Task-Based Decomposition of Factored POMDPs

Guy Shani
Information Systems Engineering
Ben Gurion University
Israel
shanigu@bgu.ac.il

***Abstract*—Recently, POMDP solvers have shown the ability to scale up significantly using domain structure, such as factored representations. In many domains the agent is required to complete a set of independent tasks. We propose to decompose a factored POMDP into a set of restricted POMDPs over subsets of task relevant state variables. We solve each such model independently, acquiring a value function. The combination of the value functions of the restricted POMDPs is then used to form a policy for the complete POMDP. We explain the process of identifying variables that correspond to tasks, and how to create a model restricted to a single task, or to a subset of tasks. We demonstrate our approach on a number of benchmarks from the factored POMDP literature, showing that our methods are applicable to models with more than 100 state variables.**

***Index Terms*—POMDP, Point-based Algorithms, Factored POMDP**

I. Introduction

Partially Observable Markov Decision Processes (POMDPs) are widely used to model agents acting in a stochastic environment under partial observability. Exact solution algorithms for POMDPs can handle only small state spaces, but approximate solution methods, and in particular, the family of point-based methods [13], generate good, approximate policies for larger domains.

In many cases, it is natural to describe the state of the environment within a POMDP using a set of *state variables*, and the effects of actions in terms of their effects on these variables. Dynamic Bayesian Networks (DBNs) with conditional probability tables (CPTs) in the form of decision-trees or graphs are often used to represent these effects compactly. This representation is known as a *factored* representation. The state space of such models is exponential in the number of variables, and quickly grows outside the reach of methods that operate on an explicit, flat representation, including point-based methods. To address this problem, researchers suggested the use of Algebraic Decision Diagram (ADDs) [3], [9], [14], [10], [17]. Recently [15], it was shown how point-based algorithms that use ADDs as the underlying representation, can solve domains with up to 25 binary state variables, given sufficient domain structure. However, it is likely that real world domains will have many more variables.

In some domains, an agent is required to complete a set of tasks. For example, in the RockSample domain (Figure 1 [19]), a robot needs to sample a number of rocks. We can treat the sampling of each rock as an independent task. In the well known Logistics domain (adapted to POMDPs in [15]) the agent needs to distribute a set of packages to their correct destinations. We can think of the delivery of each package as a separate task. In this paper we restrict the discussion to domains where tasks are independent, in that no task provides a necessary precondition for the fulfilment of another tasks. Therefore, the agent can choose any ordering of the tasks, or make progress on several tasks concurrently. For example, the Logistics agent can load multiple packages on the same truck.

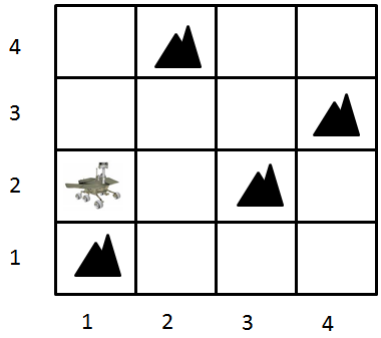

Fig. 1. The RockSample example, with a 4×4 board and 4 rocks.

Even though the tasks are independent, the optimal policy must consider all incomplete tasks at each step. In the RockSample domain, for example, planning a minimal route between the rocks, cannot be done by always greedily moving to the closest rock. Therefore, the agent has to consider the location of all rocks before deciding on the optimal path.

However, considering only a subset of the tasks can also provide a valid solution for the POMDP. We can decompose the POMDP into a set of smaller models (which we call *restricted models*), each modeling only a small number of tasks. These models, which are exponentially smaller than the original one, are much easier to solve. Then, we can compose the policies of the smaller models into a policy for the complete POMDP. Such a policy will be suboptimal, as we did not consider all tasks together. Yet, the solution will be valid, in that all tasks can be completed following the resulting policy.

We focus our attention here on factored POMDPs where to complete each task the agent needs to consider only a small subset of the state variables. For example, in the RockSample domain, to move and sample a rock, the agent needs only consider the X and Y variables, and the rock state variable. In such domains, focusing on a subset of tasks can be done by ignoring all variables which are not necessary for completing

the specific tasks. We can now define a smaller factored POMDP that models only the dynamics of the important variables, and solve it using a point based method, resulting in a value function represented as a set of ADDs.

We repeat the above procedure several times, each time using a different subset of tasks, and a different restricted POMDP. The union of the value functions from the various restricted models provides a policy for the complete POMDP. As the size of the subsets grows, more relationships between tasks are taken into consideration and the resulting policy is closer to the optimal policy of the complete POMDP.

In this paper we explain the procedures for identifying the set of task relevant variables, creating the restricted POMDPs, and combining the resulting value functions. We experiment with two factored POMDP benchmarks, showing how the quality of the resulting policy becomes closer to the policy computed for the complete POMDP. We continue to demonstrate the scalability of our approach by providing solutions for POMDPs with more than 100 binary state variables.

II. Background

A. MDPs and POMDPs

A Markov Decision Process (MDP) is a tuple $\langle S, A, tr, R\rangle$ where S is a set of world states, A is a set of actions, $tr(s, a, s')$ is the probability of transitioning from state s to state s' using action a, and $R(s, a)$ defines the reward for executing action a in state s.

A Partially Observable Markov Decision Process (POMDP) is a tuple $\langle S, A, tr, R, \Omega, O, b_0\rangle$ where S, A, tr, R compose an MDP, Ω is a set of observations and $O(a, s, o)$ is the probability of observing o after executing a and reaching state s. In this paper we focus on finite state, action, and observation spaces.

