Equivalence Notions and Model Minimization in Markov Decision Processes

Robert Givan, Thomas Dean, and Matthew Greig

Robert Givan and Matthew Greig
School of Electrical and Computer Engineering
Purdue University, West Lafayette, IN 47907
(765) 494-9068
{givan, mgreig}@ purdue.edu

Thomas Dean
Department of Computer Science
Brown University, Providence, RI 02912
(401) 863-7600
tld@cs.brown.edu

Abstract

Many stochastic planning problems can be represented using Markov Decision Processes (MDPs). A difficulty with using these MDP representations is that the common algorithms for solving them run in time polynomial in the size of the state space, where this size is extremely large for most real-world planning problems of interest. Recent AI research has addressed this problem by representing the MDP in a factored form. Factored MDPs, however, are not amenable to traditional solution methods that call for an explicit enumeration of the state space. One familiar way to solve MDP problems with very large state spaces is to form a reduced (or aggregated) MDP with the same properties as the original MDP by combining "equivalent" states. In this paper, we discuss applying this approach to solving factored MDP problems—we avoid enumerating the state space by describing large blocks of "equivalent" states in factored form, with the block descriptions being inferred directly from the original factored representation. The resulting reduced MDP may have exponentially fewer states than the original factored MDP, and can then be solved using traditional methods. The reduced MDP found depends on the notion of equivalence between states used in the aggregation. The notion of equivalence chosen will be fundamental in designing and analyzing algorithms for reducing MDPs. Optimally, these algorithms will be able to find the smallest possible reduced MDP for any given input MDP and notion of equivalence (i.e. find the "minimal model" for the input MDP). Unfortunately, the classic notion of state equivalence from non-deterministic finite state machines generalized to MDPs does not prove useful. We present here a notion of equivalence that is based upon the notion of bisimulation from the literature on concurrent processes. Our generalization of bisimulation to stochastic processes yields a non-trivial notion of state equivalence that guarantees the optimal policy for the reduced model immediately induces a corresponding optimal policy for the original model. With this notion of state equivalence, we design and analyze an algorithm that minimizes arbitrary factored MDPs and compare this method analytically to previous algorithms for solving factored MDPs. We show that previous approaches implicitly derive equivalence relations that we define here.

1 Introduction

Discrete state planning problems can be described semantically by a state-transition graph (or *model*), where the vertices correspond to the states of the system, and the edges are possible state transitions resulting from actions. These models, while often large, can be efficiently represented, e.g. with factoring, without enumerating the states.

Well-known algorithms have been developed to operate directly on these models, including methods for determining reachability, finding connecting paths, and computing

optimal policies. Some examples are the algorithms for solving Markov decision processes (MDPs) that are polynomial in the size of the state space [Puterman, 1994]. MDPs provide a formal basis for representing planning problems that involve actions with stochastic results [Boutilier *et al.*, 1999]. A planning problem represented as an MDP is given by four objects: (1) a space of possible world states, (2) a space of possible actions that can be performed, (3) a real-valued reward for each action taken in each state, and (4) a transition probability model specifying for each action α and each state p the distribution over resulting states for performing action α in state p.

Typical planning MDPs have state spaces that are astronomically large, exponential in the number of state variables. In planning the assembly of a 1000-part device, potential states could allow any subset of the parts to be "in the closet", giving at least 2^{1000} states. In reaction, AI researchers have for decades resorted to factored state representations—rather than enumerate the states, the state space is specified with a set of finite-domain state variables. The state space is the set of possible assignments to these variables, and, though never enumerated, is well defined. Representing action-transition distributions without enumerating states, using dynamic Bayesian networks [Dean and Kanazawa, 1989], further increases representational efficiency. These networks exploit independence properties to compactly represent probability distributions.

Planning systems using these compact representations must adopt algorithms that reason about the model at the symbolic level, and thus reason about large groups of states that behave identically with respect to the action or properties under consideration, e.g. [McAllester and Rosenblitt, 1991][Draper *et al.*, 1994]. These systems incur a significant computational cost by deriving and re-deriving these groupings repeatedly over the course of planning. Factored MDP representations exploit similarities in state behaviors to achieve a compact representation. Unfortunately, this increase in compactness representing the MDP provably does not always translate into a similar increase in efficiency when computing the solution to that MDP [Littman, 1997]. In particular, states grouped together by the problem representation may behave differently when action sequences are applied, and thus may need to be separated during solution—leading to a need to derive further groupings of states during solution. Traditional operations-research solution methods do not address these issues, applying only to the explicit original MDP model.

