A Distributed Anytime Algorithm for Dynamic Task Allocation in Multi-Agent Systems

Kathryn S. Macarthur, Ruben Stranders, Sarvapali D. Ramchurn and **Nicholas R. Jennings**
{*ksm08r,rs2,sdr,nrj*}*@ecs.soton.ac.uk*
School of Electronics and Computer Science, University of Southampton, UK

Abstract

We introduce a novel distributed algorithm for multi-agent task allocation problems where the sets of tasks and agents constantly change over time. We build on an existing anytime algorithm (fast-max-sum), and give it significant new capabilities: namely, an online pruning procedure that simplifies the problem, and a branch-and-bound technique that reduces the search space. This allows us to scale to problems with hundreds of tasks and agents. We empirically evaluate our algorithm against established benchmarks and find that, even in such large environments, a solution is found up to 31% faster, and with up to 23% more utility, than state-of-the-art approximation algorithms. In addition, our algorithm sends up to 30% fewer messages than current approaches when the set of agents or tasks changes.

Introduction

Multi-agent task allocation is an important and challenging problem, which involves deciding how to assign a set of agents to a set of tasks, both of which may change over time (i.e., it is a dynamic environment). Moreover, it is often necessary for heterogeneous agents to form teams (known as *coalitions*) to complete certain tasks in the environment. In coalitions, agents can often complete tasks more efficiently or accurately, as a result of their synergistic abilities. Examples of task allocation problems include the allocation of sensing tasks to robots (Zheng and Koenig 2008), and rescue tasks to ambulances (Shehory and Kraus 1998). However, existing approaches are typically centralised (Shehory and Kraus 1998; Zheng and Koenig 2008) and do not consider that the set of tasks and agents may change over time. Hence, they introduce a single point of failure and, if the task allocation problem changes due to the arrival of a new agent or task, need to recompute solutions from scratch. A number of decentralised, dynamic task allocation algorithms have been developed: namely, LA-DCOP (Scerri et al. 2005), SDPOP (Petcu and Faltings 2005), OPGA (Chapman et al. 2009), and fast-max-sum (Ramchurn et al. 2010). However, LA-DCOP does not model synergies between agents in coalitions: it simply assumes that the value of a team of agents is additive. Additionally, SDPOP can send exponentially large messages, and so doesn't scale well in the size of the task allocation problem, and, whilst OPGA, which introduces dynamism to and builds on the Distributed Stochastic Algorithm (DSA) (Fitzpatrick and Meertens 2003), is fast and uses few messages, it can get stuck in local minima. In contrast, Ramchurn et al.'s fast-max-sum, proposed as a specialisation of the standard max-sum (Aji and McEliece 2000) for task allocation, reduces the communicational and computational impact of dynamism in the environment without getting caught in local minima. Thus, we use fast-max-sum as the underlying basis for our algorithm.

Against this background, in this paper we present branch-and-bound fast-max-sum (BnB FMS), a decentralised, dynamic task allocation algorithm. BnB FMS consists of two interleaved sub-algorithms. The first attempts to reduce the number of tasks that an individual agent considers, while the second reduces the number of coalitions that need to be evaluated to find the optimal coalition for a task. Our approach yields significant reductions in both run-time and communication, thereby increasing real-world applicability.

In more detail, in this paper we advance the state-of-the-art in the following ways: first, we present a novel, online domain pruning algorithm specifically tailored to dynamic task allocation environments to reduce the number of potential solutions that need to be considered. Second, we present a new technique involving branch-and-bound search trees to reduce the execution time of fast-max-sum. Third, we empirically show that BnB FMS finds a solution up to 31% faster and uses up to 30% less communication than fast-max-sum in dynamic environments, whilst obtaining up to 23% higher solution quality than other state-of-the-art approximation algorithms.

The remainder of this paper is structured as follows: first, we formulate our task allocation problem. Then, we give some background on fast-max-sum. Next, we present BnB FMS, which we empirically evaluate in the following section. Finally, we conclude.

Problem Description

We denote the set of agents as $\mathcal{A} = \{a_1, \ldots, a_{|\mathcal{A}|}\}$, and the set of tasks as $\mathcal{T} = \{t_1, \ldots, t_{|\mathcal{T}|}\}$. An agent $a_i \in \mathcal{A}$ can perform any one task in $\mathcal{T}_i \subseteq \mathcal{T}$, the set of tasks a_i can perform. Similarly, we denote the set of agents that can perform task t_j as $\mathcal{A}_j$.

