SPIDER attack on a network of POMDPs: Towards quality bounded solutions

Pradeep Varakantham, Janusz Marecki, Milind Tambe, Makoto Yokoo*
University of Southern California, Los Angeles, CA 90089, {varakant, marecki, tambe}@usc.edu
* Dept. of Intelligent Systems, Kyushu University, Fukuoka, 812-8581 Japan, yokoo@is.kyushu-u.ac.jp

October 17, 2006

Abstract

Distributed Partially Observable Markov Decision Problems (Distributed POMDPs) are a popular approach for modeling multi-agent systems acting in uncertain domains. Given the significant computational complexity of solving distributed POMDPs, one popular approach has focused on approximate solutions. Though this approach provides for efficient computation of solutions, the algorithms within this approach do not provide any guarantees on the quality of the solutions. A second less popular approach has focused on a global optimal result, but at considerable computational cost. This paper overcomes the limitations of both these approaches by providing SPIDER (Search for Policies In Distributed EnviRonments), which provides quality-guaranteed approximations for distributed POMDPs. SPIDER allows us to vary this quality guarantee, thus allowing us to vary solution quality systematically. SPIDER and its enhancements employ heuristic search techniques for finding a joint policy that satisfies the required bound on the quality of the solution.

1 Introduction

Distributed Partially Observable Markov Decision Problems (Distributed POMDPs) are emerging as a popular approach for modeling multiagent teamwork. This approach is used for modeling the sequential decision making in multiagent systems under uncertainty [10, 4, 1, 2, 14]. The uncertainty arises on account of non-determinism in the outcomes of actions and because the world state may only be partially (or incorrectly) observable. Unfortunately, as shown by Bernstein *et al.* [3], the problem of finding the optimal joint policy for general distributed POMDPs is NEXP-Complete.

Researchers have attempted two different types of approaches towards solving these models. The first category consists of highly efficient approximate techniques, that may not reach globally optimal solutions [2, 10, 12]. The key problem with these techniques has been their inability to provide any guarantee on the quality of the solution. In contrast, the second less popular category of approaches has focused on a global optimal result [14, 6, 11]. Though these approaches obtain optimal solutions, they do so at a significant computational cost.

To address these problems with the existing approaches, we propose approximate techniques that provide an error bound on the quality of the solution. We initially propose a technique called SPIDER (Search for Policies In Distributed EnviRonments) that employs heuristic techniques in searching the joint policy space. The key idea in SPIDER is the use of a branch and bound search (based on a MDP heuristic function) in exploring the space of joint policies. We then provide further enhancements (one exact and one approximate) to improve the efficiency of the basic SPIDER algorithm, while providing error bounds on the quality of the solutions. The first enhancement is based on the idea of initially performing branch and bound search on abstract policies (representing a group of policies), and then extending to the individual policies. Second enhancement is based on bounding the search approximately given a parameter that determines the difference from the optimal solution.

We experimented with the sensor network domain presented in Nair *et al*. [11]. In our experiments, we illustrate that SPIDER dominates an existing global optimal approach called GOA and also that by utilizing the approximation enhancement, SPIDER provides significant improvements in run-time performance while not losing significantly on quality.

We illustrate an example distributed sensor net domain in Section 2, and provide the ND-POMDP formalism in Section 3.1, that is motivated by the need to model planning under uncertainty in such domains. We provide description of a relevant algorithm, GOA in Section 3.2. The key contributions of this paper, the SPIDER algorithm and its enhancements are presented in Section 4. The run-time and value comparisons between these techniques and the GOA algorithm are presented in Section 5.

2 Domain: Distributed Sensor Nets

We describe an illustrative problem provided in [11] within the distributed sensor net domain, motivated by the real-world challenge in [7]. Here, each sensor node can scan in one of four directions — North, South, East or West (see Figure 1). To track a target and obtain associated reward, two sensors with overlapping scanning areas must coordinate by scanning the same area simultaneously. For instance, to track a target in Loc1-1, sensor1 would need to scan 'East' and sensor2 would need to scan 'West' simultaneously. Thus, sensor agents have to act in a coordinated fashion.

We assume that there are two independent targets and that each target's movement is uncertain and unaffected by the sensor agents. Based on the area it is scanning, each sensor receives observations that can have false positives and false negatives. The sensors' observations and transitions are independent of each other's actions e.g.the observations that sensor1 receives are independent of sensor2's actions. Each agent incurs a cost for scanning whether the target is present or not, but no cost if it turns off. Given the sensors' observational uncertainty, the targets' uncertain transitions and the distributed nature of the sensor nodes, these sensor nets provide a useful domains for applying distributed POMDP models.

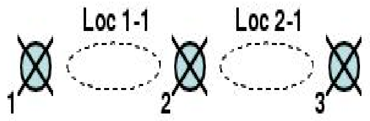

Figure 1: There are three sensors numbered 1, 2 and 3, each of which can scan in one of four directions. To track a target two sensors must scan an overlapping area, e.g. to track a target in location loc-1-1, sensor1 must scan east and sensor2 must scan west

3 Background

3.1 Model: Network Distributed POMDP

The ND-POMDP model was first introduced in [11]. It is defined as the tuple $\langle S, A, P, \Omega, O, R, b \rangle$, where $S = \times_{1 \leq i \leq n} S_i \times S_u$ is the set of world states. S_i refers to the set of local states of agent i and S_u is the set of unaffectable states. Unaffectable state refers to that part of the world state that cannot be affected by the agents' actions, e.g. environmental factors like target locations that no agent can control. $A = \times_{1 \leq i \leq n} A_i$ is the set of joint actions, where A_i is the set of action for agent i.

