Solving Stochastic Planning Problems With Large State and Action Spaces

Thomas Dean, Robert Givan, and Kee-Eung Kim

Thomas Dean and Kee-Eung Kim
Department of Computer Science
Brown University
Providence, RI 02912
(401) 863-7600
{tld,kek}@cs.brown.edu
http://www.cs.brown.edu/~{tld,kek}

Robert Givan
Department of Electrical and Computer Engineering
Purdue University
West Lafayette, IN 47907
(765) 494-9068
givan@ecn.purdue.edu
http://dynamo.ecn.purdue.edu/~givan

Abstract

Planning methods for deterministic planning problems traditionally exploit factored representations to encode the dynamics of problems in terms of a set of parameters, e.g., the location of a robot or the status of a piece of equipment. Factored representations achieve economy of representation by taking advantage of structure in the form of dependency relationships among these parameters. In recent work, we have addressed the problem of achieving the same economy of representation and exploiting the resulting encoding of structure for *stochastic* planning problems represented as Markov decision processes. In this paper, we extend our earlier work on reasoning about such factored representations to handle problems with large action spaces that are also represented in factored form, where the parameters in this case might correspond to the control parameters for different effectors on a robot or the allocations for a set of resources. The techniques described in this paper employ factored representations for Markov decision processes to identify and exploit regularities in the dynamics to expedite inference. These regularities are in the form of sets of states (described for example by boolean formulas) that behave the same with respect to sets of actions where these sets are thought of as aggregate states and aggregate actions respectively. We present theoretical foundations, describe algorithms, provide examples in which our techniques provide leverage and examples in which they fail to do so, and summarize the results of experiments with a preliminary implementation.

1. Introduction

The methods developed in this paper extend specific planning algorithms [Boutilier *et al.*, 1995; Dean & Givan, 1997] developed for handling large state spaces to handle large action spaces. The basic approach involves reformulating a problem with large state and action spaces as a problem with much smaller state and action spaces. These methods are particularly effective in cases in which there are a large set of possible actions allowed in each state, but only a small number of actions are likely to have an impact on the bottom line, *e.g.,* achieving a specified goal or minimizing costs.

In some problems, *e.g.,* particularly difficult scheduling problems, it is not easy to identify actions that can be eliminated from consideration. In other problems, such identification is easy. For example, a robot in traversing a darkened room can adjust its camera (a passive sensor) in a variety of ways, but only the sonar and infrared devices (active sensors) will have any impact on the robot's success in navigating the room. The techniques described in this paper reformulate planning problems in a preprocessing step so as to eliminate such actions from consideration in appropriate contexts, *i.e.,* sets of states.

This paper is organized as follows: We begin by introducing and formalizing a class of planning problems cast in terms of Markov decision processes. We consider methods for compactly representing Markov decision processes with large state and action spaces. We then investigate reformulation methods that allow us to solve Markov decision processes without explicitly quantifying over all states and actions. We describe model minimization techniques as a means of automatically performing such problem reformulations. In an appendix we describe a variant of a standard algorithm (value iteration) that performs the requisite problem reformulations on the fly. Finally, we explore some sample problems and summarize preliminary experiments.

2. Sequential Decision Problems

We are interested in solving *sequential decision problems* in which we are given the dynamics of a (possibly stochastic) environment and an objective function specifying the value of outcomes corresponding to sequences of states, and we are asked to generate a behavioral specification (actions to perform in different states or times) that satisfies some requirement in terms of the objective function.

Markov Decision Processes

Specifically, we consider the case of Markov decision processes (MDPs) which generalize on propositional STRIPS planning problems. Let $M = (Q, A, T, R)$, where Q is a finite set of *states,* A is a finite set of *actions,* T is a *state-transition distribution, and* R is a *reward function.* Transitions and rewards are defined so that $\forall i, j \in Q, \alpha \in A$,

$$T(i, \alpha, j) = \Pr(X_{t+1} = j | X_t = i, U_t = \alpha)$$
$$R(i, \alpha) = \text{Reward}(X_t = i, U_t = \alpha) \tag{1}$$

where the random variables X_t and U_t denote the state of the system at time t, and the action taken at time t, respectively.

We define a set of *policies* $\pi \in \Pi$ where $\pi: Q \to A$. A Markov decision *problem* is an MDP together with an *objective function*; in this paper we restrict our attention to one particular objective function, expected cumulative discounted reward with discount rate $0 < \gamma < 1$. A candidate solution to a Markov decision problem is a policy. To compare policies we employ the notion of a *value function* $v \in V$ where $v: Q \to \Re$. The value function for a fixed policy π is defined by

$$v_\pi(i) = R(i) + \gamma \sum T(i, \pi(i), j) v_\pi(j) \tag{2}$$

Here and elsewhere we assume that the rewards do not depend on the action to keep the notation a little simpler. The optimal policy is defined by the *optimal value function*

$$v^*(i) = \max_\pi v_\pi(i) \tag{3}$$

Occasionally we use vector notation in which $v(i)$ denotes the i th component of v.