Now, the value of coalition $C_j \in 2^{\mathcal{A}_j}$ performing task t_j is denoted $V(C_j, t_j) \in \mathbb{R}^+$. As we said earlier, this value is not necessarily additive in the capabilities of the agents in C_j. Since task t_j cannot be performed with no agents assigned to it, $V(\emptyset, t_j) = 0, \forall t_j \in \mathcal{T}$. Moreover, we define the *contribution* of a_i to t_j as $\delta_j(a_i, C_j)$, if coalition C_j performs t_j, where $\delta_j(a_i, C_j) = V(C_j, t_j) - V(C_j \setminus \{a_i\}, t_j)$. Given this, and assuming that task utilities are independent, our objective is to find the coalition structure $\mathcal{S}^* = \{C_1^*, \ldots, C_{|\mathcal{T}|}^*\}$ which maximises the global utility:

$$\mathcal{S}^* = \arg\max_{\mathcal{S}} \sum_{C_j \in \mathcal{S}} V(C_j, t_j) \tag{1}$$

where $\mathcal{S}$ denotes the set of all coalition structures, and subject to the constraint that no coalition overlaps (i.e., no agent does more than one task at a time):

$$C_j \cap C_k = \emptyset, \forall C_j, C_k \in \mathcal{S}, j \neq k \tag{2}$$

The state space that Equation 1 covers is potentially very large, i.e. $\prod_{i=1}^{|\mathcal{A}|} |\mathcal{T}_i|$. However we can exploit the fact that $\mathcal{T}_i \neq \mathcal{T}$, i.e., the problem is factorisable, and ensure that there is no dependence on a central entity by decentralising computation. To this end, we extend the fast-max-sum algorithm, described in the next section.

Fast-Max-Sum

To find solutions to our task allocation problem, we build upon and extend fast-max-sum (FMS) (Ramchurn et al. 2010), which extends a particular algorithm from the Generalised Distributive Law (GDL) family (Aji and McEliece 2000), known as max-sum. FMS exploits a particular formulation of task allocation environments to greatly reduce the message size and computation required when applying max-sum in dynamic environments.

In more detail, we can model Equation 1 as a factor graph (Kschischang, Frey, and Loeliger 2001): a bipartite, undirected graph, $\mathcal{G} = \langle \mathcal{N}, \mathcal{E} \rangle$. A factor graph consists of a set of nodes $\mathcal{N} = \mathcal{F} \cup \mathcal{X}$, which consists of two types of nodes: function nodes (factors), $\mathcal{F}$, and variable nodes, $\mathcal{X}$. In addition, the factor graph has a set of edges $\mathcal{E}$, which represent which variables are dependencies of which functions. The task allocation problem can be represented as a factor graph, by modelling an agent a_i as a variable x_i, which represents the task that the agent will perform. Thus, x_i takes values in $\mathcal{T}_i$. We can restrict how much of each variable's domain is represented within functions: since $V(C_j, t_j) = 0$ where $C_j = \emptyset$, we need only represent an agent being assigned to a specific task, and not being assigned to that task. This is specialised to task allocation, unlike max-sum, where each function is computed for each possible variable state. Thus, each coalition value function, $V(C_j, t_j)$, is represented by a function $f_j(\mathbf{x}_j)$, which operates over domain $\mathbf{x}_j$, where $\mathbf{x}_j$ represents which agents are assigned to t_j: for example, $\{(x_1 = t_j), (x_2 \neq t_j) \ldots\}$, where $(x_i = t_j) \in \mathbf{x}_j$ is equivalent to $a_i \in C_j$. Therefore, $V(C_j, t_j) = f_j(\mathbf{x}_j)$. We do not need to explicitly represent a variable's state if it is not assigned to t_j, since $\delta_j(a_i, C_j) = 0$ where $a_i \notin C_j$: task t_j is not concerned with which agent is assigned to which task, only which agents are assigned to t_j. It should be noted that we assume that the computation of function nodes has been distributed amongst the agents.[1]

Thus, in factor graph notation, the objective function given in Equation 1 is as follows: to find the variable assignment $\mathbf{x}^*$ which maximises the global utility:

$$\mathbf{x}^* = \arg\max_{\mathbf{x}} \sum_{f_j \in \mathcal{F}} f_j(\mathbf{x}_j) \tag{3}$$

Now that we have formulated the task allocation problem as a factor graph, we can apply FMS, which solves Equation 3 in a decentralised fashion by exchanging messages between the nodes in $\mathcal{N}$ — we extend these messages later. The message sent from function f_j to variable x_i is as follows:

$$r_{j \to i}(x_i) \text{ for } x_i = t_j \text{ and any } x_i \neq t_j \text{ where:}$$

$$r_{j \to i}(x_i) = \max_{\mathbf{x}_j \setminus x_i} \left[f_j(\mathbf{x}_j) + \sum_{x_k \in \mathbf{x}_j \setminus x_i} q_{k \to j}(x_k) \right] \tag{4}$$