ND-POMDP assumes *transition independence*, where the transition function is defined as $P(s, a, s') = P_u(s_u, s'_u) \cdot \prod_{1 \leq i \leq n} P_i(s_i, s_u, a_i, s'_i)$, where $a = \langle a_1, \ldots, a_n \rangle$ is the joint action performed in state $s = \langle s_1, \ldots, s_n, s_u \rangle$ and $s' = \langle s'_1, \ldots, s'_n, s'_u \rangle$ is the resulting state. Agent i's transition function is defined as $P_i(s_i, s_u, a_i, s'_i) = \Pr(s'_i | s_i, s_u, a_i)$ and the unaffectable transition function is defined as $P_u(s_u, s'_u) = \Pr(s'_u | s_u)$.

$\Omega = \times_{1 \leq i \leq n} \Omega_i$ is the set of joint observations where Ω_i is the set of observations for agents i. *Observational independence* is assumed in ND-POMDPs i.e., the joint observation function is defined as $O(s, a, \omega) = \prod_{1 \leq i \leq n} O_i(s_i, s_u, a_i, \omega_i)$, where $s = \langle s_1, \ldots, s_n, s_u \rangle$ is the world state that results from the agents performing $a = \langle a_1, \ldots, a_n \rangle$ in the previous state, and $\omega = \langle \omega_1, \ldots, \omega_n \rangle \in \Omega$ is the observation received in state s. The observation function for agent i is defined as $O_i(s_i, s_u, a_i, \omega_i) = \Pr(\omega_i | s_i, s_u, a_i)$. This implies that each agent's observation depends only on the unaffectable state, its local action and on its resulting local state.

The reward function, R, is defined as $R(s, a) = \sum_l R_l(s_{l1}, \ldots, s_{lk}, s_u, \langle a_{l1}, \ldots, a_{lk} \rangle)$, where each l could refer to any sub-group of agents and $k = |l|$. Based on the reward function, we construct an *interaction hypergraph* where a hyper-link, l, exists between a subset of agents for all R_l that comprise R. The *interaction hypergraph* is defined as $G = (Ag, E)$, where the agents, Ag, are the vertices and $E = \{l | l \subseteq Ag \wedge R_l \text{ is a component of } R\}$ are the edges.

The initial belief state (distribution over the initial state), b, is defined as $b(s) = b_u(s_u) \cdot \prod_{1 \leq i \leq n} b_i(s_i)$, where b_u and b_i refer to the distribution over initial unaffectable state and agent i's initial belief state, respectively. We define agent i's neighbors' initial belief state as $b_{N_i} = \prod_{j \in N_i} b_j(s_j)$. We assume that b is available to all agents (although it is possible to refine our model to make available to agent i only b_u, b_i and

b_{N_i}). The goal in ND-POMDP is to compute the joint policy $\pi = \langle \pi_1, \ldots, \pi_n \rangle$ that maximizes the team's expected reward over a finite horizon T starting from the belief state b.

An ND-POMDP is similar to an *n-ary* DCOP where the variable at each node represents the policy selected by an individual agent, π_i with the domain of the variable being the set of all local policies, Π_i. The reward component R_l where $|l| = 1$ can be thought of as a local constraint while the reward component R_l where $l > 1$ corresponds to a non-local constraint in the constraint graph.

3.2 Algorithm: Global Optimal Algorithm (GOA)

In previous work, GOA has been defined as a global optimal algorithm for ND-POMDPs. GOA is the only algorithm where we have actual experimental results for ND-POMDPs of more than two agents. GOA borrows from a global optimal DCOP algorithm called DPOP[13]. GOA's message passing follows that of DPOP. The first phase is the UTIL propagation, where the utility messages, in this case values of policies, are passed up from the leaves to the root. Value for a policy at an agent is defined as the sum of best response values from its children and the joint policy reward associated with the parent policy. Thus, given a policy for a parent node, GOA requires an agent to iterate through all its policies, finding the best response policy and returning the value to the parent — while at the parent node, to find the best policy, an agent requires its children to return their best responses to each of its policies.

Unfortunately, for different policies of the parent, each child node C_i is required to re-compute best response values corresponding to the same policies within C_i. To avoid this recalculation, each C_i stores the sum of best response values from its children for each of its policies. This UTIL propagation process is repeated at each level in the tree, until the root exhausts all its policies. In the second phase of VALUE propagation, where the optimal policies are passed down from the root till the leaves.

GOA takes advantage of the local interactions in the interaction graph, by pruning out unnecessary joint policy evaluations (associated with nodes not connected directly in the tree). Since the interaction graph captures all the reward interactions among agents and as this algorithm iterates through all the relevant joint policy evaluations, this algorithm yields a globally optimal solution.

4 Search for Policies In Distributed EnviRonments (SPIDER)

As mentioned in Section 3.1, an ND-POMDP can be treated as a DCOP, where the goal is to compute a joint policy that maximizes the overall joint reward. The bruteforce technique for computing an optimal policy would be to scan through the entire space of joint policies. The key idea in SPIDER is to avoid computation of expected values for the entire space of joint policies, by utilizing upperbounds on the expected values of policies and the interaction structure of the agents.

Akin to some of the algorithms for DCOP [9, 13], SPIDER has a pre-processing step that constructs a DFS tree corresponding to the given interaction structure. We employ the Maximum Constrained Node (MCN) heuristic used in ADOPT [9], however other heuristics (such as MLSP heuristic from [8]) can also be employed. MCN heuristic tries to place agents with more number of constraints at the top of the tree. This tree governs how the search for the optimal joint policy proceeds in SPIDER.