Solving Markov Decision Problems

The system of (*Bellman*) equations has a unique solution v^*

$$v(i) = R(i) + \max_\pi \gamma \sum T(i, \pi(i), j) v(j) \tag{4}$$

The *nonlinear operator* L defined by

$$Lv(i) = R(i) + \max_\pi \gamma \sum T(i, \pi(i), j) v(j) \tag{5}$$

is a *contraction mapping, i.e.,* $\exists (0 \le \lambda < 1)$ *s.t.* $\forall (u, v \in V)$

$$\|Lu - Lv\| \le \lambda \|u - v\| \quad \text{where} \quad \|v\| = \max_i |v(i)| \tag{6}$$

with *fixed point* v^*. L is the basis for *value iteration*, an iterative algorithm which converges to the optimal value function. In the following, we focus on value iteration because it is conceptually simple, but the techniques that we discuss apply to other MDP solution methods. The basic step in value iteration, called a *Bellman backup,* corresponds to applying the operator L and can be performed in time polynomial in the size of Q and A. The challenge is handling the case in which Q and A are exponentially large relative to the size of the problem description. Before we address this challenge, we discuss how we can even represent such problems compactly.

3. Factorial Models

Instead of assuming explicitly enumerated sets of states, we assume that states are represented in terms of J state variables, $\{X_i\}$, often called *fluents*. Similarly we assume that actions can be described in terms of a set of K control parameters, $\{U_i\}$, or *action fluents*. The state and action spaces are then defined in terms of products over sets of values for the parameters describing the states and actions.

$$A = \prod_{i=1}^{K} \Omega_{U_i} \qquad \vec{U} = (U_1, \dots, U_K)$$
$$Q = \prod_{i=1}^{J} \Omega_{X_i} \qquad \vec{X} = (X_1, \dots, X_J) \tag{7}$$

where Ω_X denotes the set of possible values for the parameter X. We typically assume that $\Omega_X = \{\text{True}, \text{False}\}$ to simplify presentation, but the techniques apply more generally. Using this *factored* representation for states and actions we can define rewards and transitions

$$R(i, \alpha) = R'(X_1, \dots, X_J, U_1, \dots, U_K)$$
$$T(i, \alpha, j) = T'(X_1, \dots, X_J, U_1, \dots, U_K, X_1, \dots, X_J) \tag{8}$$

Example Factored Representation

Figure 1.a shows a Bayesian network that represents the dependencies among state and action parameters. Bayesian networks encode the dependencies among variables. By applying the chain rule to factor the distribution $T(i, \alpha, j)$ into a set of smaller conditional probabilities and simplifying using the conditional independence relationships implicit in the Bayesian network, we obtain the following factored representation.

$$\begin{aligned}
T(i, \alpha, j) &= \Pr(\vec{X}_{t+1} = j | \vec{X}_t = i, \vec{U}_t = \alpha) \\
&= \Pr(X_{1,t+1}, \dots, X_{J,t+1} | X_{1,t}, \dots, X_{J,t}, U_{1,t}, \dots, U_{K,t}) \\
&= \prod_{i=1}^{J} \Pr(X_{i,t+1} | \text{Parents}(X_{i,t+1})) \\
&= \Pr(X_{1,t+1} | X_{1,t}, U_{1,t}) \times \Pr(X_{2,t+1} | X_{1,t}, X_{2,t}, U_{2,t}) \times \\
&\qquad \Pr(X_{3,t+1} | X_{2,t}, X_{3,t}, U_{2,t})
\end{aligned} \tag{9}$$

Further economy of representation can be obtained by using a tree representation for each of the terms (condi-

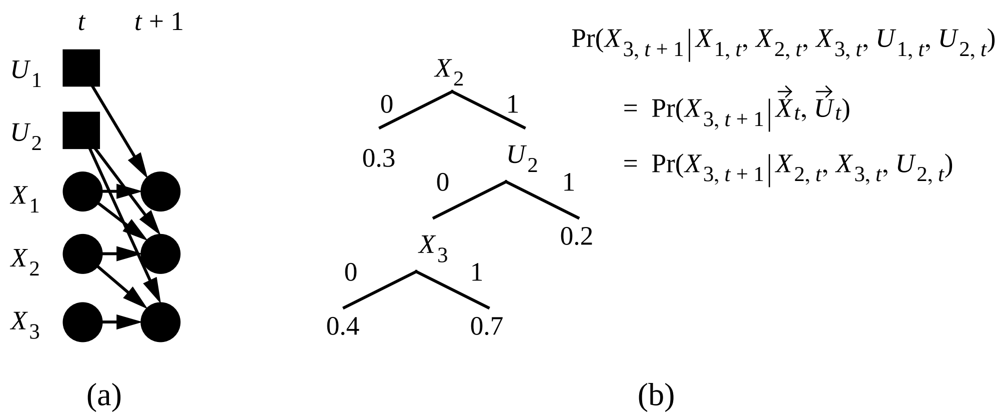


Figure 1: (a) Bayesian network representation for encoding MDP state-transition probabilities, and (b) decision tree representation for a conditional probability distribution

tional probabilities) in the factored form (see Figure 1.b) as opposed to using a simple table.