The agent maintains a *belief-state* — a vector b of probabilities such that $b(s)$ is the probability that the agent is currently at state s. b_0 defines the initial belief state — the agent belief over its initial state. The transition from belief state b to belief state b' using action a is deterministic given an observation o and defines the τ transition function. We denote $b' = \tau(b, a, o)$ where:

$$b'(s') = \frac{O(a, s', o)\sum_s b(s)tr(s, a, s')}{pr(o|b, a)} \quad (1)$$

$$pr(o|b, a) = \sum_s b(s)\sum_{s'} tr(s, a, s')O(a, s', o) \quad (2)$$

An agent operating in an environment described as an MDP or a POMDP typically tries to optimize some function of the stream of rewards it achieves. We focus here on optimizing the infinite stream of discounted rewards $\sum_{t=0}^{\infty}\gamma^t r_t$, where r_t is the reward received at time t and $\gamma \in (0, 1)$ is a discount factor.

B. Value Functions for POMDPs

The value function V for the POMDP can be approximated arbitrarily closely as a finite collection of $|S|$-dimensional vectors known as α vectors [18]. A policy over the belief space is defined by associating an action a with each vector α, so that $\alpha \cdot b$ represents the value of taking a in belief state b and following the policy afterwards. Given a value function represented as a set V of α vectors, the policy π_V is derivable using:

$$\pi_V(b) = \text{argmax}_{a:\alpha_a \in V}\, \alpha_a \cdot b \quad (3)$$

where $\alpha_a \cdot b$ is the inner product (or dot product) of vectors:

$$\alpha \cdot b = \sum_s \alpha(s)b(s) \quad (4)$$

The value function can be iteratively computed

$$V_{n+1}(b) = \max_a[b \cdot r_a + \gamma \sum_o pr(o|a, b)V_n(\tau(b, a, o))] \quad (5)$$

where $r_a(s) = R(s, a)$. The computation of the next value function $V_{n+1}(b)$ out of the current V_n (Equation 5) is known as a *backup* step, and can be efficiently implemented [13] by:

$$backup(b) = \text{argmax}_{g_a^b:a\in A}\, b \cdot g_a^b \quad (6)$$

$$g_a^b = r_a + \gamma \sum_o \text{argmax}_{g_{a,o}^{\alpha}:\alpha\in V}\, b \cdot g_{a,o}^{\alpha} \quad (7)$$

$$g_{a,o}^{\alpha}(s) = \sum_{s'} O(a, s', o)tr(s, a, s')\alpha^i(s') \quad (8)$$

Updating V over the entire belief space, hence computing an optimal policy is computationally hard. A possible approximation is to compute an optimal value function over a subset of the belief space [13]. Such a value function is only an approximation of a full solution, but will hopefully generalize well to other belief states. Point-based algorithms choose a subset B of the belief space, reachable from b_0, and compute a value function only over these belief points using point-based backups (Equation 6).

Point-based algorithms [13], [21], [19], [16] all use a set of basic operations, and differ by the way they select the subset of belief states, and by the order by which backup operations are executed.

C. Factored POMDPs

Traditionally, the MDP/POMDP state space is defined by a set of states, which we call a flat representation. For many problems, however, it is natural to define a set of state variables $X = \{X_1, ..., X_n\}$ such that each state $s =< x_1, ..., x_n >$ is an assignment for the state variables [3]. The transition function $tr(s, a, s')$ is replaced by the distribution $pr(X_i'|X, a)$[1].

We can represent state transitions using a dynamic Bayesian network (DBN) for each action, modeling the transition relationships between variables. The transition probabilities are specified using conditional probability tables (CPTs), that can be coded as decision trees. The reward and observation functions can also be represented using decision trees, where the leaves are either probabilities (for observations) or values (for rewards).

An Algebraic Decision Diagram [1] is an extension of Binary Decision Diagrams, that can be used to compactly represent decision trees. A decision tree can have many identical subtrees, and an ADD unifies these subtrees, resulting

[1]We follow Boutilier and Poole in denoting the pre-action variables by X and the post-action variables by X'.

in a rooted DAG rather than a tree. A variable is missing from a path from the root to a value leaf if it does not influence the value. We can say that the function is indifferent to the value of that variable given the path. The ADD representation becomes more compact as the problem becomes more structured.

Recently, [15] explained how ADDs can be used in point-based algorithms over factored POMDPs. While a straight forward implementation scales poorly, they suggest several special operations for ADDs that allow handling of mid-sized POMDPs.

In a factored RockSample domain [19], [15], for example, an agent moves on a two dimensional map trying to sample a set of rocks that may contain an interesting substance. In this example we need two (multi-valued) state variables to represent the X and Y coordinates for the robot, and a variable for each rock, modeling whether that rock contains the interesting substance or not. Figure 2 shows a part of the DBN defining the transitions and a part of the ADD defining the rewards for this model.