This message consists of two values: where x_i is assigned to t_j, and where it is not. When a variable receives such a message from a function, it responds by sending an updated $q_{i \to j}$ message to each of its neighbours, as follows:

$$q_{i \to j}(x_i = t_j) = q \text{ and } q_{i \to j}(x_i \neq t_j) = q' \text{ where:}$$

$$q = \sum_{f_k' \in \mathcal{F}_i \setminus j} r_{k \to i}(x_i \neq -t_k)$$

$$q' = \max_{t_b \neq t_j} \left[r_{b \to i}(x_i = t_b) + \sum_{f_k \in \mathcal{F}_i \setminus f_b, f_j} r_{k \to i}(x_i \neq t_k) \right] \tag{5}$$

where $\mathcal{F}_i = \{f_j | t_j \in \mathcal{T}_i\}$ and $t_b \in \mathcal{T}_i \setminus t_j$. Again, the function only considers the utility gained for $x_i = t_j$ and $x_i \neq t_j$. These messages continue to be passed, until their values no longer change (i.e., the algorithm has converged).[2] Nevertheless, to choose its value, the variable computes its marginal function, $z_i(x_i = t_j)$ as follows:

$$z_i(x_i = t_j) = \left(r_{j \to i}(x_i = t_j) + \sum_{f_k \in \mathcal{F}_i \setminus f_j} r_{k \to i}(x_i \neq t_k) \right) \tag{6}$$

In so doing, we obtain the total utility for all states of the variable. Then, the variable's value is found with $\arg\max_{x_i} z_i(x_i)$.

By only considering $x_i = t_k$ and $x_i \neq t_k$, FMS reduces the communicated domain size of each variable from d to 2, compared to a naïve implementation of max-sum. Hence, the computational complexity of a factor with n variables of domain size d in FMS is $O(2^n)$, in contrast to $O(d^n)$ required by naïvely applying max-sum to the same environment. However, the overall worst case computational complexity of FMS is $O(|\mathcal{F}| \times 2^{d_{\max}})$, where $d_{\max}$ is the maximum arity of a function — i.e., the maximum number of agents that could complete a task in the environment. Hence, evaluating each factor remains the key bottleneck of FMS.

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

[2] Convergence is only guaranteed on acyclic factor graphs. However, strong empirical results (Farinelli et al. 2008) exist that show max-sum finds good quality solutions on cyclic graphs.

Branch-and-Bound Fast-Max-Sum

In order to combat this key bottleneck of FMS and enable it to scale effectively in the number of agents and tasks, it is crucial to use techniques that reduce the size of the state space explored: by reducing the number of tasks and coalitions considered, whilst ensuring that such techniques remain both effective and efficient as the environment changes. In doing this, we create an efficient, decentralised task allocation algorithm for use in dynamic environments.

Our approach is similar to that of BnB MS (Stranders et al. 2009). However, the key difference is that we explicitly cater for environments that change over time, whereas BnB MS would need to be completely re-run after every change in the environment, as we show later. Moreover, BnB MS is based on max-sum, and thus incurs additional communication and computation needed to search through full variable domains, as opposed to exploiting the ability to use binary values in task allocation environments, as we showed earlier. Thus, in this section, we present branch-and-bound fast-max-sum (BnB FMS), which addresses the shortcomings of FMS and BnB MS to produce a novel, scalable, dynamic task allocation algorithm. To do this, we remove tasks that an agent should never perform with our online domain pruning algorithm (ODP), and interleave this with optimising Equation 6 using branch-and-bound and FMS.

Online Domain Pruning

As we mentioned earlier, the functions we consider represent the utility gained from using coalitions of agents to perform tasks. This allows us to compute a variable's contribution to a function for a given task, as part of a particular variable assignment. Now, if we sum the contributions of all $(x_i = t_j)$ in a particular $\mathbf{x}_j$, we obtain $f_j(\mathbf{x}_j)$. Hence, these contribution values can be used to locally compute which variable states (if any) will never provide a significant enough gain in utility. This is achieved by computing exact upper and lower bounds on each variable's contributions.

Algorithm 1 ODP at function f_j

Procedure SENDPRUNINGMESSAGETO(x_i)
Compute $\delta_j(x_i)^{ub}$ and $\delta_j(x_i)^{lb}$
Send $\langle \delta_j(x_i)^{ub}, \delta_j(x_i)^{lb} \rangle$ to x_i.
Procedure: On receipt of PRUNE from x_i
$\mathcal{X}_j \leftarrow \mathcal{X}_j \setminus x_i$ `// Remove from domain`
Procedure: On receipt of ADD from x_i
$\mathcal{X}_j \leftarrow \mathcal{X}_j \cup x_i$ `// Add to domain`
Run procedure SENDPRUNINGMESSAGETO(x_k) for all $x_k \in \mathcal{X}_j$

In more detail, we compute the upper bound on the contribution of x_i to f_j as $\delta_j(x_i)^{ub} = \max_{\mathbf{x}_j} \delta_j(x_i, \mathbf{x}_j)$. Similarly, we compute the lower bound on x_i's contribution to f_j by $\delta_j(x_i)^{lb} = \min_{\mathbf{x}_j} \delta_j(x_i, \mathbf{x}_j)$. Hence, $\delta_j(x_i)^{ub} \geq \delta_j(x_i)^{lb}, \forall f_j \in \mathcal{F}_i, x_i \in \mathcal{X}$. We can compute these bounds for any arbitrary function simply by using brute force, or by analytically deriving specific bounds for the specific value function used. The computational impact of computing these bounds by brute force can be mitigated by caching the bounds for a function so they need only be computed once.