In the following, we assume that for each fluent X we have a factored representation of the probability that the fluent will be true given the values of fluents in the previous stage, *e.g.*, see Figure 1.a. Note further that this representation induces a partition of the space $Q \times A$ which we denote by P_X, e.g.,

$$P_{X_3} = \{\neg X_2, X_2 \wedge \neg U_2 \wedge \neg X_3, X_2 \wedge \neg U_2 \wedge X_3, X_2 \wedge U_2\} \tag{10}$$

in the case of conditional probability distribution shown in Figure 1.b.

4. Value Iteration

Value iteration is a standard technique for computing the optimal value function for an explicitly represented MDP (*i.e.*, one in which we have explicit sets Q and A for the state and action sets rather than factorial representations of such sets) in run-time polynomial in the sizes of the state and action sets.

The nonlinear operator that is at the heart of value iteration requires computing for each state a value maximizing over all actions and taking expectations over all states

$$Lv(i) = \max_\alpha (R(i, \alpha) + \gamma \sum T(i, \alpha, j) v(j)) \tag{11}$$

In structured methods for solving MDPs with factorial representations, such as those described in [Boutilier *et al.*, 1995] [Dean & Givan, 1997], instead of considering the consequences of an action taking individual states to individual states, we consider how an action takes sets of states to sets of states. Here we extend this idea to consider how sets of actions takes sets of states to sets of states (see Figure 2). The result is a reformulation of the Bellman backup that, instead of quantifying over individual states and actions, quantifies over the blocks of various partitions of the state and action spaces.

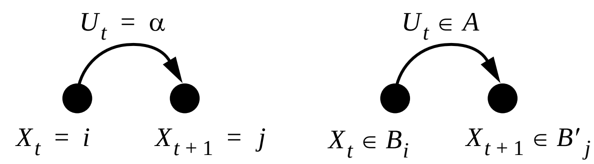


Figure 2: State-to-state transitions versus block-to-block transitions

$$Lv(B_i) = \max_{A_k} (R(B_i, A_k) + \gamma \sum T'(B_i, A_k, B'_j) v(B'_j)) \tag{12}$$

In this case, value functions are defined over the blocks of partitions. The trick is to find partitions such that for any starting block B, destination block B', and block of actions A, for any action $\alpha \in A$ and state $i \in B$, the probability $\Pr(X_{t+1} \in B' | X_t = i, U_t = \alpha)$ is the same (or, in the case of approximations, nearly the same). In the next section, we describe methods which induce partitions that ensure this sort of uniform transition behavior. In the appendix, we describe how to perform Bellman backups which construct the requisite partitions on the fly.

5. Constructing a Minimal Model

In this section we discuss methods to automatically convert an MDP that is represented in factored form as above into a possibly much smaller MDP in explicit form that captures all the information we care about from the original MDP. Specifically, we start with an MDP M whose state space Q and action space A are represented as the set of truth assignments to corresponding sets of propositional fluents X_i and U_i, respectively. We then automatically extract from M an MDP M' whose state and action spaces are simple sets, such that the optimal value function V^* of M' can be easily interpreted as the optimal value function for our original MDP M, and likewise for the optimal policy π^* of M'. M' may have a much smaller state and/or action space than M, and so may allow the practical application of standard MDP solution techniques which could not be effectively applied to the explicit form of M. We call the smallest M' with these properties the *minimal model* of M. This work is an extension of our work on constructing minimal models for MDPs with factored state spaces — we extend that work here to the case of factored action spaces.

In our previous work on model minimization with factored state spaces, the algorithm built up a partition of the state space of the factored MDP M — this partition was represented, for example, by a set of boolean formulas, one for each block of the partition. This partition was chosen in such a way that the blocks of the partition could be viewed as states in an *aggregate state* MDP which was equivalent to M in the sense described above. The algorithm was designed to select the *coarsest* such partition, resulting in the smallest equivalent aggregate state MDP, *i.e.*, the minimal model. In order to extend this work to allow factored action spaces, we now consider partitions of a larger space, the product space of Q and A, denoted $Q \times A$.

Definition 1: Given an MDP $M = (Q, A, T, R)$, either factored or explicit, and a partition P of the space $Q \times A$, we make the following definitions:

a. The *projection* $P|_Q$ *of* P *onto* Q is the partition of Q such that two states p and q are in the same block if and only if for every action a in A, the pairs $\langle p, a\rangle$ and $\langle q, a\rangle$ are in the same block of P. Likewise, the *projection* $P|_A$ *of* P *onto* A is the partition of A such that two actions a_1 and a_2 are in the same block A_1 if and only if for every state p in Q, the pairs $\langle p, a_1\rangle$ and $\langle p, a_2\rangle$ are in the same block of P. In like manner, in factored representations, we can speak of the projection of P onto any set of fluents (meaning the projection onto the set of truth assignments to those fluents).

b. Given a state p, an action a, and a set B of states, the *block transition probability* $T(p, a, B)$ *from* p *to* B *under* a to denote the sum for all states $q \in B$ of the transition probability from state p to state q under action a, *i.e.*, $\Pr(X_{t+1} \in B \mid X_t = p, U_t = a)$.