In this paper, we employ the following notation to denote policies and expected values of joint policies:
π^{root} denotes the joint policy of all agents involved.
π^i denotes the joint policy of all agents in the sub-tree for which i is the root.
π^{-i} denotes the joint policy of agents that are ancestors to agents in the sub-tree for which i is the root.
π_i denotes a policy of the ith agent.
$\hat{v}[\pi_i, \pi^{-i}]$ denotes the upper bound on the expected value for π^i given π_i and policies of parent agents i.e. π^{-i}.
$v[\pi^i, \pi^{-i}]$ denotes the expected value for π^i given policies of parent agents π^{-i}.
$v^{max}[\pi_i, \pi^{-i}]$ denotes the threshold used for policy computation.

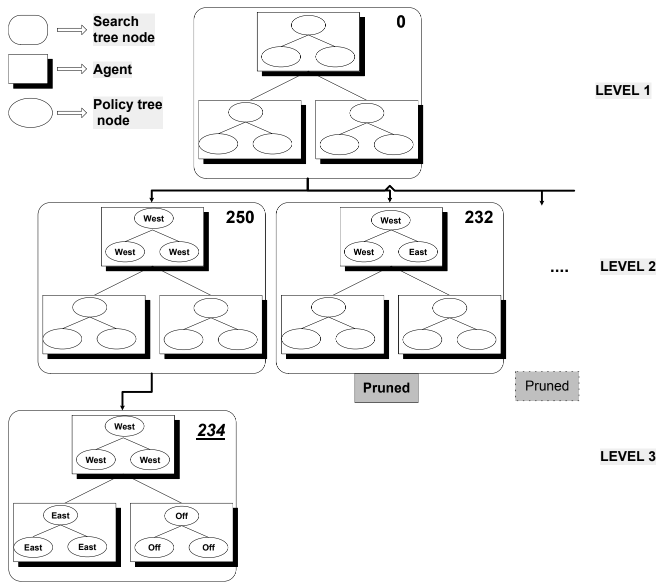

Figure 2: Execution of SPIDER, an example

SPIDER algorithm is based on the idea of branch and bound search, where the nodes in the search tree represent the joint policies, π^{root}. Figure 2 shows an example search tree for the SPIDER algorithm, using an example of the three agent chain. We create a tree from this chain, with the middle agent as the root of the tree. Note that in our example figure each agent is assigned a policy with T=2. Each rounded rectange (search tree node) indicates a partial/complete joint policy and a rectangle indicates an agent. Heuristic or actual expected value for a joint policy is indicated in the top right corner of the rounded rectangle. If the number is italicized and underlined, it implies that the actual expected value of the joint policy is provided. SPIDER begins with no policy assigned to any of the agents (shown in the level 1 of the search tree). Level 2 of the search tree indicates that the joint policies are sorted based on upper bounds computed for root agent's policies. Level 3 contains a node with a complete joint policy (a policy assigned to each of the agents). The expected value for this joint policy is used to prune out the nodes in level 2 (the ones with upper bounds ¡ 234)

In SPIDER, each non-leaf agent i potentially performs two steps:

1. Obtaining upper bounds and sorting: In this step, agent i computes upper bounds on the expected values, $\hat{v}[\pi_i, \pi^{-i}]$ of the joint policies π^i corresponding to each of its policy π_i and fixed parent policies. A MDP based heuristic is used to compute these upper bounds on the expected values. Detailed description about this MDP heuristic and other possible heuristics is provided in Section 4.1. All policies of agent i, Π_i are then sorted based on these upper bounds (also referred to as heuristic values henceforth) in descending order. Exploration of these policies (in step 2 below) are performed in this descending order. As indicated in the level 2 of the search tree of Figure 2, all the joint policies are sorted based on the heuristic values, indicated in the top right corner of each joint policy. This step is performed to provide a certain order to exploration (explained below). The intuition behind exploring policies in descending order of upper bounds, is that the policies with higher upper bounds could yield joint policies with higher expected values.

2. Exploration and Pruning: Exploration here implies computing the best response joint policy $\pi^{i,*}$ corresponding to fixed parent policies of agent i, π^{-i}. This is performed by iterating through all policies of agent i i.e. Π_i and computing best response policies of i's children (obtained by performing steps 1 and 2 at each of the child nodes) for each of agent i's policy, π_i. Exploration of a policy π_i yields actual expected value of a joint policy, π^i represented as $v[\pi^i, \pi^{-i}]$.

   Pruning refers to the process of avoiding exploring policies (or computing expected values) at agent i by using the maximum expected value, $v^{max}[\pi^i, \pi^{-i}]$ encountered until this juncture. Henceforth, this $v^{max}[\pi^i, \pi^{-i}]$ will be referred to as *threshold*. A policy, π_i need not be explored if the upper bound for that policy, $\hat{v}[\pi_i, \pi^{-i}]$ is less than the *threshold*. This is because the best joint policy that can be obtained from that policy will have an expected value that is less than the expected value of the current best joint policy.

On the other hand, each leaf agent in SPIDER computes the best response policy (and consequently its expected value) corresponding to fixed policies of its ancestors, π^{-i}. This is accomplished by computing expected values for each of the policies (corresponding to fixed policies of ancestors) and selecting the policy with the highest expected value.