Recent AI research has addressed this problem by giving algorithms that in each case amount to state space aggregation algorithms [Baum and Nicholson, 1998][Boutilier and Dearden, 1994][Lin and Dean, 1995] [Boutilier *et al.*, 1995b] [Boutilier and Poole, 1996][Dearden and Boutilier, 1997][Dean and Givan, 1997][Dean *et al.*, 1998]—reasoning directly about the factored representation to find blocks of states that are equivalent to each other. In this work, we reinterpret these approaches in terms of partitioning the state space into blocks of equivalent states, and then building a smaller explicit MDP, where the states in the smaller MDP are the blocks of equivalent states from the partition of the original MDP state space. The smaller MDP can be shown to be equivalent to the original in a well-defined sense, and is amenable to traditional solution techniques. Typically, an algorithm for solving an MDP that takes advantage of an implicit (*i.e.* factored) state-space representation, such as [Boutilier *et al.*, 2000], can be al-

ternatively viewed as transforming the problem to a reduced MDP, and then applying a standard MDP-solving algorithm to the explicit state space of the reduced MDP.

One of our contributions is to describe a useful notion of state equivalence. This notion is a generalization of the notion of *bisimulation* from the literature on the semantics of concurrent processes [Milner, 1989][Hennessy and Milner, 1985]. Generalized to the stochastic case for MDP states, we call this equivalence relation *stochastic bisimilarity.* Stochastic bisimilarity is similar to a previous notion from the probabilistic transition systems literature [Larson and Skou, 1991], with the difference being the incorporation of reward.

We develop an algorithm that performs the symbolic manipulations necessary to group equivalent states under stochastic bisimilarity. Our algorithm is based on the iterative methods for finding a bisimulation in the semantics of concurrent processes literature [Milner, 1989][Hennessy and Milner, 1985]. The result of our algorithm is a model of (possibly) reduced size whose states (called blocks or aggregates) correspond to groups of states in the original model. The aggregates are described symbolically. We prove that the reduced model constitutes a reformulation of the original model: any optimal policy in the reduced MDP generalizes to an optimal policy in the original MDP.

If the operations required for manipulating the aggregates can each be done in constant time then our algorithm runs in time polynomial in the number of states in the reduced model. However, the aggregate manipulation problems, with general propositional logic as the representation are NP-hard, and so, generally speaking, aggregate manipulation operations do not run in constant time. One way to attempt to make the manipulation operations fast is to limit the expressiveness of the representation for the aggregates—when a partition is called for that cannot be represented, we use some refinement of that partition by splitting aggregates as needed to stay within the representation. Using such representations, the manipulation operations are generally more tractable, however the reduced MDP state space may grow in size due to the extra aggregate splitting required. Previous algorithms for manipulating factored models implicitly compute reduced models under restricted representations. This issue leads to an interesting tradeoff between the strength of the representation used to define the aggregates (affecting the size of the reduced MDP), and the cost of manipulation operations. Weak representations lead to poor model reduction, but expressive representations lead to expensive operations (as shown, e.g., in [Dean *et al*., 1997][Goldsmith and Sloan, 2000]).

The basic idea of computing equivalent reduced processes has its origins in automata theory [Hartmanis and Stearns, 1966] and stochastic processes [Kemeny and Snell, 1960], and has been applied more recently in model checking in computer-aided verification [Burch *et al.,* 1994][Lee and Yannakakis, 1992]. Our model minimization algorithm can be viewed as building on the work of [Lee and Yannakakis, 1992] by generalizing non-deterministic transitions to stochastic transitions and introducing a notion of utility.

We claim a number of contributions for this paper. First, we develop a notion of equivalence between MDP states that relates the literatures on automata theory, concurrent process semantics, and decision theory. Specifically, we develop a useful variant of the notion of bisimulation, from concurrent processes, for MDPs. Second, we show that

the mechanisms for computing bisimulations from the concurrent processes literature generalize naturally to MDPs and can be carried out on factored representations, without enumerating the state space. Third, we show that state aggregation (in factored form), using automatically detected stochastic bisimilarity, results in a (possibly) reduced model, and we prove that solutions to this reduced model (which can be found with traditional methods) apply when lifted to the original model. Finally, we carefully compare previous algorithms for solving factored MDPs to the approach of computing a minimal model under some notion of state equivalence (stochastic bisimilarity or a refinement thereof) and then applying a traditional MDP-solving technique to the minimal model.