c. P is *reward homogeneous with respect to* M if and only if every two state-action pairs $\langle p, a\rangle$ and $\langle q, b\rangle$ in the same block B of P have the same immediate reward $R(p, a) = R(q, b)$. We denote this reward $R_P(B)$.

d. P is *dynamically homogeneous with respect to* M if for every two state-action pairs $\langle p, a\rangle$ and $\langle q, b\rangle$ in the same block B in P, the block transition probability to each block in $P|_Q$ is the same for p under action a as it is for q under action b. That is, for each block B' in $P|_Q$, $T(p, a, B') = T(q, b, B')$. We denote this transition probability $T_p(B, B')$.

e. P is *homogeneous with respect to* M if it is both reward and dynamically homogeneous with respect to M.

f. When P is homogeneous with respect to M, the *quotient MDP* M_P is defined to be the MDP $M_P = (P|_Q, P|_A, T_P, R_P)$, where $T_P(B_1, A_1, B_2)$ is defined as $T(p, a, B_2)$ for any state p in B_1 and a in A_1 (homogeneity guarantees that this value is independent of the choice of p and a). M_P is formed from M by simultaneous state space and action space *aggregation*.

g. Given a function $f : (P|_Q \to X)$ for any set X, the function f *lifted to* M, written f_M, is function from Q to X which maps each state q in Q to $f(B)$, where B is the block of P to which q belongs. Intuitively, if we think of f as a function about M_P, then f_M is the corresponding function about M.

The following theorem implies that if we want the optimal value function or an optimal policy for an MDP M it suffices to compute a homogeneous partition P for M, and then compute the optimal value function and/or the optimal policy for M_P. Since it is possible that M_P has a much smaller state space than M, this can lead to a significant savings.

Theorem 1: Given an MDP $M = (Q, A, T, R)$ and a partition P of $Q \times A$ which is homogeneous with respect to M, if V^* and η^* are the optimal value function and optimal policy for M_P, then $(V^*)_M$ and $(\eta^*)_M$ are the corresponding optimal value function and optimal policy for M.

We say that one partition refines another if every block of the second partition is contained in some block of the first partition. It can be shown that every homogeneous partition of a given MDP refines a distinguished partition, the *minimal homogeneous partition.* We show below how to automatically compute this partition. Our algorithm is a generalization of the model minimization algorithm we presented for MDPs with factored state spaces but explicit action spaces. From here on we use the notation M_P to denote the quotient MDP for M relative to the minimal homogeneous partition of M, which we denote P_M.

Before presenting the algorithm, we give a brief discussion of the run-time complexity issues involved in using this algorithm as a means to obtain the optimal policy for an MDP. First, there is no guarantee that M_P has a smaller state space than M — it is easy to construct MDPs for which the minimal homogeneous partition is the trivial partition where every block has a single state in it (for example, any MDP where every state has a different reward value). Second, even in cases where M_P is much smaller than M, the process of finding M_P can be exponentially hard. We have shown in our previous work that when M is represented with a factored state space (but an explicit action space), it is NP hard to find M_P. This result generalizes immediately to the case where M also has a factored action space. The news is not all bad, however — all the above results concern worst-case performance — it is possible for M_P to be exponentially smaller than M and for it to be found in time polynomial in the size of M_P.

In the algorithm below, we assume that M is represented with factored state and action spaces, and we wish to find the minimal homogeneous partition of $Q \times A$. We represent partitions of $Q \times A$ as finite sets of disjoint propositional formulas over the fluents defining Q and A. So, for example, if Q is defined by the fluents $\{X_1, \ldots, X_J\}$, and A is defined by the fluents $\{U_1, \ldots, U_K\}$, then a simple partition of $Q \times A$ could be given by the set of three formulas $\{X_3, \neg X_3 \wedge U_2, \neg X_3 \wedge \neg U_2\}$.

Given a partition P of $Q \times A$ represented as just described, we assume a heuristic procedure for computing the projections $P|_Q$ and $P|_A$ of P onto Q and A, respectively. Computing projections in this representation is an *NP*-hard task in itself[1], hence our reliance on heuristic propositional manipulations here. The usefulness of the algorithm will depend in part on the effectiveness of the heuristics selected for computing projections. A second *NP*-hard propositional operation we assume is *partition intersection* — in this operation two partitions P_1 and P_2 are combined to form one partition $P_1 \cap P_2$ whose blocks are the non-empty pairwise intersections of the blocks from P_1 and P_2. It is the non-emptiness requirement that makes this operation *NP*-hard, as it is just propositional satisfiability in the above representation for blocks.

The algorithm is based on the following property of the homogeneous partitions P of an MDP M: by definition, a partition is homogeneous if and only if every block in the partition is *stable* — a block B in a partition P is *stable* if and only if for every block B' in $P|_Q$, every pair $\langle p, a \rangle$ in B has the same block transition probability $T(p, a, B')$ from p under a to B'.