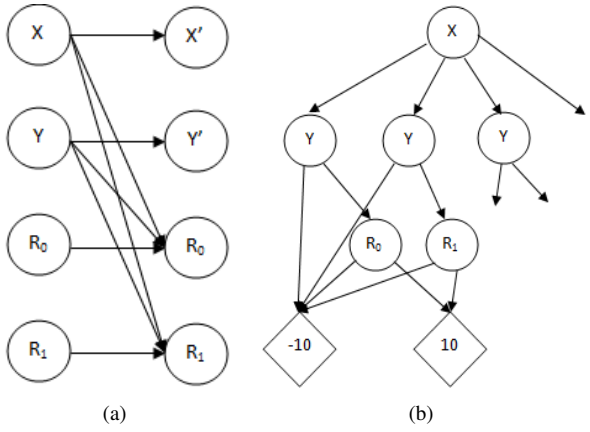

Fig. 2. A part of the transition DBN (a) and the reward ADD (b) for the action Sample. The change in the rock state and the reward depends on the previous state of the rock and on the location of the agent, but not on other rocks. In this figures the variable labels R_i represent the rock variables, rather than rewards.

III. Decomposing Factored POMDPs

Even though ADDs can capture some forms of value function structure efficiently, when the policy is sensitive to the value of all variables, ADDs can no longer represent it compactly. [15] report that the average size of the resulting α-vectors grows significantly with the number of state variables. This is an indication that the optimal policy must capture more variable dependencies.

We suggest to restrict the dependencies between variables, by creating smaller POMDPs that model only subsets of these variables. Therefore, we restrict the maximal size of the resulting ADDs. As the complexity of all point-based operations depends on the α-vector ADD size, policy computation becomes much faster.

Below, we explain how to identify meaningful subsets of state variables, how to decompose the factored POMDP into a set of restricted POMDPs, and how to combine the resulting policies into a policy for the complete POMDP. Our decomposition methods for identifying task-relevant variables can be considered to be an extension of similar ideas for fully observable Markov Decision Processes (MDPs) (see, e.g. [4]).

A. Relevant Variables and Actions for Tasks

In many domains, the agent is required to achieve a goal — an absorbing end state where the execution terminates. Sometimes, a goal is composed of several *tasks* — subgoals that the agent should achieve. A task could not be unmade — once the agent completes a task, it remains completed for the rest of the execution. Upon achieving all the tasks, the goal is achieved and the execution terminates.

Our method is applicable to domains where the agent must perform a set of independent tasks $\{T_i\}$. We say that a task is independent if it does not depend on completing another task as a precondition. We further assume that to complete a task, the agent must consider only a small subset of the state variables — the *relevant* variables for the task. We denote the set of relevant variables for task T by ξ_T. To accomplish a task, we also need to define A_T — the set of relevant actions that influence the values of the variables in ξ_T.

While in many cases it is easy to identify tasks and relevant variables for the tasks using domain knowledge, it is also possible to do so by considering the domain dynamics (transitions and rewards). We assume here that a successful completion of a task is followed by a positive reward. In other cases, e.g., in cost-based POMDPs [2], a successful completion may be more difficult to identify, and may perhaps be identified by a specific value set for some state variable. In this case, we may require that the sub-tasks will be identified manually.

We start by looking at the reward function of the factored POMDP — $R(< X_1, ..., X_n >, a)$. By traversing the minimal ADD representation of the reward, we can find paths that lead to positive rewards. The set of variables that are specified on the path to a positive reward $X_{r_0}, ..., X_{r_k}$ are the only variables that are relevant for the reward. That is, for each X_i such that $X_i \neq X_{r_j}$:

$$R(< X_{r_0}, ..., X_{r_k}, X_i >, a) = R(< X_{r_0}, ..., X_{r_k} >, a) \quad (9)$$

for any possible value of X_i.

For each such path to a positive reward r in a reward ADD we define a task T_r. We initialize the relevant variables ξ_{T_r} to the variable set $\{X_{r_0}, ..., X_{r_k}\}$. We initialize A_T to the action that generated the reward r.

We now look at the transition probabilities, which are described using a DBN for each action a. We identify each such DBN that influences a variable X_i in ξ_T. That is, a DBN where X_i appears in its influenced variables. Then, we add all the parents of X_i in the DBN to ξ_T. We add the action associated with the DBN to the set of relevant actions A_T. We repeat this process until the sets ξ_T and A_T do not grow anymore.

As the sets of relevant variables and relevant actions define the task, we will write $T = \{\xi_T, A_T\}$. After following the process above for each reward r we may find that some tasks are subsets of other tasks — $T_1 \subset T_2$, that is, $A_{T_1} \subset A_{T_2}$ and $X_{T_1} \subset X_{T_2}$. In that case, we remove T_1.

Algorithm 1 DiscoverRelevantVariables(M)

Input: A factored POMDP M
Output: A set of tasks $\mathcal{T}$

1: $\mathcal{T} = \phi$
2: **for each** action $a \in A$ **do**
3: **for each** path $< X_{r_0}, ..., X_{r_k} > \in R(< X_1, ..., X_n >, a)$ ending in a positive reward **do**
4: $\xi_T \leftarrow \{X_{r_0}, ..., X_{r_k}\}$
5: $A_T \leftarrow \{a\}$
6: **repeat**
7: **for each** action $a' \in A$ **do**
8: **for each** variable X_i not in ξ_T **do**
9: **if** X_i is a parent of any $X_j \in \xi$ in $pr(X'_j|X, a')$ **then**
10: Add X_i to ξ_T
11: Add a' to A_T
12: **until** ξ doesn't change
13: Add $\{\xi_T, A_T\}$ to $\mathcal{T}$
14: **for each** task $\{\xi_T, A_T\} \in \mathcal{T}$ **do**
15: **if** $\exists\{\xi'_T, A'_T\} \in \mathcal{T}$ such that $\xi_T \subset \xi'_T$ and $A_T \subset A'_T$ **then**
16: Remove $\{\xi_T, A_T\}$ from $\mathcal{T}$
17: **return** $\mathcal{T}$

Algorithm 1 describes the identification of the tasks and the relevant variables for each task. In line 3, we identify paths that lead to positive rewards. While the number of such paths can be exponential in the number of state variables, this is rarely the case, because an exponential number of paths results from little structure in the state space, making a factored representation unsuitable. When searching for these paths in practice, we start from the positive reward leaves, and move up towards the parents, collecting the paths as we go.