Algorithm 1 provides the pseudo code for SPIDER. This algorithm outputs the best joint policy, $\pi^{i,*}$ (with an expected value greater than *threshold*) for the agents in the sub-tree with agent i as the root. Lines 3-8 compute the best response policy of a leaf agent i by iterating through all the policies (line 4) and finding the policy with the highest expected value (lines 5-8). Lines 9-23 computes the best response joint policy

Algorithm 1 SPIDER$(i, \pi^{-i}, threshold)$

```
 1: $\pi^{i,*} \leftarrow null$
 2: $\Pi_i \leftarrow$ GET-ALL-POLICIES $(horizon, A_i, \Omega_i)$
 3: if IS-LEAF(i) then
 4:     for all $\pi_i \in \Pi_i$ do
 5:         $v[\pi_i, \pi^{-i}] \leftarrow$ JOINT-REWARD $(\pi_i, \pi^{-i})$
 6:         if $v[\pi_i, \pi^{-i}] > threshold$ then
 7:             $\pi^{i,*} \leftarrow \pi_i$
 8:             $threshold \leftarrow v[\pi_i, \pi^{-i}]$
 9: else
10:     $children \leftarrow$ CHILDREN $(i)$
11:     $\hat{\Pi}_i \leftarrow$ SORTED-POLICIES$(i, \Pi_i, \pi^{-i})$
12:     for all $\pi_i \in \hat{\Pi}_i$ do
13:         $\tilde{\pi}^i \leftarrow \pi_i$
14:         if $\hat{v}[\pi_i, \pi^{-i}] < threshold$ then
15:             Go to line 12
16:         for all $j \in children$ do
17:             $jThres \leftarrow threshold - \Sigma_{k \in children, k != j} \hat{v}_k[\pi_i, \pi^{-i}]$
18:             $\pi^{j,*} \leftarrow$ SPIDER$(j, \pi_i \parallel \pi^{-i}, jThres)$
19:             $\tilde{\pi}^i \leftarrow \tilde{\pi}^i \parallel \pi^{j,*}$
20:             $\hat{v}_j[\pi_i, \pi^{-i}] \leftarrow v[\pi^{j,*}, \pi_i \parallel \pi^{-i}]$
21:         if $v[\tilde{\pi}^i, \pi^{-i}] > threshold$ then
22:             $threshold \leftarrow v[\tilde{\pi}^i, \pi^{-i}]$
23:             $\pi^{i,*} \leftarrow \tilde{\pi}^i$
24: return $\pi^{i,*}$
```

Algorithm 2 SORTED-POLICIES(i, Π_i, π^{-i})

```
 1: $children \leftarrow$ CHILDREN $(i)$
 2: $\hat{\Pi}_i \leftarrow null$ /* Stores the sorted list */
 3: for all $\pi_i \in \Pi_i$ do
 4:     $\hat{v}[\pi_i, \pi^{-i}] \leftarrow$ JOINT-REWARD $(\pi_i, \pi^{-i})$
 5:     for all $j \in children$ do
 6:         $\hat{v}_j[\pi_i, \pi^{-i}] \leftarrow$ GET-HEURISTIC$(\pi_i \parallel \pi^{-i}, j)$
 7:         $\hat{v}[\pi_i, \pi^{-i}] \xleftarrow{+} \hat{v}_j[\pi_i, \pi^{-i}]$
 8:     $\hat{v}[\pi_i, \pi^{-i}] \xleftarrow{+} \Sigma_{k \in children} \hat{v}[\pi_k, \pi_i \parallel \pi^{-i}]$
 9:     $\hat{\Pi}_i \leftarrow$ INSERT-INTO-SORTED $(\pi_i, \hat{\Pi}_i)$
10: return $\hat{\Pi}_i$
```

for agents in the sub-tree with i as the root. Sorting of policies (in descending order) based on heuristic policies is done on line 11.

Exploration of a policy i.e. computing best response joint policy corresponding to fixed parent policies is done in lines 12-23. This includes computation of best joint policies for each of the child sub-trees (lines 16-23). This computation in turn involves distributing the *threshold* among each of the children (line 17), recursively calling the SPIDER algorithm for each of the children (line 18) and maintaining the best expected value and the best joint policy (lines 21-23). **Pruning** of policies is performed in lines 20-21 by comparing the upper bound on the expected value against the *threshold*.

4.1 Heuristic Functions

The job of the heuristic function is to provide a quick estimate of the upper bound for the second component in Eqn 2 i.e. the expected value obtainable from the sub-tree for which i is the root. To achieve quick computation of this upper bound, we assume full observability for the agents in the $Tree(i)$ (does not include i)and compute the joint value, $\hat{v}[\pi^i, \pi^{-i}]$ for fixed policies of the agents in the set $\{Parents(i) \cup i\}$.

We use the following notation for presenting the equations for computing upper bounds/heuristic values:

$$
\begin{aligned}
p_r^t &\triangleq P_r(s_r^t, s_u^t, \pi_r(\vec{\omega}_r^t), s_r^{t+1}) \cdot O_r(s_r^{t+1}, s_u^{t+1}, \pi_r(\vec{\omega}_r^t), \omega_r^{t+1}) \\
\hat{p}_r^t &\triangleq p_r^t, if \ r \in \{Parents(i) \cup i\} \qquad (1) \\
&\triangleq P_r(s_r^t, s_u^t, \pi_r(\vec{\omega}_r^t), s_r^{t+1}), if \ r \in Tree(i) \\
p_u^t &\triangleq P(s_u^t, s_u^{t+1}) \\
s^t &= \langle s_{l1}^t, \ldots, s_{lk}^t, s_u^t \rangle \\
r_l^t &\triangleq R_l(s^t, \pi_{l1}(\vec{\omega}_{l1}^t), \ldots, \pi_{lk}(\vec{\omega}_{lk}^t)) \\
\hat{r}_l^t &\triangleq \max_{\{a_{l_j}\}|l_j \in Tree(i)} R_l(s^t, \ldots, \pi_{l_r}(\vec{\omega}_{l_r}^t), \ldots, a_{l_j}, \ldots) \\
v_l^t &\triangleq V_{\pi_l}^t(s_{l1}^t, \ldots, s_{lk}^t, s_u^t, \vec{\omega}_{l1}^t, \ldots \vec{\omega}_{lk}^t)
\end{aligned}
$$