The basic operation of the algorithm, which we call Backup, takes a partition of $Q \times A$ and returns a refinement of that partition. Iteration of this operation is guaranteed to eventually reach a fixed point (because a partition of a finite space can only be refined finitely many times). The operation will have the property that any fixed point must be stable. It will also have the property that each refinement made must be present in minimal homogeneous partition P_M (*i.e.,* any two state-action pairs split apart from each other by the operation must be in different blocks in P_M), thus guaranteeing that the fixed point found is in fact P_M (assuming that the refinements, or block divisions, present in the partition used to start the iteration are also present in P_M).

We define the $\text{Backup}(P)$ operation for input partition P in terms of the partitions P_Y (for each state space fluent Y) provided in specifying the action dynamics for the MDP M. Let X_P be the set of all state space fluents which are mentioned in any formula in representation of the current partition P. First, construct the intersection I_P of all partitions P_Y such that the fluent Y is in X_P:

$$I_P = P \cap \bigcap_{Y \in X_P} P_Y \tag{13}$$

I_P makes all the distinctions needed within $Q \times A$ so that within each block of I_P the block transition probability to each block of $P|_Q$ is the same (*i.e.,* for any block B of $P|_Q$ and any two pairs $\langle q_1, a_1 \rangle$, $\langle q_2, a_2 \rangle$ in the same block of I_P, the block transition probability from q_1 to B under a_1 is the same as the block transition probability from q_2 to B under a_1). However, I_P may make distinctions that don't need to be made, violating the desired properties mentioned above that ensure we compute the *minimal* partition. For this reason it is necessary to define $\text{Backup}(P)$ as the *clustering* of the partition I_P in which blocks of I_P are combined if they have identical block transition behavior to the blocks of $P|_Q$. Formally, we define an equivalence relation on the blocks of I_P such that two blocks B_1 and B_2 are equivalent if and only if for every block B in $P|_Q$, every pair $\langle p_1, a_1 \rangle$ in B_1, and every pair $\langle p_2, a_2 \rangle$, the block transition probabilities $T(p_1, a_1, B)$ and $T(p_2, a_2, B)$ are the same. We then define $\text{Backup}(P)$ to be partition whose blocks are the unions of the equivalence classes of blocks of I_P under this equivalence relation (in our block representa-

1. Given a propositional formula Φ and new propositional symbols σ, τ_1, τ_2, consider the projection of the partition given by the formulas $\{(\Phi \wedge \sigma) \vee (\tau_1 \wedge \tau_2),\ (\Phi \wedge \neg\sigma) \vee (\tau_1 \wedge \neg\tau_2),\ \neg\Phi \vee \neg\tau_2\}$ onto the symbol σ. This partition has two blocks if Φ is satisfiable and one block otherwise (τ_1 and τ_2 here just ensure block non-emptiness).

tion, union can be easily constructed by taking the disjunction of the block formulas for the blocks to be unioned).

We can now define the model minimization algorithm for MDPs with factored action and state spaces. We start with an initial partition of $Q \times A$ determined by the reward partition P_R in which two pairs $\langle p_1, a_1 \rangle$ and $\langle p_2, a_2 \rangle$ are in the same block if and only if the same reward is given for taking a_1 in p_1 as for taking a_2 in p_2 — this partition is given directly as part of the factored representation of the MDP. We then iterate the Backup operation on this partition, computing $\mathrm{Backup}(P_R)$, then $\mathrm{Backup}(\mathrm{Backup}(P_R))$, and so forth until we produce the same partition twice in a row (since each successive partition computed refines the previous one, it suffices to stop if two partitions with the same number of blocks are produced in successive iterations).

We have the following theorems about the algorithm just described:

Theorem 2: Given an MDP M represented with factored state and action spaces, the partition computed by the model minimization algorithm is the minimal homogeneous partition P_M.

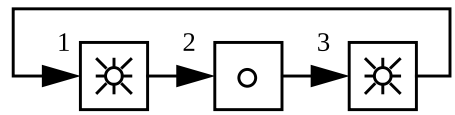
Theorem 3: Given an MDP M with factored state and action spaces, the model minimization performs a polynomial number of partition intersection and projection operations relative to the number of blocks in the computed partition.

6. Examples and Preliminary Experiments

In this section, we present examples that illustrate cases in which our approach works and cases in which our approach fails. We chose simple robotics examples to take advantage of the reader's intuition about spatial locality. Of course, our algorithm has no such intuition and the hope is that the algorithm can take advantage of locality in much more general state and action spaces in which our spatial intuitions provide little or no guidance. We also describe some experiments involving a preliminary implementation of the model minimization algorithm.

Examples

Consider a robot navigating in a simple collection of rooms. Suppose there are n rooms connected in a circular chain and each room has a light which is either on or off. The robot is in exactly one of the n rooms and so there are $n2^n$ states. At each stage, the robot can choose to go forward or stay in the current room and choose to turn on or off each of the n lights and so there are 2^{n+1} actions. There are n independent action parameters, one for each room. Figure 3 shows the basic layout and the relevant state and action variables.



$$\left.\begin{array}{l} R \in \{1, 2, 3\} \\ L_1 \in \{\mathrm{on}, \mathrm{off}\} \\ L_2 \in \{\mathrm{on}, \mathrm{off}\} \\ L_3 \in \{\mathrm{on}, \mathrm{off}\} \end{array}\right\} \text{state variables}$$

$$\left.\begin{array}{l} M \in \{\mathrm{stay}, \mathrm{go}\} \\ A_1 \in \{\mathrm{turn\_on}, \mathrm{turn\_off}\} \\ A_2 \in \{\mathrm{turn\_on}, \mathrm{turn\_off}\} \\ A_3 \in \{\mathrm{turn\_on}, \mathrm{turn\_off}\} \end{array}\right\} \text{action variables}$$

Figure 3: State and action variables for the robot example.