Using some preliminary pre-processing, we can also collect for each variable X_i the set of variables that are affecting it, i.e. the set of variables X_j s.t. X_j is a parent of X_i in any $pr(X_j|X, a)$. This requires a single pass over all the ADDs describing the model transitions. Using this cached data, line 9 becomes trivial to compute.

As such, Algorithm 1 is very fast in practice. In all the domains that we experiment with, it takes a fraction of a second to compute Algorithm 1.

For example, in the RockSample example, looking at the reward ADD in Figure 2(b), we will begin with the positive leaf reward node 10. Moving upwards, we will find the paths X, Y, R_i for all rocks R_i. We will then observe in the transition ADD (Figure 2(a)) that X, Y, R_i do not have additional parents, i.e. they are not affected by any other variable. Thus, in this case, $\mathcal{T} = \{\xi_i, A_i\}$ where $\xi_i = \{X, Y, R_i\}$ and $A_i = \{MoveUp, MoveDown, MoveLeft, MoveRight, Check_i, Sample\}$ is the set of all movement actions, the action used to remotely sense whether rock i is good, and the rock sampling action.

B. Defining the Restricted Models

Given a factored POMDP $M =< X, A, tr, R, O, \Omega, b_0 >$, which we call the *complete* POMDP, and a task $T = \{\xi_T, A_T\}$, we define $M_T =< \xi_T, A_T, tr, R, O, \Omega, b_{0_T} >$ — the factored POMDP model restricted to the task T.

The set of variables and the set of actions of the restricted POMDP are taken directly from the task definition. The transition probabilities of M_T are identical to the transition probabilities of M, restricted to the relevant variables and actions.

The observation space Ω and function O are used as in the original model, but are restricted only to the actions that are relevant to the task. Thus, every observation that is relevant for the current task, i.e., that is the result of some action that is relevant to the task, can be achieved in the restricted model.

To define b_{0_T} — the initial belief for the restricted POMDP, we need to compute the start state probabilities over the relevant variables only. Given the original initial belief $b_0 = pr(< x_0, ..., x_m >)$ we define:

$$b_{0_T} = pr(< x_{T_0}, ..., x_{T_k} >) = \sum_{X' \notin T_X} pr(< x_{T_0}, ..., x_{T_k}, X' >). \quad (10)$$

That is, we sum the probabilities over all the possible assignments to the variables that are not in X_T.

We can define a restricted model over a set of tasks by joining their relevant variables and actions sets. A POMDP restricted for a set of tasks models some dependencies between tasks. It may be desirable for a task to participate in a number of restricted models, modeling its dependency on a larger number of tasks. However, even if task T_0 was modeled once together with task T_1 and once with task T_2, it still does not imply that the complete policy properly evaluates T_0 compared to T_1 and T_2 together. Nevertheless, even such limited task dependencies can provide some gain, as we later show.

C. Computing Restricted and Complete Policies

The definition of the restricted models, either for a single task or for a set of tasks, results in a factored POMDP. We therefore continue to solve the restricted POMDPs using the techniques detailed in [15]. We use a point-based algorithm to compute a value function for the restricted POMDP resulting in a set of α-vectors.

We now need to "translate" these α-vectors back to the original POMDP. Here, we use a helpful property of ADDs; In an ADD, a variable does not appear if it is irrelevant for the specified value. As such, the α-vectors resulting from the restricted POMDPs are immediately applicable to the complete POMDP. One way to understand this is by considering these α-vectors as computing the value of policies in the complete POMDP over a restricted set of actions.

When selecting an action to be executed in the complete POMDP, we apply the resulting α-vectors as if they were computed for the complete POMDP, i.e., by applying Equation 3. Thus, the selected action is the one with the highest reward given the true belief state of the complete POMDP, in one of the restricted subtask POMDPs. Hence, there is no policy merging step required, aside for joining all the α-vectors computed for the restricted POMDPs into a single set. This method causes the subtask (or subset of tasks modeled in a single restricted POMDP) with the highest current reward to be accomplished first. For example, in the RockSample domain, with a single task for each restricted POMDP, the closest rock will be handled first, because the values associated with actions for sampling it will have a higher value, given the effect of the discount factor on longer planning horizons.

After the task has been completed, e.g. the rock has been sampled, the belief will change such that the α-vectors associated with that task no longer have a high value. Therefore, the beliefs following the completion of a task, will not prefer these α-vectors, and the agent will not try to re-accomplish the task, e.g. resample the rock. This is done naturally by our model, without any external mechanism for removing the policies of completed tasks from the complete policy.

Therefore, a solution to a set of restricted POMDPs provides a (non-optimal) solution to the complete POMDP. We can define for each task a separate restricted POMDP, compute a policy for that restricted POMDP, and combine the resulting value functions to obtain a value function (and policy) for the complete POMDP.