The value function for an agent i is provided by the equation:

$$
\begin{aligned}
V_{\pi^i}^{\eta-1}(s^{\eta-1}, \vec{\omega}^{\eta-1}) = \sum_{l \in E^{-i}} v_l^{\eta-1} + \sum_{l \in E^i} v_l^{\eta-1}, where \qquad (2) \\
v_l^{\eta-1} = r_l^{\eta-1} + \sum_{\omega_l^\eta, s^\eta} p_{l_1}^{\eta-1} \ldots p_{l_k}^{\eta-1} p_u^{\eta-1} v_l^\eta
\end{aligned}
$$

Upper bound on the expected value for a link is computed using the following equation:

$$
\hat{v}_l^{\eta-1} = \hat{r}_l^{\eta-1} + \max_{\{a_j\}|j \in Tree(i)} \sum_{\omega_{l_r}^\eta|l_r \in Parents(i), s^\eta} \hat{p}_{l_1}^{\eta-1} \ldots \hat{p}_{l_k}^{\eta-1} p_u^{\eta-1} \hat{v}_l^\eta
$$

4.2 Abstraction

In SPIDER, the exploration/pruning phase can only begin after the heuristic (or upper bound) computation and sorting for the policies has finished. With this technique of abstraction, SPIDER-ABS, we provide an approach of interleaving exploration/pruning phase with the heuristic computation and sorting phase, thus possibly circumventing the exploration of a group of policies based on heuristic computation of one representative policy. The key steps in this technique are defining the representative/abstract policy and how *pruning* can be performed based on heuristic computations of this abstract policy.

Firstly, in addressing the issue of abstract policy, there could be multiple ways of defining this abstraction/grouping of policies. In this paper, we present one type of abstraction that utilizes a lower horizon policy to represent a group of higher horizon policies. An example of this kind of abstraction is illustrated in Figure 3. In the figure, a T=2 (Time horizon of 2) policy (of scanning east and then scanning west for either observation) represents all T=3 policies that have the same actions at the first two decision points (as the T=2 policy).

Secondly, with respect to *pruning* in the abstract policy space for agent i, we compute a *threshold* for the abstract policies based on the current *threshold*. With the kind of abstraction mentioned in the above para, we propose a heuristic that computes the maximum possible reward that can be accumulated in one time step and multiply it by the number of time steps to time horizon. Towards computing the maximum possible reward, we iterate through all the actions of all the agents involved (agents in the sub-tree with i as the root) and compute the maximum joint reward for any joint action.

For computing optimal joint policy for T=3 with SPIDER-ABS, a non-leaf agent i initially scans through all T=1 policies and sorts them based on heuristic computations. These T=1 policies are then *explored* in descending order of heuristic values and ones that have heuristic values less than the *threshold* for T=1 policies (computed using the heuristic presented in above para) are pruned. *Exploration* in SPIDER-ABS has the same definition as in SPIDER if the policy being *explored* has a horizon of policy computation which is equal to the actual time horizon (in the example it is 3). However, if a policy has a horizon less than the time horizon, then it is substituted by a group of policies that it represents (referred to as *extension* henceforth). Before substituting the abstract policy, this group of policies are sorted based on the heuristic values. At this juncture, if all the substituted policies have horizon of policy computation equal to the time horizon, then the *exploration*/*pruning* phase akin to the one in SPIDER ensues. In case of partial policies (horizon of policy less than time horizon), further *extension* of policies occurs. Similarly, a horizon based extension of policies or computation of best response is adopted at leaf agents in SPIDER-ABS.