Now consider the following variations on the dynamics and rewards. To keep things simple we consider only deterministic cases.

Case 1: Rewards: The robot gets a reward of 1 for being in room n and otherwise receives no reward. **Dynamics:** If the light is on in the room where the robot is currently located and the robot chooses to go forward then it will end up in the next room with probability 1 at the next stage; if the light is off and it chooses to go forward then it will remain in the current room with probability 1. If the robot chooses to stay, then it will remain in the current room with probability 1. If the robot sets the action parameter A_i to $\mathrm{turn\_on}$ ($\mathrm{turn\_off}$) at stage t, then with probability 1 at stage $t+1$ the light in room i will be on (off). **Analysis:** The number of states in the minimal model is $2n$ and the number of actions $6n$ (see Figure 4).

Case 2. Rewards: Exactly as in case 1. **Dynamics:** Suppose that instead of turning lights on or off the robot is able to choose to toggle or not toggle a switch for each light. If the robot does not toggle the light then the light remains as it was; if the robot does toggle the switch then the light changes status: on to off or off to on. **Analysis:** In this case, the number of states is $n2^n$. Consider transitions from the block $B_1 = (R = 1) \wedge (L_1 = \mathrm{off})$ to the block $B_2 = (R = 2) \wedge (L_2 = \mathrm{off})$ and note that the probability of ending up in B_2 starting from B_1 depends on whether L_2 is on or off. This dependence requires that we split B_1 into two blocks corresponding to those in which L_2 is on and those in which it is off. This splitting will continue to occur implicating the lights in ever more distant rooms until we have accounted for all of the lights. The number of aggregated actions is relatively small however; the same

Number of rooms	2	3	4	5	6	7	8	9
Number of aggregate blocks	14	20	26	32	38	44	50	56
Elapsed time in seconds	0.07	0.15	0.27	0.43	0.93	1.24	2.00	2.75
# States in Unminimized MDP	8	24	64	160	384	896	2024	4608
#Actions in Unminimized MDP	8	16	32	64	128	256	512	1024

Table 1: Experimental Results

$$\text{states} \begin{cases} (R = i) \wedge (L_i = \text{off}) \\ (R = i) \wedge (L_i = \text{on}) \end{cases}$$

$$\text{actions} \begin{cases} (M = \text{go}) \wedge (A_i = \text{turn\_off}) \\ \qquad \wedge (A_{i+1} = \text{turn\_off}) \\ (M = \text{go}) \wedge (A_i = \text{turn\_on}) \\ \qquad \wedge (A_{i+1} = \text{turn\_off}) \\ (M = \text{go}) \wedge (A_i = \text{turn\_off}) \\ \qquad \wedge (A_{i+1} = \text{turn\_on}) \\ (M = \text{go}) \wedge (A_i = \text{turn\_on}) \\ \qquad \wedge (A_{i+1} = \text{turn\_on}) \\ (M = \text{stay}) \wedge (A_i = \text{turn\_on}) \\ (M = \text{stay}) \wedge (A_i = \text{turn\_off}) \end{cases}$$

Figure 4: Aggregate states and actions for the minimal model of Case 1.

aggregated actions as in the Case 1.

Case 3: Rewards: Reward in a state is inversely proportional to the number of lights turned on. This would require an extra variable for the total number of lights turned on in order to represent the dynamics compactly. **Dynamics:** As in case 1 except that the robot is only able to turn on lights that are local, *i.e.,* in the current or next room. **Analysis:** In this case both the state and action spaces blow up in the minimal model.

Experiments

The case analyses in the previous section only tell part of the story. Our model reduction algorithm is polynomial in the size of the minimal model assuming that partition intersection and projection operations blocks can be performed in polynomial time in the size of the domain description. Unfortunately, these operations are computationally complex in the worst case. There are two methods that we can use to deal with this complication: First, we could simply proceed using the best known heuristics for manipulating partitions represented as sets of logical formulas and hope that the worst case doesn't occur in practice. Second, we could choose a restrictive representation for partitions such that the partition operations can be performed in polynomial time in the size of their output. Neither is entirely satisfactory. The first has the problem that the worst case can and does occur in practice. The second has the problem that the less expressive representation for partitions may not allow us to represent the desired partition and that the resulting reduced model may be exponentially larger than the minimal model (see [Dean & Givan, 1997] for details).

To understand the issues a little better we have implemented a version of the model minimization algorithm described in Section 5 . It is not quite this algorithm however as we have adopted the second of the methods for dealing with the issue of the complexity of partition operations raised in the previous paragraph. We assume that all blocks in all partitions are represented as conjunctions of state and action parameters. The resulting implementation is not guaranteed to find the minimal model and hence we refer to it as a *model reduction* algorithm.