However, even though achieving each separate task can be done without considering the other tasks, this solution will not be optimal. In the case of a discounted POMDP, we might execute the task in a non-optimal sequence, which will result in a lower expected discounted reward. In a POMDP where each action has some cost (negative reward) associated with it, the restricted POMDPs may cause us to suffer the cost of an action multiple times.

We can reduce this problem by using larger restricted POMDPs over subsets of tasks, thus optimally modeling the relationships between tasks in a subset. However, even if we were to try to model only all pairs of tasks, the number of restricted models will make the solution no faster than solving the original model. Instead, we use a sampling approach.

This suggests an anytime approach, where we solve as many restricted POMDPs as time allows, resulting in a finer solution for the complete POMDP. Also, we can start with restricted POMDPs over smaller subsets of tasks, which are faster to compute, and keep growing the subsets until we reach the full model, if time permits.

Finally, solving the reduced POMDPs can be done in parallel. These restricted models have no dependencies and therefore there is no synchronization needed between the solutions. We can distribute the computation to multiple machines, making this technique useable for large clusters.

IV. Empirical Evaluation

In this section we demonstrate our method over two scalable benchmarks. We first experiment with mid-sized domains, showing how the policy computed for the restricted POMDPs is of similar quality to a policy computed for the complete POMDP. We then move to larger domains, where we cannot hope to compute a policy over the complete POMDP, yet our methods rapidly compute a valid solution. All the experiments below were executed on an 8 core 3 GHz CPU with 16 GB of RAM, using Java 6.0.

A. Domains and Setup

To provide evidence to the applicability of our methods, we experimented with two factored POMDP domains. In the RockSample domain [19], [15], motivated by the Mars Rover, an agent is required to sample a set of rocks in the surrounding environment, and then return the samples to its docking station. To sample a rock the agent must reach its location. A rock may contain an interesting substance or not, and the agent has sensors that can detect the interesting substance from a distance. The accuracy of the sensor diminishes as the distance grows. In this domain, the sampling of a single rock is considered a task. The relevant variables are the state of the rock, and the X and Y coordinates of the agent. The relevant actions are movement actions, sampling the rock and activating the long range sensor towards the specific rock. All tasks share the X and Y variables and the movement and sampling actions, but the rock variable and the sensing actions are task specific.

In the Logistics domain [15] the agent is required to deliver a set of packages. Each package starts at a random city and must reach a predefined destination city. The agent has a number of trucks that can drive between cities. A package can be loaded onto a truck if the package and the truck are at the same city, or unloaded if the package is on the truck. This domain is very stochastic — driving, loading, and unloading may fail. The agent is augmented with noisy ping actions for finding the current location of a truck or a package. In Logistics, the delivery of a single package is considered a task. For each such task the relevant variables are the location of the package, and the location of the trucks. It is also possible to further restrict the POMDP by assigning a single truck for a task. While this does not follow from the relevant variables algorithm we provide above, such a decomposition is obvious using domain knowledge.

In both domains we used a large discount factor $\gamma = 0.995$. Thus, we ensure that it is important to get to the rewards (complete tasks) as fast as possible, but that tasks that are completed late still contribute much to the Average Discounted Reward (ADR).

For modeling factored domains we use the framework of Shani *e*t al. [16] for representing factored POMDPs using

ADDs. Our algorithms extend that framework and use the same underlying ADD functionality. Hence, the "complete" models that we evaluate are simply the execution of the ADD-based point-based algorithms discussed in that paper.

In the experiments below we used the FSVI point-based algorithm [16][2]. As FSVI is an approximation algorithm, the computed policies are not optimal, but FSVI has been shown in the past to provide good policies. When computing the complete policy, our ADD-based FSVI implementation uses equivalent underlying mechanisms for belief update and point-based backup computation as the two other implementations of factored POMDP solvers, Symbolic Perseus [14], and Symbolic HSVI [17].

We experiment with restricted models with increasing number of tasks. When creating models with multiple tasks we use the following process to select subsets of k out of the n tasks; We iterate over the tasks, and for each task we randomly select $k-1$ additional tasks. This process ensures that each task will participate in at least 1 restricted model, and each task is equally likely to participate in multiple models. We hence always use n restricted models, regardless of k.

B. Evaluating Approximation Quality

We begin our experiments with evaluating the approximation quality of our approach. As we have explained above, ignoring some possible dependencies between tasks, will typically cause the resulting policy to be suboptimal. However, we currently cannot analytically bound the loss on policy quality. We therefore experiment with mid-sized instances of the two domains, where the complete problem can still be solved.

As we take a sampling approach in grouping sub-tasks together, we ran 50 iterations of the sampling process above for each value of k (except for $k=1$ and the complete model where no sampling is used). We ran FSVI on the complete model and each combination of restricted models for 500 trials, and computed the average collected discounted reward over these trials.

Table I shows the computed ADR as an indication to the quality of the policy, and the policy computation time. We did not execute a distributed version of the algorithm, where each decomposition is solved on a different machine. However, as this is a natural extension of our method, we report the policy computation time per model, as an estimation of the complete runtime of the entire algorithm in a distributed environment.

When no relationships are being modeled the performance is relatively low. However, even modeling some task relationships gets us very close to the quality of the policy computed for the complete POMDP. This is an indication that for larger problems, where we cannot possibly compute a policy for the complete POMDP, our method still provides good approximations.

