{10 \times |T|}{2})$. We ran the three algorithms on each of the 50 instances, over $10 \times |T|$ time steps, running each algorithm at each time step. We show the mean total number of tasks completed by each algorithm in Figure 4, along with 95% confidence intervals.

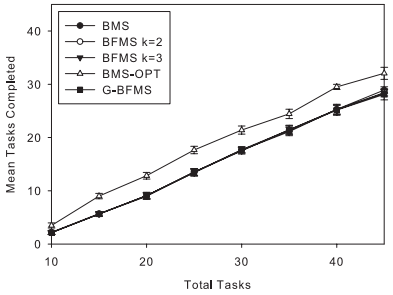


Figure 4: Mean total tasks completed.

We can see from Figure 4 that BFMS with $k = 2$ and $k = 3$ completes within 96–100% of the tasks as BMS, and that

performance of BFMS with $k = 3$ is very similar to that of BMS. In addition, we found that BFMS took a much smaller amount of time to converge to a solution at every time step, compared to BMS and OPTBMS which took far longer.

7. RELATED WORK

Using the GHS algorithm [7] as a starting point, we looked into work similar to iGHS in order to locally optimise a spanning tree, without affecting the whole graph. We found some work into distributed dynamic MST algorithms: for example, Cheng et al.'s Dynamic MST protocol [2], which finds the MST of the entire graph. However, these algorithms do not suit our purpose, for we wish to reduce computation and communication done by the algorithm overall, and hence, the amount of spanning tree changed must be limited.

Given this, some work closer to ours is that of localised spanning tree calculation, which is of particular interest in wireless ad-hoc networks [9]. In such domains, low power consumption and memory limit the scope of potential algorithms to those that are localised and power-efficient. In addition, localised minimum spanning tree algorithms such as that of [9] are often constrained by bounds on node degree (i.e., the number of outgoing links from each node), and minimising the length of each links between nodes, in order to conserve precious resources. Fortunately, our domain is not so limited in terms of node degree (in fact, large node degrees would help to increase solution quality), or link length: we can assume that, if links are present in the graph of the scenario, then they are available to be chosen in the MST.

The work of Li et al. is similar to ours, in that their Incident MST and RNG Graph (IMRNG)[9] method finds the MST of nodes within two 'hops' from each node in the graph. However, the mechanism is, in a sense, locally centralised, in that each node finds out the weights of its neighbouring links through message passing, before calculating the MST locally, and informing its neighbours of the result. Our algorithm is not centralised in this way, as iGHS and FMS are both entirely distributed, in that removing a node will not cause either algorithm to collapse. In addition, the mechanism is heavily constrained in order to be compatible with ad-hoc networks, forcing node degree to be at most 6, and fixing the hop count to 2, whereas we allow k (our measure of hop count) and node degree to take any value.

8. CONCLUSIONS AND FUTURE WORK

We have presented an efficient, dynamic, superstabilizing algorithm for distributed constraint optimisation, which is able to provide approximate solutions with quality guarantees, whilst reducing redundant computation and communication in dynamic environments. Our main directions for future work are to consider how the algorithm could adapt to areas of varying communication, and to enable any number of nodes to be added and/or removed simultaneously. More specifically, we are interested to find out if dynamically varying the value of k in response to varied available bandwidth could optimise our solution quality in environments where communication capabilities can vary. Second, our algorithm is constrained in the number of nodes that can be added or removed at any one time, not allowing at simultaneous addition and/or removal of nodes which are less than $2k + 1$ from each other. Thus, we will endeavour to reduce, or possibly remove, this limit.

[9]RNG: Relative Neighbourhood Graph

Acknowledgements

This research was undertaken as part of the ALADDIN (Autonomous Learning Agents for Decentralised Data and Information Systems) Project and is jointly funded by a BAE Systems and EPSRC (Engineering and Physical Research Council) strategic partnership (EP/C 548051/1).

References

[1] S. M. Aji and R. J. McEliece. The generalized distributive law. *IEEE Transactions on Information Theory*, 46(2):325–343, 2000.

[2] C. Cheng, I. Cimet, and S. Kumar. A protocol to maintain a minimum spanning tree in a dynamic topology. *SIGCOMM Computer Communication Review*, 18(4): 330–337, 1988.

[3] E W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11): 643–644, 1974.

[4] S Dolev and T Herman. Superstabilizing protocols for dynamic distributed systems. *Chicago Journal of Theoretical Computer Science*, 1997(4):1–40, 1997.

[5] A. Farinelli, A. Rogers, A. Petcu, and N. R. Jennings. Decentralised coordination of low-power embedded devices using the max-sum algorithm. In *Proc. AAMAS-08*, pages 639–646, 2008.

[6] A. Farinelli, A. Rogers, and N. Jennings. Bounded approximate decentralised coordination using the max-sum algorithm. In *DCR Workshop, IJCAI-09*, pages 46–59, July 2009.

[7] R. G. Gallager, P. A. Humblet, and P. M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Trans. TOPLAS*, 5(1):66–77, 1983.

[8] F. R. Kschischang, B. J. Frey, and H. A. Loeliger. Factor graphs and the sum-product algorithm. *IEEE Transactions on information theory*, 47(2):498–519, 2001.

[9] X. Li, Y. Weng, P. Wan, W. Song, and O. Frieder. Localized low-weight graph and its applications in wireless ad hoc networks. In *INFOCOM-04*, volume 1, pages 431–442, March 2004.

[10] P. J. Modi, W. S. Shen, M. Tambe, and M. Yokoo. ADOPT: Asynchronous distributed constraint optimization with quality guarantees. *Artificial Intelligence*, 161(1-2):149–180, 2006.

[11] A. Petcu and B. Faltings. A scalable method for multiagent constraint optimization. In *Proc. IJCAI-05*, volume 19, pages 266–271. AAAI Press, 2005.

[12] A. Petcu and B. Faltings. S-DPOP: Superstabilizing, Fault-containing Multiagent Combinatorial Optimization. In *Proc. AAAI-05*, pages 449–454, 2005.

[13] S. D. Ramchurn, A. Farinelli, K. S. Macarthur, M. Polukarov, and N Jennings. Decentralised Coordination in RobocupRescue. (to appear) *The Computer Journal*, 2010.

[14] Y. Weiss and W. T. Freeman. On the optimality of solutions of the max-product belief propagation algorithm in arbitrary graphs. *IEEE Transactions on Information Theory*, 47(2):723–735, 2001.