Section 2 discusses the relevant background material. Section 3 presents some candidate notions of equivalence between states in an MDP, including stochastic bisimulation, and Section 4 builds an algorithm for computing the minimal model for an MDP under stochastic bisimulation. Section 5 compares existing algorithms for working with a factored MDP to our approach. Section 6 covers extensions to this work to handle large action spaces and to select reduced models approximately. Section 7 shows brief empirical results, and the remaining section draws some conclusions. The proofs of our results appear in the appendix, except where noted in the main text.

2 Background Material

2.1 Sequential Decision Problems

2.1.1 Finite Sequential Machines

A *non-deterministic finite sequential machine* (FSM) F (adapted from [Hartmanis and Stearns, 1966]) is a tuple $\langle Q, A, O, T, R\rangle$ where Q is a finite set of states, A is a finite set of inputs (actions), and O is a set of possible outputs. The transition function, T, is a subset of $Q \times A \times Q$ that identifies the allowable transitions for each input in each state. The output function, R, is a mapping from Q to O giving for each state the output generated when transitioning into that state. We say that a state sequence $q_0, \ldots, q_k$ is possible under inputs $\alpha_1, \ldots, \alpha_k$ from A when T contains all tuples of the form $\langle q_{x-1}, \alpha_x, q_x\rangle$. We say that $q_0, \ldots, q_k$ can generate output sequence $o_1, \ldots, o_k$ when R maps each q_x for $x > 0$ to o_x. We can then say that $o_1, \ldots, o_k$ is a possible output sequence when following input sequence $\alpha_1, \ldots, \alpha_k$ from start state q_0 if $o_1, \ldots, o_k$ can be generated from some state sequence $q_0, \ldots, q_k$ possible under $\alpha_1, \ldots, \alpha_k$. Finally, we denote an input sequence as ξ, an output sequence as ϕ, and use $\rightarrow_{F,i}$ to denote generation so that $\xi \rightarrow_{F,i} \phi$ means that output sequence ϕ is possible in FSM F starting at state i under input sequence ξ.

2.1.2 Markov Decision Processes

A *Markov decision process* (MDP) M is a quadruple $\langle Q, A, T, R\rangle$ in which Q is a finite state space, A is a finite action space, T is a mapping from $Q \times A \times Q$ to [0,1], and R is a reward function assigning a non-negative real-numbered utility to each state in Q.[1]

[1] More general reward function forms are often used. For example, one could have R be a mapping from Q x A x Q to real values, in which case it is the transition that carries the reward, not being in a given state. Our method generalizes to these more general reward functions However we adopt state based reward to

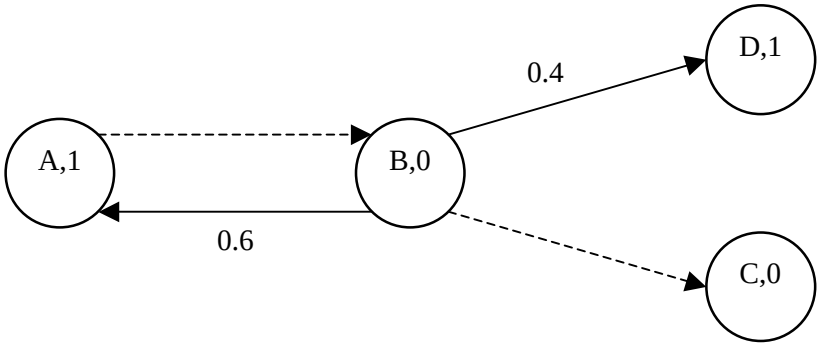

Figure 1. A graph representation of a Markov decision process in which Q = {A, B, C, D}, A = {a, b} (action a is shown with a solid line, b with a dashed line), R(A) = R(D) =1, R(B) = R(C) = 0, and the transition probabilities (T) are given on the associated transitions. The probability of a transition is omitted when that probability is one or zero and deterministic self-loop edges are also omitted, to improve readability.

Transitions are defined by T so that $\forall i, j \in Q$, and $\forall \alpha \in A$, $T(i, \alpha, j)$ equals $\Pr(X_{t+1} = j \mid X_t = i, U_t = \alpha)$, where the random variables X_t and U_t denote the state of the system and the action taken at time t, respectively. Figure 1 shows an MDP represented as a directed graph. The nodes are labeled with the states they represent along with the reward assigned to that state. The edges represent possible transitions labeled with the action and probability of that transition given the action and originating state. In this paper, we refer to this graph representation and to an MDP in general as a *model* for the underlying dynamics of a planning problem [Boutilier *et al*., 1999].