Algorithm 2 ODP at variable x_i.

Procedure STARTUP()
$\tilde{\mathcal{T}}_i = \mathcal{T}_i$ `// Initialise modified domain`
$\mathbf{\Delta} = \emptyset$ `// Initialise stored bounds`
Send ADD to all f_j where $t_j \in \tilde{\mathcal{T}}_i$.
Procedure: On receipt of $\langle \delta_j(x_i)^{ub}, \delta_j(x_i)^{lb} \rangle$ message from f_j
if $t_j \notin \mathcal{T}_i$ **then**
  $\mathcal{T}_i \leftarrow \mathcal{T}_i \cup t_j$ `// Add to domain`
  $\tilde{\mathcal{T}}_i \leftarrow \tilde{\mathcal{T}}_i \cup t_j$ `// Add to modified domain`
if $\langle \delta_j(x_i)^{ub}, \delta_j(x_i)^{lb} \rangle \notin \mathbf{\Delta}$ **then**
  $\mathbf{\Delta}_j = \langle \delta_j(x_i)^{ub}, \delta_j(x_i)^{lb} \rangle$.
  if $\exists \mathbf{\Delta}_k, \forall f_k$ where $t_k \in \tilde{\mathcal{T}}_i$. **then**
    while $\exists t_j \in \tilde{\mathcal{V}}_i : \delta_j(x_i)^{ub} < \max_{t_k \in \tilde{\mathcal{T}}_i} \delta_k(x_i)^{lb}$ **do**
      $\tilde{\mathcal{T}}_i \leftarrow \tilde{\mathcal{T}}_i \setminus t_j$ `// Remove from domain`
      $\mathbf{\Delta} \leftarrow \mathbf{\Delta} \setminus \mathbf{\Delta}_j$ `// Remove from stored bounds`
      Send PRUNE to t_j
Procedure: a_i can no longer perform t_j
$\mathcal{T}_i \leftarrow \mathcal{T}_i \setminus t_j$
if $t_j \in \tilde{\mathcal{T}}_i$ **then** `// Removed task has not been pruned`
  Send ADD to all f_k where $t_k \in \mathcal{T}_i \cap \tilde{\mathcal{T}}_i$
  $\tilde{\mathcal{T}}_i \leftarrow \mathcal{T}_i$ `// Start pruning again`

This allows bound computation to be less computationally expensive than re-computing messages such as those in Equations 4 and 5, as the values of messages such as these rely on messages received from others. Thus, whilst there does exist a tradeoff between computing these bounds in order to reduce the complexity of FMS, and simply running FMS, we show in our empirical evaluation that, in practice, running ODP and FMS is much faster than just using FMS.

As mentioned above, contribution bounds can be used to find any variable states which will never give a significant enough gain in utility. We call such states *dominated* states. Thus, in this phase, we use localised message-passing to find and remove dominated states from variable domains. In addition, we recognise that changing variable domains (and addition and removal of variables themselves) may change which states are dominated, and we specifically cater for this by maintaining and updating state information for each variable, only passing new messages when they are absolutely necessary. Moreover, we show that this phase prunes all dominated states and never compromises solution quality.

In more detail, we define dominated variable states as those that can never maximise Equation 1, regardless of what others do. Formally, at variable x_i, task t_j is dominated if there exists a task t^* such that:

$$\delta_j(x_i, \mathbf{x}_{-i} \cup (x_i = t_j)) \leq \delta^*(x_i, \mathbf{x}_{-i} \cup (x_i = t^*)), \forall \mathbf{x}_{-i} \in \mathbf{X} \quad (7)$$

where $\mathbf{x}_{-i}$ represents the state of all variables excluding x_i, and $\mathbf{X}$ is the set of all possible joint variable states for $\mathbf{x}_{-i}$.

We can decentralise the computation of these dominated states through local message-passing, as described in Algorithms 1 and 2, where, at a variable, $\mathbf{\Delta}$ represents the stored contribution bounds received from functions, and $\tilde{\mathcal{T}}_i$ represents a variable's *modified* domain — i.e., with values removed when pruned. Storing the received bounds and modified domains allows ODP to run online, as a variable can update this domain in response to changes in the environment using both newly- and previously-received bounds (as we will discuss later). This is different to (Stranders et al.