In addition we report the size of the combined policy, i.e. the number of α-vectors in the final value function (denoted $|V|$ in Table I). The number of vectors increases with the size of the restricted models, and since we combine the value functions of the restricted models together, the result is much larger than the policy for the complete solution. The required time during execution is directly related to the number of α-vectors, as the current belief state must be checked against each α-vector. That being said, the required inner product operation can be implemented very efficiently [15] and in practice the selection of the best vector takes only a few milliseconds.

[2]There is no particular reason to prefer FSVI for solving the models, and any other offline solution method that generates α-vectors can be used instead.

TABLE I
MEDIUM SIZE DOMAINS — ROCKSAMPLE WITH A 8×8 BOARD AND 8 ROCKS (2^{14} STATES) AND LOGISTICS WITH 4 CITIES, 2 TRUCKS, AND 4 PACKAGES (2^{16} STATES). TIME IS REPORTED IN SECONDS.

RockSample $8,8,8$				
Restriction	ADR	Total time	Time per restricted model	$\lvert V\rvert$
Complete	37.02	160	N/A	337.1
1 rock	24.235	9	1.12	148.5
2 rocks	27.294	32	4	408.1
3 rocks	33.189	87	10.87	750.1
4 rocks	34.46	178	22.25	1077.7
5 rocks	35.141	387	48.37	1470.4
6 rocks	36.725	991	123.875	1805.6
Logistics $4,2,4$				
Restriction	ADR	Total time	Time per restricted model	$\lvert V\rvert$
Complete	18.389	820	N/A	253.2
1 package	8.41	57	14.25	290.8
2 packages	16.4	679	169.75	565.3
1 package, 1 truck	15.34	98	12.25	607.9
2 packages, 1 truck	17.38	748	93.5	1140.1

C. Large Scale Domains

We now move to larger instances of the two domains, where solving the complete POMDP using the techniques in [15] is infeasible. Even the atomic operations of a point-based backup still takes many minutes to complete, making even a single iteration of FSVI impossible. These models are also too large to execute the policies using exact belief updates. Therefore, we use the product of marginals belief approximation from [15]. Thus, the belief states that we use are inaccurate, and when we select an α-vector given the inaccurate belief it is possible that the selected action will not be as good as the action selected given the exact belief state.

Table II shows our results on these models. Comparing the policy computation time per restricted model to the smaller models we experimented with previously, when a single task is used (first line of each domain in Table I), we see that the policy computation time per restricted model is very similar. That is, the number of tasks hardly changes the computation time per restricted model. Therefore, our approach scales linearly with the number of variables, because only the number of restricted models grows, not the amount of time required to solve a restricted model.

The reported total time in Table II includes the time required to define the restricted POMDPs. Although identifying the tasks is very fast, the technical definition of the restricted POMDP may take a few seconds for each model. Of course, this overhead increases with the size of the domain, because

TABLE II
EXPERIMENTS OVER LARGE SCALE DOMAINS USING A SINGLE TASK PER RESTRICTED MODEL.

RockSample 16×16 board				
	$\|S\|$	ADR	Total time	Time per restricted model
20 rocks	2^{28}	38.1	75	3.75
50 rocks	2^{58}	66.171	189	3.78
100 rocks	2^{108}	122.46	614	6.14
Logistics 4 cities 2 trucks				
	$\|S\|$	ADR	Total time	Time per restricted model
10 package	2^{34}	45.147	147	14.7
20 packages	2^{64}	59.212	357	17.85
50 packages	2^{154}	88.945	967	19.34

the complete model ADDs which must be processed become larger, which explains the non-linear growth in total time when the number of tasks increases.

D. Discussion

Our methods are geared towards parallel solution of task subsets. It is apparent from the tables that as the subset size increases, solving all subsets sequentially results in a higher computation cost with a lower solution quality. However, if we solve the subsets in parallel, we only care about the time required for a single subset, which is much less than the time required for a complete solution of the model.

Our results show that the quality of the solution improves as the size of the task subset increases. It might be, though, that smartly selecting the subsets can reduce this phenomena, i.e., that we can still achieve good performance while maintaining a small subset size. It is likely that research in this direction will have the highest added benefit to our approach.

The sub-tasks can be solved in any order, or even in parallel. As we assume task independencies, and the results of one task do not affect other tasks the order by which they are solved is immaterial. One could, however, imagine an anytime variation of our algorithm, where subsets of tasks are solved sequentially, and the accumulated value function is used to produce a policy for the entire model, even before all subsets have been solved. In this variation of the algorithm, we must make sure that as many sub-tasks are included in the earlier subsets. For example, it may be beneficial to start with disjoint subsets, and only after all sub-tasks have been considered at least one, allow repetitions.

Our current algorithm combines the value functions of the sub-tasks into a single value function. When selecting the best α-vector for a given belief state, we essentially select an α-vector that was computed for a single task. We can think of this as choosing a task to solve, and then selecting an action (through an α-vector) that will advance us on the chosen task. A possible concern that may arise here is that the agent may execute a single step towards the completion of task T_1, and then choose a different task T_2. The action that is currently optimal for T_2 may cause some setback in achieving T_1. In our case, however, when tasks are defined through positive rewards, the agent always advances the task whose current expected discounted payoff is highest. An action chosen for that task will (stochastically) result in a belief state with higher expected reward for that task. Thus, as the task T_1 was originally selected because it had the highest expected discounted payoff, its payoff following the action is expected to increase, and it would thus remain the best task to pursue. Indeed, cycles like that do not occur in any of the domains that we experimented with.