An MDP is essentially an FSM for which the output set O is the real numbers $\mathbb{R}$, and transition probabilities have been assigned. However, in FSMs, inputs are traditionally sequences of input symbols (actions) to be verified, whereas in MDPs "inputs" are usually specified by giving a *policy* to execute. A policy π for an MDP is a mapping from the state space to the action space, $\pi: Q \rightarrow A$, giving the action to select for each possible state. The set of all possible policies is denoted Π. To compare policies, we will employ *value functions* $v: Q \rightarrow \mathbb{R}$ mapping states to real values. The set of value functions, V, is partially ordered by domination, $v_1 \leq_{\text{dom}} v_2$, which holds when $v_1(i) \leq v_2(i)$ at every state i.

2.1.3 Solving Markov Decision Problems

A Markov Decision Problem (also abbreviated MDP by abuse of notation) is a Markov decision process, along with an *objective function* that assigns a value function to each policy. In this paper, we restrict ourselves to one particular objective function: expected, cumulative, discounted reward, with discount rate γ where $0 < \gamma < 1$ [Bellman, 1957][Howard, 1960][Puterman, 1994].[2] This objective function assigns to each policy

simplify the presentation.

[2] Other objective functions such as finite-horizon total reward or average reward can also be used and our approach can easily be generalized to those objective functions.

the value function measuring the expected total reward received from each state, where rewards are discounted by a factor of γ at each time step. The value function v_π assigned by this objective function to policy π is the unique solution to the set of equations

$$v_\pi(i) = R(i) + \gamma \Sigma_j T(i, \pi(i), j)\, v_\pi(j).$$

An *optimal policy* π^* dominates all other policies in value at all states, and it is a theorem that an optimal policy exists. Given a Markov Decision Problem, our goal is typically to find an optimal policy π^*, or its value function v_{π^*}. All optimal policies share the same value function, called the *optimal value function* and written v^*.

An optimal policy can be obtained from v^* by a greedy one step look-ahead at each state—the optimal action for a given state is the action that maximizes the weighted sum of the optimal value at the next states, where the weights are the transition probabilities. The function v^* can be found by solving a system of Bellman equations

$$v(i) = R(i) + \max_\alpha \gamma \Sigma_j T(i, \alpha, j)\, v(j).$$

Value iteration is a technique for computing v^* in time polynomial in the sizes of the state and action sets (but exponential in $1/\gamma$) [Puterman, 1994][Littman *et al.*, 1995], and works by iterating the operator L on value functions, defined by

$$Lv(i) = R(i) + \max_{\alpha \in A} \gamma \Sigma_j T(i, \alpha, j)\, v(j).$$

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

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

and has fixed point v^*. The operator L is called *Bellman backup*. Repeated Bellman backups starting from any initial value function converge to the optimal value function.

2.2 Partitions in State Space Aggregation

A *partition* P of a set $S = \{s_0, s_1, \ldots, s_n\}$ is a set of sets $\{B_1, B_2, \ldots, B_m\}$ such that each B_i is a subset of S, the B_i are disjoint from one another, and the union of all the B_i equals S. We call each member of a partition a *block*. A *labeled partition* is a partition along with a mapping that assigns to each member B_i a label b_i. Partitions define equivalence relations—elements share a block of the partition if and only if they share an equivalence class under the relation. We now extend some of the key notions associated with FSM and MDP states to blocks of states. Given an MDP $M = \langle Q, A, T, R \rangle$, a state $i \in Q$, a set of states $B \subset Q$, and an action $\alpha \in A$, the *block transition probability* from i to B under α, written $T(i, \alpha, B)$, by abuse of notation, is given by: $T(i, \alpha, B) = \Sigma_{j \in B} T(i, \alpha, j)$. We say that a set of states $B \subset Q$ has a well-defined reward if there is some real number r such that for every $j \in B$, $R(j) = r$. In this case we write $R(B)$ for the value r.

Analogously, consider FSM $F = \langle Q, A, O, T, R \rangle$, state $i \in Q$, set of states $B \subset Q$, and action $\alpha \in A$. We say the *block transition from i to B is allowed under* α when $T(i, \alpha, j)$ is true for some state j in B, denoted with the proposition $T(i, \alpha, B)$, and computed by $\vee_{j \in B} T(i, \alpha, j)$. We say a set of states has a well-defined output $o \in O$ if for every $j \in B$, $R(j) = o$. Let $R(B)$ be both the value o and the proposition that the output for B is defined.