2009), where variable domains would have to be completely restored and the algorithm re-run after every change in the environment, incurring potentially unnecessary overheads.

In more detail, ODP begins at the variables (lines 1–4, Algorithm 2), which make a copy of their domains so that they can keep track of their pruned domain as well as their full domain (so that it can be restored after a change, if needed), and announce which tasks they can perform. When a function receives one of these announcements, it will send upper and lower contribution bounds (Algorithm 1, lines 6–8), which the variable uses to update its domain (if needed) and stored bounds (lines 5–10, Algorithm 2). Next, if the variable is aware of the bounds on its contribution to every task it can perform (i.e., has stored bounds for every task in its domain) then it can begin searching for dominated states. If any dominated states are found, then they are removed from the modified domain, and functions representing those states are informed (lines 11–15, Algorithm 2). In response, the functions will update their own domains. Below, we prove that ODP will never prune an optimal state, under the assumption that an agent's optimal state only changes if there is a change in the environment.

Theorem 1. *ODP never prunes optimal states from $\tilde{\mathcal{T}}_i$.*

Proof. Let t^* be the optimal state of x_i. To prune t^* from $\tilde{\mathcal{T}}_i$, there must exist at least one $t_j \in \mathcal{T}_i \setminus t^*$ where Equation 7 holds. However, by definition, t^* is x_i's optimal state if Equation 7 holds for all $t_k \in \mathcal{T}_i \setminus t^*$. Therefore, it is impossible for such a t_j to exist, and t^* can never be pruned. □

Therefore, since ODP cannot prune an optimal state from $\tilde{\mathcal{T}}$, it follows that ODP can never prune all states from $\tilde{\mathcal{T}}$. Given this, ODP will always converge to a point where no more states can be pruned.

Once the algorithm has completed, BnB FMS begins execution (as described in the next section). However, if x_i detects that it can no longer perform task t_j, then t_j is removed from x_i's domain and updated domain (lines 16–20). If x_i had previously pruned t_j from $\tilde{\mathcal{T}}_i$ during ODP, then no further messages need to be sent. However, if t_j *was* in $\tilde{\mathcal{T}}_i$ after ODP this means that t_j could have been x_i's optimal state, so x_i must re-add any previously removed states to $\tilde{\mathcal{T}}_i$, and re-send ADD messages in case the removal of t_j has impacted which states are dominated. After each of these steps, the operation of BnB FMS continues.

BnB FMS

Once ODP has completed, if, for any x_i, $|\tilde{\mathcal{T}}_i| > 1$ (i.e., pruning was unable to solve the problem completely), then x_i will begin the execution of FMS by sending q messages as described in Equation 5. However, if $|\tilde{\mathcal{T}}_i| = 1$, then FMS does not need to be run at x_i, as the domain has been pruned to the solution. If FMS does need to be run, then the agent proceeds with FMS when it receives r messages (see Equation 4) from its neighbours.

The final element of our algorithm is the use of branch-and-bound search in FMS: specifically, when a function node is performing its maximisation in order to reduce the joint variable state space that needs to be explored. Now, as given earlier, in Equation 4, the maximisation at a factor is a computational bottleneck, because a function node f_j must enumerate all $2^{|\mathcal{T}_j|}$ valid states and choose the one with the maximum utility. Stranders et al. (2009) use a similar technique to this, but do not exploit the fact that $V(\emptyset, t_j) = 0, \forall t_j \in \mathcal{T}$, instead considering every possible state a variable can take. In contrast, we specialise this to task allocation and reduce the size of the search tree by only considering whether or not an agent is assigned to a task.

In this phase, we use search trees to prune joint states from the state space being explored at each function in the factor graph. To reduce this state space, we build a search tree, where each node is a partial joint state $\hat{\mathbf{x}}$ (where some variable states are not yet set), and leaf nodes are full joint states (where the values of all variables are set). We apply branch-and-bound on this search tree by computing the maximum value that can be gained from each branch (combining both utility and received messages), and pruning dominated branches. To do this, we first rename the expression in brackets from Equation 4 as $\tilde{r}$, so that message r is:

$$r_{j\to i}(x_i) \text{ for } x_i = t_j \text{ and any } x_i \neq t_j \text{ where:}$$

$$r_{j\to i}(x_i) = \max_{\mathbf{x}_j \setminus x_i} \tilde{r}_{j\to i}(\mathbf{x}_j) \tag{8}$$

where:

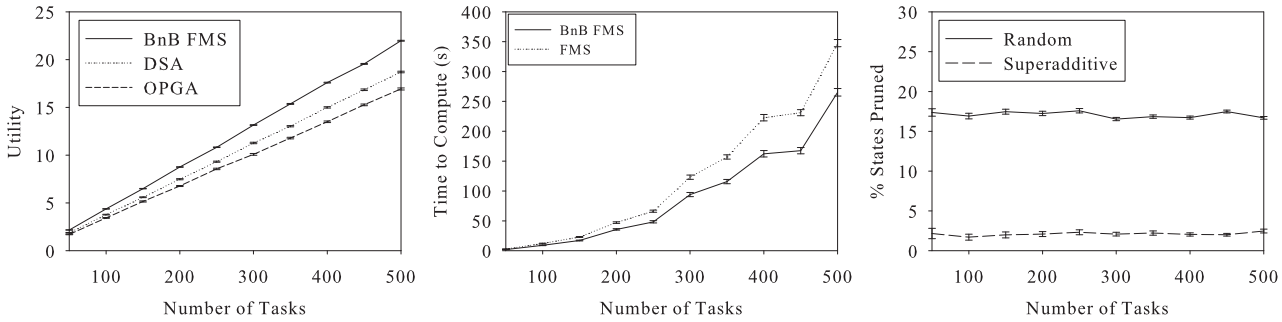
$$\tilde{r}_{j\to i}(\mathbf{x}_j) = f_j(\mathbf{x}_j) + \sum_{x_k \in X_j \setminus x_i} q_{k\to j}(x_k) \tag{9}$$

Hence, we wish to find $\max_{\mathbf{x}_j \setminus x_i} \tilde{r}_{j\to i}(\mathbf{x}_j)$.

Algorithm 3 Online Joint State Pruning at function f_j

```
1: procedure FINDMAX(x_i)
2: x̂_r = {−, ..., −, x_i, −, ..., −}
3: Return EXPANDTREE(x̂_r)
4: procedure EXPANDTREE(x̂)
5: if x̂ is not fully assigned then
6:    x_k = first unassigned variable in x̂
7:    x̂_true = x̂ ∪ (x_k = t_j); x̂_false = x̂ ∪ (x_k ≠ t_j)
8:    Compute r̃_{j→i}(x̂_true)^ub, r̃_{j→i}(x̂_true)^lb, r̃_{j→i}(x̂_false)^ub and
      r̃_{j→i}(x̂_false)^lb
9:    if r̃_{j→i}(x̂_true)^ub < r̃_{j→i}(x̂_false)^lb then
10:      Return EXPANDTREE(x̂_false)
11:   else if r̃_{j→i}(x̂_false)^ub < r̃_{j→i}(x̂_true)^lb then
12:      Return EXPANDTREE(x̂_true)
13: else
14:   Return r̃_{j→i}(x̂)
```

We give our online branch and bound algorithm in Algorithm 3 (specifically, procedure FINDMAX), and explain it below. First, we construct our search tree (line 2): rooted at $\hat{\mathbf{x}}_r = \{-, \ldots -, x_i, -, \ldots, -\}$, where $-$ indicates an unassigned variable, x_i is the variable that the message is being sent to — i.e., the state that is fixed for that message. Then, in line 6, we choose the first unassigned variable in $\hat{\mathbf{x}}_r$ (say, x_1), and create a branch of the tree for each possible value of that variable — in our formulation, this would form two more partial states, $\{(x_1 = t_j), -, \ldots -, x_i, -, \ldots, -\}$ and $\{(x_1 \neq t_j), -, \ldots -, x_i, -, \ldots, -\}$ (line 7). Next, we estimate upper and lower bounds on the maximum value of $\tilde{r}_{j\to i}(\hat{\mathbf{x}})$ that can be obtained by further completing the partial state for each subtree (line 8). In more detail, the upper



(a) Utility. (b) Wall time. (c) States pruned.

Figure 1: Empirical Results: Scalability and Solution Quality.

bound is computed as follows:

$$\tilde{r}_{j\to i}(\hat{\mathbf{x}})^{ub} = f_j(\hat{\mathbf{x}})^{ub} + \tilde{q}(\hat{\mathbf{x}})^{ub} \tag{10}$$

where $f_j(\hat{\mathbf{x}})^{ub}$ is found, depending on the function, by brute force or otherwise, and $\tilde{q}(\hat{\mathbf{x}})^{ub}$ is computed as:

$$\tilde{q}(\hat{\mathbf{x}})^{ub} = \sum_{\substack{\hat{x}_k \in \mathcal{X}_j \setminus i, \\ \hat{x}_k \neq -}} q_{k\to j}(\hat{x}_k) + \sum_{\substack{\hat{x}_k \in \mathcal{X}_j \setminus x_i, \\ \hat{x}_k = -}} \max_{x_k \in t_j, t_{-j}} q_{k\to j}(x_k) \tag{11}$$

where t_{-j} indicates any value other than t_j. Similarly, we compute the lower bound as follows:

$$\tilde{r}_{j\to i}(\hat{\mathbf{x}})^{lb} = f_j(\hat{\mathbf{x}})^{ub} + \tilde{q}(\hat{\mathbf{x}})^{lb} \tag{12}$$