At the time of submission, we had just begun to experiment and so we only summarize here one simple, but illustrative set of experiments. We encoded a sequence of Case 1 robot problems with increasing numbers of rooms. Our primary interest was to determine the actual size of the reduced models produced by our algorithm and the running time. The hope is that both factors would scale nicely with the size of the room. In fact this turned out to be the case and results for 2 to 9 rooms are shown in Table 1.

We are in the process of performing experiments on similar robot domains with more complex room topologies and additional (non-spatial) dimensions. We are also considering examples that involve resource allocation and transportation logistics in an effort to understand better where the methods described in the this paper provide leverage.

7. Related Work and Discussion

This work draws on work in automata theory that attempts to account for structure in finite state automata [Hartmanis & Stearns, 1966]. Specifically, the present work builds on the work of Lee and Yannakakis [1992] involving on-line model minimization algorithms, extending their work to handle Markov decision processes with large state [Dean & Givan, 1997] and now action spaces. The present work owes a large debt to the work of Boutilier *et al.* [1995, 1996] which assembled the basic ideas for using a factored representation together with a stochastic analog of goal regression to reason about aggregate behaviors in stochastic systems. We explore the connections between model minimization and goal regression in [Givan & Dean, 1997]. Tsitsiklis and Van Roy [1996] also describe algorithms for solving Markov decision processes with factorial models. Our basic treatment Markov decision processes borrows from Puterman [1994].

The primary contributions of this paper consist of (a) noting that we can factor the dynamics of a planning domain along the lines of action parameters as well as state parameters (fluents) and (b) that we can extend the notions quotient graph, minimal model, and model reduction to handle large action spaces as well as large state spaces. The methods presented in this paper extract some but probably not all of the useful structure in the description of the dynamics. They suffer from the problem that if the minimal model is large (say exponential in the number of state and action parameters), then they will incur a cost at least linear in the size of the minimal model. In some cases, this cost can be ameliorated by computing an ε-approximate model [Dean *et al.*, 1997], but such approximation methods offer no panacea for hard scheduling problems.

Many scheduling problems can be represented compactly in terms of a set action parameters in much the same way as large state spaces can be represented in terms of a set of state variables or fluents, but the techniques in this paper do not by themselves provide a great deal of leverage in solving scheduling problems. It is the nature of hard scheduling problems that there are a large set of actions most of which have some impact on solution quality, the trick is to restrict attention to only those actions that have a significant impact on the bottom line. We see the methods described in this paper as contributing to a compilation or preprocessing technology used to exploit structure so as to reduce complexity thereby reducing reliance on human cleverness in representing planning problems.

8. References

[Boutilier *et al.*, 1995] Boutilier, Craig; Dearden, Richard; and Goldszmidt, Moises, 1995. *Exploiting structure in policy construction*. In Proceedings IJCAI 14. IJCAII. 1104-1111.

[Dean and Givan, 1997] Dean, Thomas and Givan, Robert, 1997. *Model minimization in Markov decision processes.* In Proceedings AAAI-97. AAAI.

[Dean *et al.*, 1997] Dean, Thomas; Givan, Robert; and Leach, Sonia, 1997. *Model reduction techniques for computing approximately optimal solutions for Markov decision processes*. In Thirteenth Conference on Uncertainty in Artificial Intelligence.

[Givan & Dean, 1997] Givan, Robert; Dean, Thomas, 1997. *Model Minimization, regression, and propositional STRIPS planning.* In Proceedings of IJCAI 15, IJCAII.

[Lee and Yannakakis, 1992] Lee, David and Yannakakis, Mihalis, 1992. *On-line minimization of transition systems*. In Proceedings of 24th Annual ACM Symposium on the Theory of Computing.

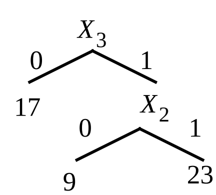
[Puterman, 1994] Puterman, Martin L., 1994. *Markov Decision Processes*. John Wiley & Sons, New York.

[Tsitsiklis and Van Roy, 1996] Tsitsiklis, John N. and Van Roy, Benjamin, 1996, *Feature-based methods for large scale dynamic programming*. Machine Learning 22:59-94.

9. Appendix A: Structured Bellman Backups

As explained in Section 4 , we want to partition the state and action spaces into blocks, so that we can quantify over blocks of states and actions instead of over individual states and actions. Section 5 provides a general method for computing the necessary partitions. In this section, we consider how to implement a structured form of value iteration thereby extending work described in [Boutilier *et al.*, 1995] and [Boutilier & Dearden, 1996]. In terms of value iteration, we want to represent the value function on each iteration as function over blocks of a partition of the state space. Typically the initial value function given to value iteration is the immediate reward function, which we assume has a factored form similar to that shown in Figure 1.b. For example, we might have

$R(\vec{X}_t, \vec{U})$



(14)

Note that the above representation implies a partition of Q consisting of the three blocks represented by the following formulas: $\{\neg X_3, X_3 \wedge \neg X_2, X_3 \wedge X_2\}$. Our problem now reduces to the following: given a value function v defined on blocks of a partition P_v how do we compute the Bellman backup corresponding to Lv with its associated partition P_{Lv} .