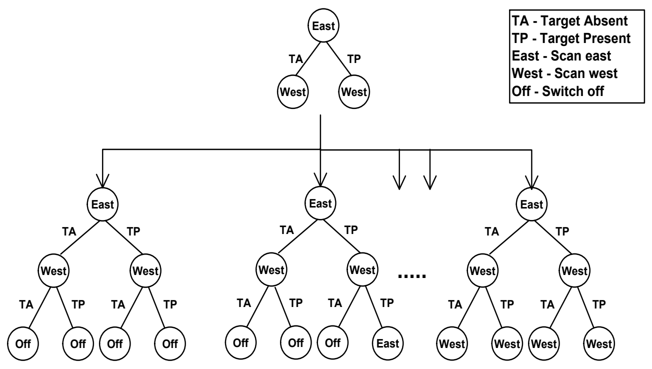

Figure 3: Example of abstraction

Algorithm 3 SPIDER-ABS$(i, \pi^{-i}, threshold)$

```
1: $\pi^{i,*} \leftarrow null$
2: $\Pi_i \leftarrow$ GET-ALL-POLICIES $(1, A_i, \Omega_i)$
3: **if** IS-LEAF(i) **then**
4:   **for all** $\pi_i \in \Pi_i$ **do**
5:     $absHeuristic \leftarrow$ GET-ABS-HEURISTIC $(\pi_i, \pi^{-i})$
6:     $absHeuristic \overset{*}{\leftarrow} (timeHorizon - \pi_i.horizon)$
7:     **if** $\pi_i.horizon = timeHorizon$ **then**
8:       $v[\pi_i, \pi^{-i}] \leftarrow$ JOINT-REWARD $(\pi_i, \pi^{-i})$
9:       **if** $v[\pi_i, \pi^{-i}] > threshold$ **then**
10:        $\pi^{i,*} \leftarrow \pi_i; threshold \leftarrow v[\pi_i, \pi^{-i}]$
11:    **else if** $v[\pi_i, \pi^{-i}] + absHeuristic > threshold$ **then**
12:      $\hat{\Pi}_i \leftarrow$ GET-POLICIES $(\pi_i.horizon + 1, A_i, \Omega_i, \pi_i)$
13:      /* Insert policies in the beginning of $\Pi_i$ in sorted order*/
14:      $\Pi_i \overset{+}{\leftarrow}$ INSERT-SORTED-POLICIES ($\hat{\Pi}_i$)
15:    REMOVE($\pi_i$)
16: **else**
17:   $children \leftarrow$ CHILDREN $(i)$
18:   $\Pi_i \leftarrow$ SORTED-POLICIES$(i, \Pi_i, \pi^{-i})$
19:   **for all** $\pi_i \in \Pi_i$ **do**
20:     $\tilde{\pi}^i \leftarrow \pi_i$
21:     $absHeuristic \leftarrow$ GET-ABS-HEURISTIC $(\pi_i, \pi^{-i})$
22:     $absHeuristic \overset{*}{\leftarrow} (timeHorizon - \pi_i.horizon)$
23:     **if** $\pi_i.horizon == timeHorizon$ **then**
24:       **if** $\hat{v}[\pi_i, \pi^{-i}] < threshold$ **then**
25:         Go to line 19
26:       **for all** $j \in children$ **do**
27:         $jThres \leftarrow threshold - \Sigma_{k \in children, k!=j} \hat{v}_k[\pi_i, \pi^{-i}]$
28:         $\pi^{j,*} \leftarrow$ SPIDER$(j, \pi_i \parallel \pi^{-i}, jThres)$
29:         $\tilde{\pi}^i \leftarrow \tilde{\pi}^i \parallel \pi^{j,*}; \hat{v}_j[\pi_i, \pi^{-i}] \leftarrow v[\pi^{j,*}, \pi_i \parallel \pi^{-i}]$
30:       **if** $v[\tilde{\pi}^i, \pi^{-i}] > threshold$ **then**
31:         $threshold \leftarrow v[\tilde{\pi}^i, \pi^{-i}]; \pi^{i,*} \leftarrow \tilde{\pi}^i$
32:     **else if** $\hat{v}[\pi^i, \pi^{-i}] + absHeuristic > threshold$ **then**
33:       $\hat{\Pi}_i \leftarrow$ GET-POLICIES $(\pi_i.horizon + 1, A_i, \Omega_i, \pi_i)$
34:       /* Insert policies in the beginning of $\Pi_i$ in sorted order*/
35:       $\Pi_i \overset{+}{\leftarrow}$ INSERT-SORTED-POLICIES ($\hat{\Pi}_i$)
36:   REMOVE($\pi_i$)
37: **return** $\pi^{i,*}$
```

4.3 Value ApproXimation (VAX)

In this section, we present an approximate enhancement to SPIDER called VAX. The input to this technique is an approximation parameter ϵ, which determines the difference between the optimal solution and the approximate solution. This approximation parameter is used at each agent for pruning out joint policies. The pruning mechanism in SPIDER and SPIDER-Abs dictates that a joint policy be pruned only if the threshold is exactly greater than the heuristic value. However, the idea in this technique is to prune out joint policies even if *threshold* plus the approximation parameter, ϵ is greater than the heuristic value.

Going back to the example of Figure 2, if the heuristic value for the second joint policy in step 2 were 238 instead of 232, then that policy could not be be pruned using SPIDER or SPIDER-Abs. However, in VAX with an approximation parameter of 5, the joint policy in consideration would also be pruned. This is because the *threshold* (234) at that juncture plus the approximation parameter (5) would have been greater than the heuristic value for that joint policy. As presented in the example, this kind of *pruning* can lead to fewer *explorations* and hence lead to an improvement in the overall run-time performance. However, this can entail a sacrifice in the quality of the solution because this technique can prune out a candidate optimal solution. A bound on the error made by this approximate algorithm is provided by Proposition 3.

4.4 Theoretical Results

Proposition 1 *Heuristic provided using the centralized MDP heuristic is admissible.*

Proof. For the value provided by the heuristic to be admissible, it should be an over estimate of the expected value for a joint policy. Thus, we need to show that:

For $l \in E^i$: $\hat{v}_l^t \geq v_l^t$.

We use mathematical induction on t to prove this.

Base case: $t = T - 1$. Since, $\max_z x_z \geq \sum_z pr_z \cdot x_z$, for $0 \leq pr_z \leq 1$ and a set of real numbers $\{x_z\}$, we have $\hat{r}_l^t > r_l^t$. Thus $\hat{v}_l^t \geq v_l^t$.

Assumption: Proposition holds for $t = \eta$, where $1 \leq \eta < T - 1$. Thus, $\hat{v}_l^\eta \geq v_l^\eta$

We now have to prove that the proposition holds for $t = \eta - 1$ i.e. $\hat{v}_l^{\eta-1} \geq v_l^{\eta-1}$.

The heuristic value function is provided by the following equation:

$$\hat{v}_l^{\eta-1} = \hat{r}_l^{\eta-1} + \max_{\{a_j\}|j \in Tree(i)} \sum_{\omega_{l_r}^\eta | l_r \in Parents(i), s^\eta} \hat{p}_{l_1}^{\eta-1} \ldots \hat{p}_{l_k}^{\eta-1} p_u^{\eta-1} \hat{v}_l^\eta$$

Rewriting the RHS and using Eqn 1

$$\begin{aligned} =& \hat{r}_l^{\eta-1} + \max_{\{a_j\}|j \in Tree(i)} \sum_{\omega_{l_r}^\eta | l_r \in Parents(i), s^\eta} p_u^{\eta-1} \\ & \prod_{m \in \{Parents(i) \cup i\}} p_{l_m}^{\eta-1} \prod_{n \in Tree(i)} \hat{p}_{l_n}^{\eta-1} \hat{v}_l^\eta \\ =& \hat{r}_l^{\eta-1} + \sum_{\omega_{l_r}^\eta | l_r \in Parents(i), s^\eta} p_u^{\eta-1} \prod_{m \in \{Parents(i) \cup i\}} p_{l_m}^{\eta-1} \\ & \prod_{n \in Tree(i)} \max_{\{a_j\}|j \in Tree(i)} \hat{p}_{l_n}^{\eta-1} \hat{v}_l^\eta \end{aligned}$$