where $f_j(\hat{\mathbf{x}})^{ub}$ is, again, either found by brute force or otherwise, and $\tilde{q}(\hat{\mathbf{x}})^{lb}$ is computed as:

$$\tilde{q}(\hat{\mathbf{x}})^{lb} = \sum_{\substack{\hat{x}_k \in \mathcal{X}_j \setminus i, \\ \hat{x}_k \neq -}} q_{k\to j}(\hat{x}_k) + \sum_{\substack{\hat{x}_k \in \mathcal{X}_j \setminus x_i, \\ \hat{x}_k = -}} \min_{x_k \in t_j, t_{-j}} q_{k\to j}(x_k) \tag{13}$$

With these bounds, any branch where the upper bound for $\tilde{\mathbf{x}}_{true}$ is lower than the lower bound for $\tilde{\mathbf{x}}_{false}$, or the upper bound for $\tilde{\mathbf{x}}_{false}$ is lower than the lower bound for $\tilde{\mathbf{x}}_{true}$ can be pruned (represented in Algorithm 3 lines 11–14 by not expanding pruned branches). The algorithm proceeds by expanding the next unassigned variable in the partial joint state represented by the remaining branch of the tree (line 6), and computing lower and upper bounds on the children. If possible, branches will be pruned, and so on, until we arrive at the leaves of the tree (where upper and lower bounds will be equal), which represent complete joint variable states (line 13), and we have found the joint state with the maximum utility (line 14). Next, we prove that this phase of BnB FMS will not prune a locally optimal solution:

Theorem 2. *A locally optimal joint state will not be removed from the search tree under Algorithm 3.*

Proof. $\delta_j(x_i \neq t_j, \mathbf{x}_j) = 0, \forall \mathbf{x}_j$. Therefore, $f_j(\mathbf{x}_j \cup (x_i = t_k)) = f_j(\mathbf{x}_j \cup (x_i = t_l)), \forall x_i \in \mathcal{X}_j, k \neq l$ and $k, l \neq j$. Hence, these binary values are representative of the entire domains of $x_i \in \mathcal{X}$, and so all possible states of $x_i \in \mathcal{X}_j$ are covered by the search tree. Given this and the fact that the bounds used are true upper and lower bounds, we can guarantee that the optimal joint state will not be pruned from the search tree. □

Therefore, under BnB FMS, the joint state chosen at each agent in response to the messages they have received will always be locally optimal.

Empirical Evaluation

To empirically determine the savings brought about by BnB FMS, we conducted two sets of experiments: to demonstrate scalability and solution quality, and to evaluate communicational savings in response to dynamism, to show that our algorithm is applicable in real-world task allocation environments. We compare BnB FMS to state-of-the-art algorithms mentioned earlier, namely DSA (Fitzpatrick and Meertens 2003), SDPOP (Petcu and Faltings 2005), FMS (Ramchurn et al. 2010), and OPGA (Chapman et al. 2009). We compare to BnB MS only in the dynamism experiments, since FMS has already been shown to achieve the same utility as max-sum, with less computation (Ramchurn et al. 2010).

Scalability and Solution Quality

To evaluate the scalability and solution quality achieved by BnB FMS, we generated 200 environments containing $|\mathcal{T}| \in \{50, 100, \cdots, 500\}$ and $|\mathcal{A}| = \frac{|\mathcal{T}|}{2}$, so that agents don't all end up being connected to the same few tasks. We then generated links amongst agents and tasks to create random graphs, trees, and scale-free graphs, for link density (average number of edges per node) $d \in \{3, 4, 5\}$. We evaluate differing link densities because as the number of links between tasks and agents increases, it becomes much harder to find an allocation of agents to tasks, as per Equation 4. We define f_j as a random coalition value function. We ran each algorithm over all graphs, running $|\mathcal{T}| + |\mathcal{A}|$ iterations of FMS and BnB FMS, and measured the wall time of a synchronous implementation of each algorithm, utility achieved, and the number of states pruned by BnB FMS. Our results are plotted with 95% confidence intervals in Figure 1.

SDPOP (our optimal benchmark) exhausted available memory even with 50 tasks and 25 agents, which was the smallest environment under evaluation, so it is not plotted on the graphs. Additionally, all algorithms performed, broadly, the same, regardless of graph type and density, so we only reproduce random graph results with $d = 3$ here. Figure 1(a) shows that, in terms of utility gained, BnB FMS outperforms DSA by up to 15% and OPGA by up to 23%, without

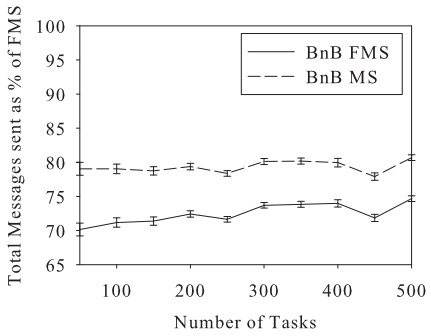


Figure 2: Empirical Results: Dynamism.

sacrificing the good solution quality achieved by FMS (BnB FMS and FMS gained the same utility, so the utility gained by FMS is not plotted) and requires up to 31% less time than FMS (Figure 1(b)). The times taken by DSA and OPGA are very small (at most, 24 ms) and so are not plotted on the graph: however, DSA and OPGA do not achieve as high solution quality as FMS and BnB FMS, which take between 1s and 260s to run to achieve that quality.