Our approach assumes that tasks are defined using positive rewards, which is perhaps the most natural way for a model designer to denote desirable outcomes. The negative rewards do not take part in the sub-task identification, but are defined within each restricted model over the state variables over the model. That is, our algorithm ignores negative rewards in identifying the sub-tasks, but not when computing the policies.

As we have stated above, our methods are applicable only to cases where the agent has achievable tasks, and these tasks are independent, i.e., where there is no particular order on accomplishing tasks. In domains where tasks are related, our method may fail completely, because if n related tasks are not considered together in a single sub-model, then the proper ordering of tasks may not be found. In that case, we can construct cases where the agent fails to achieve some tasks. An interesting extension to our ideas is to discover such task dependencies, through the influence of actions on state variables, and force all related tasks to appear in the same sub-model. This, however, may cause the sub-models to grow to an unsolvable size, making our approach useless. A more interesting method for handling these cases is in changing the initial states and terminal states of some sub-models, to reflect the solution of previous tasks, and the desirable conclusion of the considered tasks. We leave investigation of these ideas to future research.

In contrast to the independence assumption, in many cases tasks are dependent, that is, the completion of task T_1 requires the completion of another task T_2 earlier on. In such cases our algorithm would identify two different tasks where the restricted model for T_1 subsumes the restricted model for T_2. In that case, we would compute one redundant policy (for T_2), and we would have a larger restricted model. In the limit, where all tasks are dependent, the last task would be equivalent to the complete model, rendering our approach useless. One can imagine a pre-processing phase where task ordering is discovered and then restricted models that rely on task completion are constructed without containing the relevant variables for the previous tasks, but we leave this for future research.

Another desirable extension to our method would be to apply some reuse of already computed sub-tasks. For example, in relational MDPs [5] allow us to model exchangeable items efficiently. For example, the goal may be to get a red package to the destination city, as opposed to package number 5. It may be possible to use ideas from relational MDPs in order to extend our approach to "translate" policies that were learned for one subset of tasks to another subset of tasks. For example, if we learned a policy for one package whose destination is city 2, we can apply the same policy to all other packages whose destination is city 2, reducing the number of restricted POMDPs that we solve. Such extensions require us to discover some symmetry in the task definition and leverage

this symmetry. In relational models, this symmetry is a part of the model definition, but in general POMDPs, discovering such properties may be difficult. In some cases a reward can be repeatedly achieved, even though the model could be factored into sub-models. For example, in the network administration domain [14], [15] an administrator must keep a network of computers running, by restarting malfunctioning machines. In this example, the administrator gets a higher reward for maintaining one machine (a server) than maintaining the other machines. In such a case, although the model can be factored by our algorithm into separate tasks, a task never ends. Thus, in the resulting combined policy the policy designed to keep the server running is always dominant, and all other computers are ignored, leading to very bad performance.

Our approach currently handles a factored state space with non-factored (flat) observations. In many cases a factored observation model is also useful (e.g. [23]). Imagine, for example, in the RockSample domain, a single sensing action that returns a k bit observation vector, where k is the number of rocks, and when the ith bit is on, then the sensors senses the mineral in rock i. In that case, we can find out that the observation over rock i is influenced only by variable R_i (denoting the rock state), and restrict the model to contain only that observation variable. Although this seems like a natural extension of our algorithm, we leave its actual implementation to future work.

V. Related Work

The decomposition of models for planning under uncertainty has been discussed in the past. Perhaps the most studied approach to decomposition is to identify *regions* — clusters of states — compute a policy within a region and compute a meta-policy for moving between regions (see, e.g. [3], [12]).

Another approach to decomposition is by adding *activities* — actions that accomplish a task, and learning how to execute each activity independently. Then, we can learn a high-level policy that switches between activities [11]. Such approaches accomplish one task at a time.

An alternative to learning value functions represented through α-vectors is to learn so called Finite State Controllers [8]. In this context, several researchers suggested to learn hierarchies of controllers [7], [22]. These hierarchies can capture structure in a natural way.

All the above methods do not share the restrictions of our approach. As we have explained, we are interested in domains where there is no required order by which tasks must be executed, as no task provides a required precondition for another task. The above methods are all designed to handle the more difficult case, where tasks are dependent, and yet a decomposition is required in order to tackle large models. Therefore, most of the methods above require significant effort for the decomposition phase. We are unaware of any POMDP decomposition method that has shown the ability to scale up to the model sizes we experimented with.

A different method for scaling up POMDPs is by compression. [14] have suggested a value directed compression that creates a smaller, solvable, model, showing an impressive ability to scale up to huge models. The compression techniques they use, however, eliminates the factored representation — that is, the compressed model is no longer a factored POMDP. This property may induce a difficulty when trying to scale up even more. Our approach keeps the restricted models factored.

[20] has also suggested a compression technique for factored POMDPs. Like us, they are also interested in a subclass of POMDPs. Specifically, they are interested in domains where variables are important only during a limited part of the policy execution. The result of their compression is, again, non factored. This line of research demonstrates that for subclasses of POMDPs, we can exploit the properties of the domain to achieve a significant improvement.

The idea of *masked* α-vectors [19] also bears some resemblance to our approach. Masked α-vectors obtain a non-zero value only in a subset of states. Our approach is more generic, however, because while we limit the values only to a subset of variables, the absence of a variable does not result in a value of zero.