Since $\max_z x_z \geq \sum_z pr_z \cdot x_z$, for $0 \leq pr_z \leq 1$ and $\hat{v}_l^\eta \geq v_l^\eta$ (from the assumption)

$$\geq \hat{r}_l^{\eta-1} + \sum_{\omega_{l_r}^\eta | l_r \in Parents(i), s^\eta} p_u^{\eta-1} \prod_{m \in \{Parents(i) \cup i\}} p_{l_m}^{\eta-1}$$
$$\prod_{n \in Tree(i)} \sum_{\omega_{l_n} | l_n \in Tree(i)} p_{l_n}^{\eta-1} v_l^\eta$$
$$\geq \hat{r}_l^{\eta-1} + \sum_{(\omega_{l_r}^\eta | l_r \in Parents(i), s^\eta)} \sum_{(\omega_{l_n} | l_n \in Tree(i))} p_u^{\eta-1}$$
$$\prod_{m \in \{Parents(i) \cup i\}} p_{l_m}^{\eta-1} \prod_{n \in Tree(i)} p_{l_n}^{\eta-1} v_l^\eta$$
$$\geq r_l^{\eta-1} + \sum_{(\omega_l^\eta, s^\eta)} p_u^{\eta-1} p_{l_1}^{\eta-1} \ldots p_{l_k}^{\eta-1} v_l^\eta$$

Thus proved. ■

Proposition 2 *SPIDER provides an optimal solution.*

Proof. SPIDER examines all possible joint policies given the interaction structure of the agents. The only exception being when a joint policy is *pruned* based on the heuristic value. Thus, as long as a candidate optimal policy is not pruned, SPIDER will return an optimal policy. As proved in Proposition 1, the expected value for a joint policy is always an upper bound. Hence when a joint policy is pruned, it cannot be an optimal solution.

Proposition 3 *In a problem with n agents, error bound on the solution quality for VAX with an approximation parameter of ϵ is given by $n\epsilon$.*

Proof. VAX prunes a joint policy only if the heuristic value (upper bound) is greater than the threshold by atleast ϵ. Thus, at each agent an error of atmost ϵ is introduced. Hence, there is an overall error bound of $n\epsilon$.

5 Experimental Results

All our experiments were conducted on the sensor network domain provided in Section 2. Network configurations presented in Figure 4 were used in these experiments. Algorithms that we experimented with as part of this paper include GOA, SPIDER, SPIDER-ABS and VAX. We performed two sets of experiments: (i) firstly, we compared the run-time performance of the algorithms mentioned above and (ii) secondly, we experimented with VAX to study the tradeoff between run-time and solution quality. Experiments were terminated if they exceeded the time limit of 10000 seconds[1].

Figure 5(a) provides the run-time comparisons between the optimal algorithms GOA, SPIDER, SPIDER-Abs and the approximate algorithm, VAX with varying epsilons. X-axis denotes the type of sensor network configuration used, while Y-axis indicates the amount of time taken (on a log scale) to compute the optimal solution. The

[1]Machine specs for all experiments: Intel Xeon 3.6 GHZ processor, 2GB RAM

time horizon of policy computation for all the configurations was 3. For each configuration (3-chain, 4-chain, 4-star and 5-star), there are five bars indicating the time taken by GOA, SPIDER, SPIDER-Abs and VAX with 2 different epsilons. GOA did not terminate within the time limit for 4-star and 5-star configurations. SPIDER-Abs dominated the other two optimal algorithms for all the configurations. For instance, for the 3-chain configuration, SPIDER-ABS provides 230-fold speedup over GOA and 2-fold speedup over SPIDER and for the 4-chain configuration it provides 22-fold speedup over GOA and 4-fold speedup over SPIDER. VAX with two different approximation parameters (ϵ was 40 and 80 respectively for VAX-1 and VAX-2) provided a further improvement in performance over SPIDER-Abs. For instance, for the 5-star configuration VAX-2 provided a 41-fold speedup over SPIDER-Abs.

Figures 5(b) provides a comparison of the solution quality obtained using the different algorithms for the problems tested in Figure 5(a). X-axis denotes the solution quality while Y-axis indicates the approximation parameter, ϵ. Since GOA, SPIDER, and SPIDER-Abs are all global optimal algorithms, the solution quality is the same for all those algorithms. VAX-1 and VAX-2 indicate the VAX algorithm for two different approximation parameters. Even though set a high ϵ we obtained an actual solution quality that was close to the optimal solution quality. In 3-chain and 4-star configurations both VAX-1 and VAX-2 have almost the same quality as the global optimal algorithms. For other configurations as well, the loss in quality is less than 15% of the optimal solution quality.

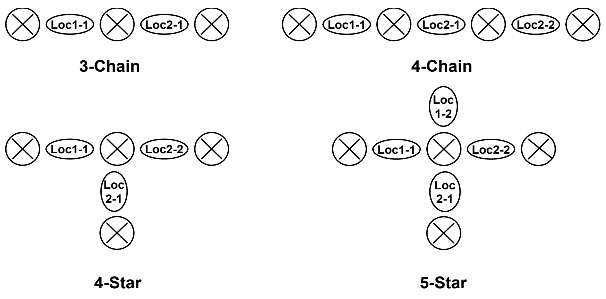

Figure 4: Sensor network configurations

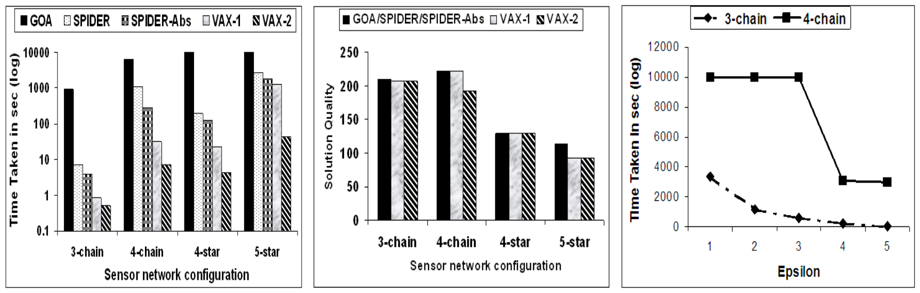

Figure 5: Comparison of GOA, SPIDER, SPIDER-Abs and VAX for two different epsilons on (a) Runtime and (b) Solution quality; (c) Time to solution for VAX with varying epsilon for T=4