In addition, in Figure 1 (c), we show the percentage of states pruned by ODP for random and superadditive f_j. For both functions, pruning by ODP is affected only by the structure of the function (around 15% more states are pruned with a random function), not the size of the environment, as shown by the percentage of states removed being constant. This demonstrates that FMS will always consider more states than BnB FMS, demonstrating scalability.

Coping with Dynamism

Our second experiment demonstrates the performance of BnB FMS in a dynamic scenario, to compare the communication used to FMS, and running domain pruning after every change (which is similar to what BnB MS does, so we call it BnB MS). To do this, we generated 200 random graphs as above, and performed n random changes[3] on each graph at each of 10 timesteps, where n is drawn from a Poisson distribution for each timestep, with $\lambda = \frac{|\mathcal{T}|}{10} - 1$, running each algorithm at each timestep. We measured the sum of messages sent over the entire execution, including preprocessing steps: we plot the messages sent by BnB FMS and BnB MS as a percentage of those sent by FMS in Figure 2, with 95% confidence intervals. Again, we found graph density and structure made no impact on the performance of the algorithms, so we only reproduce results for a random graph with $d = 3$ here. In addition, we found that varying λ produced similar results. Figure 2 shows that BnB MS sends up to 23% fewer messages than FMS. As a result of specifically catering for dynamic environments, BnB FMS sends 30% fewer messages than FMS (10% fewer than BnB MS, using much smaller messages), whilst, as shown earlier, achieving the same utility as FMS in less time. In addition, the saving in messages is almost constant as the environment grows, because the number of messages sent is partially linked to the number of states pruned from the graph, as pruning states is equivalent to pruning edges in the factor graph. This, again, demonstrates the scalability of BnB FMS.

[3]Chosen uniformly from adding or removing an agent or task, where existing tasks/agents are randomly selected and removed, and new tasks/agents are created with connections to d randomly-selected tasks/agents.

Conclusions

In this paper, we presented BnB FMS, a distributed algorithm for task allocation. BnB FMS interleaves online domain pruning with fast-max-sum and applies branch-and-bound search when computing messages to be sent by a function in order to prune the state space to be explored further. Our empirical evaluation of BnB FMS shows that it achieves up to 23% more utility than state-of-the-art approximations. Solutions are found in up to 31% less time than FMS (which achieves the same utility as BnB FMS), and require at least 25% fewer messages than comparable algorithms in dynamic environments. Our future work will focus on bounding the quality of the solutions produced by BnB FMS so that it can be used in environments where theoretically provable solution quality is a key requirement.

Acknowledgments

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

References

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

Chapman, A.; Micillo, R. A.; Kota, R.; and Jennings, N. R. 2009. Decentralised dynamic task allocation: A practical game-theoretic approach. In *Proc. of AAMAS 2009*, 915–922.

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

Fitzpatrick, S., and Meertens, L. 2003. Distributed coordination through anarchic optimization. In Lesser, V.; Jr., C. L. O.; and Tambe, M., eds., *Distributed Sensor Networks: A Multiagent Perspective*. 257–295.

Kschischang, F. R.; Frey, B. J.; and Loeliger, H. A. 2001. Factor graphs and the sum-product algorithm. *IEEE Trans. on Information Theory* 47(2):498–519.

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

Ramchurn, S. D.; Farinelli, A.; Macarthur, K. S.; and Jennings, N. R. 2010. Decentralised Coordination in RobocupRescue. *The Computer Journal* 53(9):1447–1461.

Scerri, P.; Farinelli, A.; Okamoto, S.; and Tambe, M. 2005. Allocating tasks in extreme teams. In *Proc. of AAMAS 2005*, 727–734.

Shehory, O., and Kraus, S. 1998. Methods for task allocation via agent coalition formation. *Artificial Intelligence* 101:165–200.

Stranders, R.; Farinelli, A.; Rogers, A.; and Jennings, N. 2009. Decentralised coordination of mobile sensors using the max-sum algorithm. In *Proc. of IJCAI 2009*, 299–304.

Zheng, X., and Koenig, S. 2008. Reaction functions for task allocation to cooperative agents. In *Proc. of AAMAS 2008*, 559–566.