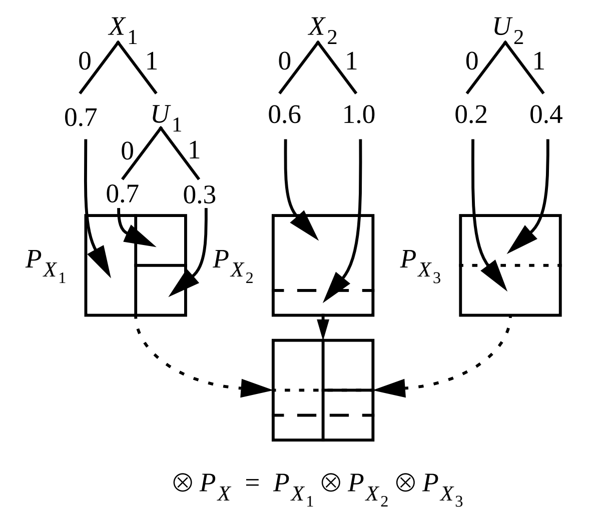


Figure 5: Combining partitions from multiple conditional probability distributions

We work backward from the blocks of P_v each of which is represented as a formula involving the fluents describing the state and action spaces; for each block, we want to determine how we might end up in that block. Suppose the partition P_v consists of formulas involving the fluents X_1, X_2 and X_3, then we have to distinguish between states in the previous stage that have different state transition probabilities with respect to these fluents. To make all of the distinctions necessary to compute the probability that we end up in some block of P_v we need to combine the partitions associated with each of the variables used in describing the blocks of P_v; this combination is the coarsest partition which is a refinement of all the P_X and it is denoted $\otimes P_X$, *e.g.*, see Figure 5.

The partition that we use to represent Lv namely P_{Lv} will be a coarsening of $\otimes P_X$, *i.e.*, a partition in which each block is a union of blocks in $\otimes P_X$. Next we have to calculate the value of each block in $\otimes P_X$.

Note that while P_v is a partition of the state space, $\otimes P_X$ is actually a partition of the product of the state and action spaces; think of each block as represented by a formula consisting of two subformulas: one involving only state variables and the other involving only action fluents. Let $\otimes P_X|_Q$ and $\otimes P_X|_A$ denote the projections of $\otimes P_X$ restricted to Q and A respectively. These two partitions allow us to take expectations and perform maximizations in the Bellman backup without quantifying over all states and actions. There is one further complication that we have to deal with before we can implement structured Bellman backups. We need to be able to calculate

$$T(B, A, B') = \Pr(X_{t+1} \in B' \mid X_t \in B, U_t \in A) \qquad (15)$$

for any $B' \in P_v$, $A \in \otimes P_X|_A$, and $B \in \otimes P_X|_Q$. We could do this if each B' was represented as an assignment to fluents, but this need not be the case; the blocks of P_v could be represented as arbitrary formulas. And so we compute a refinement of P_v which we denote P_v^+ in which each block $B'' \in P_v$ is replaced by the set of blocks corresponding to the set of all assignments to the fluents in the formula representing B''.

Figure 6 illustrates the basic objects involved in performing a structured Bellman backup. The objective is to compute the Bellman backup for all states and actions (Figure 6.a) but without quantifying over all states and actions. We begin with a representation of the n-stages to go value function v specified in terms of the blocks of the partition P_v. When we are done we will have a representation of the $n+1$ -stages to go value function Lv specified in terms of the blocks of the partition P_{Lv} (Figure 6.g).

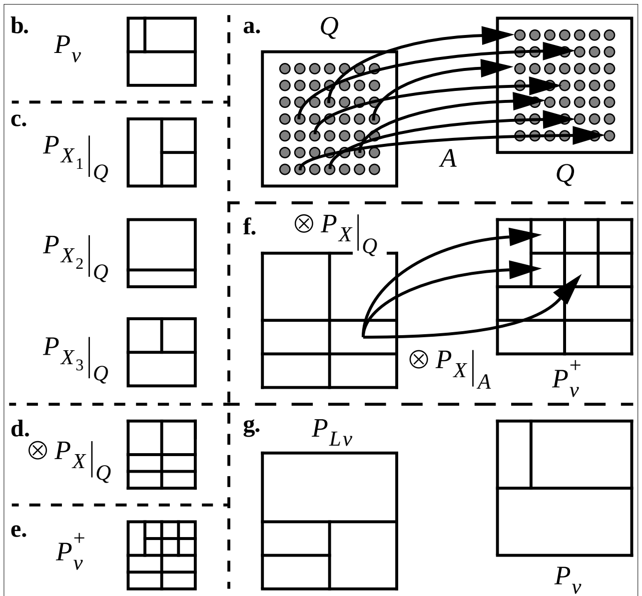


Figure 6: Partitions involved in performing a structured Bellman backup