Figure 5(c) provides the time to solution for 3-chain and 4-chain configurations with VAX (for varying epsilons). X-axis denotes the approximation parameter, ϵ used, while Y-axis denotes the time taken to compute the solution (on a log-scale). The time horizon for both configurations was 4. As epsilon is increased, the time to solution decreases drastically. For instance, in the 3-chain case there was a total speedup of 50-fold when the epsilon was changed from 1 to 5. One interesting aspect with this result is that this speedup is achieved without any loss in solution quality (remains at 261 for all the five epsilons). This was the case with both the configurations, the quality remained the same irrespective of the approximation parameter.

6 Related Work

Researchers have typically employed two types of techniques for solving distributed POMDPs. The first set of techniques compute global optimal solutions. Hansen*et al.* [6] and Szer *et al.* [14] provide techniques that compute globally optimal solutions without restricting the domain. Hansen *et al.* present an algorithm for solving partially observable stochastic games (POSGs) based on dynamic programming and iterated elimination of dominant policies. Szer *et al.* [14] provide an optimal heuristic search method for solving Decentralized POMDPs with finite horizon (given a starting belief point). This algorithm is based on the combination of a classical heuristic search algorithm, A* and decentralized control theory. The key difference between SPIDER and MAA* is that while SPIDER improves the joint policy one agent at a time, MAA* improves it one time step at a time (simultaneously for all agents involved). Furthermore, MAA* was illustrated only with two agents, while in our case we show results with more than 2 agents.

The second set of techniques seek approximate policies. Emery-Montemerlo *et al.* [4] approximate POSGs as a series of one-step Bayesian games using heuristics to find the future discounted value for actions. This algorithm trades off limited lookahead for computational efficiency, resulting in policies that are locally optimal with respect to the selected heuristic. We have earlier discussed Nair *et al.* [10]'s JESP algorithm that uses dynamic programming to reach a local optimal for finite horizon decentralized POMDPs. Researchers have also concentrated on policy search techniques for obtaining locally optimal solutions. Peshkin *et al.* [12] and Bernstein *et al.* [2] are examples of such techniques that search for local optimal policies. Interactive POMDP (I-POMDP) model by [5] is presented as an alternative to the distributed POMDP model and particle filters have been proposed to solve them. Though all the above techniques have improved the efficiency of policy computation considerably, they are unable to provide error bounds on the quality of the solution. This aspect of quality bounds differentiates SPIDER from all the above techniques.

References

[1] R. Becker, S. Zilberstein, V. Lesser, and C.V. Goldman. Solving transition independent decentralized Markov decision processes. *JAIR*, 22:423–455, 2004.

[2] D. S. Bernstein, E.A. Hansen, and S. Zilberstein. Bounded policy iteration for decentralized POMDPs. In *IJCAI*, 2005.

[3] D. S. Bernstein, S. Zilberstein, and N. Immerman. The complexity of decentralized control of MDPs. In *UAI*, pages 32–37, 2000.

[4] R. Emery-Montemerlo, G. Gordon, J. Schneider, and S. Thrun. Approximate solutions for partially observable stochastic games with common payoffs. In *AAMAS*, pages 136–143, 2004.

[5] P. Gmytrasiewicz and P. Doshi. A framework for sequential planning in multiagent settings. *Journal of Artificial Intelligence Research*, 24:49–79, 2005.

[6] E. Hansen, D. Bernstein, and S. Zilberstein. Dynamic programming for partially observable stochastic games. In *AAAI*, pages 709–715, 2004.

[7] V. Lesser, C. Ortiz, and M. Tambe. *Distributed sensor nets: A multiagent perspective*. Kluwer, 2003.

[8] R. Maheswaran, M. Tambe, E. Bowring, J. Peaerce, and P. Varakantham. Taking dcop to the real world : Efficient complete solutions for distributed event scheduling. In *AAMAS*, 2004.

[9] P. J. Modi, W. Shen, M. Tambe, and M. Yokoo. An asynchronous complete method for distributed constraint optimization. In *AAMAS*, pages 161–168, 2003.

[10] R. Nair, D. Pynadath, M. Yokoo, M. Tambe, and S. Marsella. Taming decentralized POMDPs: Towards efficient policy computation for multiagent settings. In *IJCAI*, pages 705–711, 2003.

[11] R. Nair, P. Varakantham, M. Tambe, and M. Yokoo. Networked distributed POMDPs: A synthesis of distributed constraint optimization and POMDPs. In *AAAI*, pages 133–139, 2005.

[12] L. Peshkin, N. Meuleau, K.-E. Kim, and L. Kaelbling. Learning to cooperate via policy search. In *UAI*, pages 489–496, 2000.

[13] Adrian Petcu and Boi Faltings. A scalable method for multiagent constraint optimization. In *IJCAI*, 2005.

[14] D. Szer, F. Charpillet, and S. Zilberstein. MAA*: A heuristic search algorithm for solving decentralized POMDPs. In *IJCAI*, 2005.