Another approach to solving POMDPs also shares interesting relationships with our method. [6] suggest to solve large domains by limiting the value function to be a composition of *basis functions*. Our approach also creates value functions of limited structure because our α-vectors are limited to subsets of variables. However, we do not impose further restrictions on the structure of the α-vectors and we do not need to discover good basis functions.

VI. Conclusion and Future Work

This paper suggests a novel method for the decomposition of factored POMDPs, focusing on domains where the agent needs to complete multiple independent tasks. Our method creates restricted models over subsets of state variables. We explain how to identify state variables that must be grouped together, and how to create models that capture some of the task dependencies. Our methods can be used in an anytime setting, where a policy refinement is constantly computed, and in a distributed setting, where each restricted model is solved on a different machine without synchronization.

We experimented with two domain benchmarks, showing that our methods produce policies with reasonable quality within seconds for mid-size domains. We also showed results on computing policies for large scale models with $2^{154} = 2.2 \times 10^{46}$ states in about 16 minutes. Still, our methods can scale up to much larger domains.

In the future, we intend to explore informed methods for combining tasks together. Currently, we sample subsets of tasks uniformly. However, it is likely that a smart selection of subsets of tasks will create better policies. Also, we intend to continue experimenting with larger domains and with distributed architectures. Another problem that we currently face is the large number of vectors that we compute. In the future, we plan to reduce this overhead by combining vectors that suggest the same action or by pruning unneeded vectors.

References

[1] R. I. Bahar, E.A. Frohm, C.M. Gaona, G.D. Hachtel, E. Macii, and A. Pardo. Algebraic Decision Diagrams and Their Applications. In *International Conference on CAD*, pages 188–191, 1993.

[2] B. Bonet and H. Geffner. Solving POMDPs: RTDP-Bel vs. Point-based algorithms. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1641–1646, 2009.

[3] C. Boutilier and D. Poole. Computing optimal policies for partially observable decision processes using compact representations. In *AAAI-96*, pages 1168–1175, 1996.

[4] Craig Boutilier and Richard Dearden. Using abstractions for decision-theoretic planning with time constraints. In *In Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 1016–1022, 1994.

[5] C. Guestrin, D. Koller, C. Gearhart, and N. Kanodia. Generalizing plans to new environments in relational MDPs. In *IJCAI-03*, 2003.

[6] C. Guestrin, D. Koller, and R. Parr. Solving factored POMDPs with linear value functions. In *IJCAI workshop on Planning under Uncertainty and Incomplete Information*, 2001.

[7] E. Hansen and R. Zhou. Synthesis of hierarchical finite-state controllers for POMDPs. In *ICAPS-03*, 2003.

[8] E. A. Hansen. Solving POMDPs by searching in policy space. In *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence*, pages 211–219, 1998.

[9] E. A. Hansen and Z. Feng. Dynamic programming for POMDPs using a factored state representation. In *Artificial Intelligence Planning Systems*, pages 130–139, 2000.

[10] J. Hoey, A. von Bertoldi, P. Poupart, and A. Mihailidis. Assisting persons with dementia during handwashing using a partially observable Markov decision process. In *International Conference on Vision Systems (ICVS)*, 2007.

[11] A. Jonsson and A. Barto. Causal graph based decomposition of factored MDPs. *Journal of Machine Learning Research*, 7:2259–2301, 2006.

[12] S. Mannor, I. Menache, A. Hoze, and U. Klein. Dynamic abstraction in reinforcement learning via clustering. In *ICML-04*, 2004.

[13] J. Pineau, G. Gordon, and S. Thrun. Point-based value iteration: An anytime algorithm for POMDPs. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1025 – 1032, August 2003.

[14] P. Poupart. *Exploiting Structure to Efficiently Solve Large Scale POMDPs*. PhD thesis, University of Toronto, 2005.

[15] G. Shani, R. Brafman, P. Poupart, and S. Shimony. Efficient ADD operations for point-based algorithms. In *ICAPS-08*, 2008.

[16] G. Shani, R. Brafman, and S. Shimony. Forward search value iteration for POMDPs. In *IJCAI-07*, 2007.

[17] H. S. Sim, K. Kim, J. H. Kim, D. Chang, and M. Koo. Symbolic heuristic search value iteration for factored POMDPs. In *AAAI-08*, pages 1088–1093, 2008.

[18] R. Smallwood and E. Sondik. The optimal control of partially observable processes over a finite horizon. *OR*, 21:1071–1088, 1973.

[19] T. Smith and R. Simmons. Point-based POMDP algorithms: Improved analysis and implementation. In *UAI 2005*, 2005.

[20] T. Smith, D. R. Thompson, and D. Wettergreen. Generating exponentially smaller POMDP models using conditionally irrelevant variable abstraction. In *ICAPS-07*, 2007.

[21] M. T. J. Spaan and N. Vlassis. Perseus: Randomized point-based value iteration for POMDPs. *JAIR*, 24:195–220, 2005.

[22] M. Toussaint, L. Charlin, and P. Poupart. Hierarchical POMDP controller optimization by likelihood maximization. In *UAI-08*, 2008.

[23] Z. Zamani, S. Sanner, P. Poupart, , and K. Kersting. Symbolic dynamic programming for continuous state and observation POMDPs. In *26th Annual Conference on Advances in Neural Information Processing Systems (NIPS-12)*, 2012.