Given an MDP $M = \langle Q, A, T, R \rangle$ (or FSM $F = \langle Q, A, O, T, R \rangle$), and a partition P of the state space Q, a quotient model M/P (or F/P for FSMs) is any model of the form $\langle P, A, T', R' \rangle$ (or $\langle P, A, O, T', R' \rangle$ for FSMs) where for any blocks B and C of P, and action α, $T'(B, \alpha, C) = T(i, \alpha, C)$ and $R'(B) = R(i)$ for some i in B. For state $i \in Q$, we denote the block of P to which i belongs as i/P. In this paper, we give conditions on P that guarantee the quotient model is unique and equivalent to the original model, and give methods for finding such P. We also write M/E (likewise, F/E for FSMs), where E is an equivalence relation, to denote the quotient model relative to the partition induced by E (i.e. the set of equivalence classes under E), and i/E for the block of state i under E.

A partition P' is a *refinement* of a partition P, written $P' \underline{\ll} P$ if and only if each block of P' is a subset of some block of P. If, in addition, some block of P' is a proper subset of some block of P, we say that P' is *finer* than P, written $P' \ll P$. The inverse of refinement is *coarsening* ($\underline{\gg}$) and the inverse of finer is *coarser* ($\gg$). The term *splitting* refers to dividing a block B of a partition P into two or more sub-blocks that replace the block B in partition P to form a finer partition P'. We will sometimes treat an equivalence relation E as a partition (the one induced by E) and refer to the "blocks" of E.

2.3 Factored Representations

2.3.1 Factored Sets and Partitions

A set S is represented in *factored form* if the set is specified by giving a set F of true/false[3] variables, along with a Boolean formula over those variables, such that S is the set of possible assignments to the variables that are consistent with the given formula.[4] When the formula is not specified, it is implicitly "true" (true under any variable assignment). When S is given in factored form, we say that S is *factored*. A *factored partition* P of a factored set S is a partition of S whose members are each factored using the same set of variables as are used in factoring S.[5] Except where noted, partitions are represented by default as a set of mutually inconsistent DNF Boolean formulas, where each block is the set of truth assignments satisfying the corresponding formula.

Because we use factored sets to represent state spaces in this paper, we call the variables used in factoring *state variables* or, alternately, *fluents*. One simple type of partition is particularly useful here. This type of partition distinguishes two assignments if and only if they differ on a variable in a selected subset F' of the variables in F. We call such a partition a *fluentwise* partition, denoted Fluentwise(F'). A fluentwise partition can be represented by the set F' of fluents, which is exponentially smaller than any list of the partition blocks. E.g, if $F = \{X_1, X_2, X_3\}$ and $F' = \{X_1, X_2\}$ then the partition Fluentwise(F') has four blocks described by the formulas: $X_1 \wedge X_2$, $X_1 \wedge \neg X_2$, $\neg X_1 \wedge X_2$, and $\neg X_1 \wedge \neg X_2$.

---

[3] For simplicity of presentation we will consider every variable to be Boolean although our approach can easily be generalized to handle any finite-domain variable.

[4] It follows that every factored set is a set of variable assignments. Any set may be trivially viewed this way by considering a single variable ranging over that set (if non-Boolean variables are allowed). Interesting factorings are generally exponentially smaller than enumerations of the set.

[5] Various restrictions on the form of the formulas lead to various representations (e.g. decision trees).

2.3.2 Factored Mappings and Probability Distributions

A mapping from a set X to a set Y can be specified in factored form by giving a labeled partition of X, where the labels are elements of Y. A conditional probability distribution $\Pr(A|B)$ is a mapping from the domain of B to probability distributions over the domain of A, and so can be specified by giving a labeled partition—this is a *factored conditional probability distribution*. A joint probability distribution over a set of discrete variables can be represented compactly by exploiting conditional independencies as a Bayesian belief network [Pearl, 1988]. Here, equivalent compactness is achieved as follows. First, the joint distribution can be written as a product of conditional distributions using the chain rule (for any total ordering of the variables). Next, each of the conditional distributions involved can be simplified by omitting any conditioning variables that are irrelevant due to conditionally independence. Finally, the simplified distributions are written in factored form. A joint distribution so written is called a *factored joint probability distribution*. We show an example of such a factored joint distribution in Figure 2.

2.3.3 Factored Markov Decision Processes

Factored MDPs can be represented using a variety of approaches, including Probabilistic STRIPS Operators (PSOs) [Hanks, 1990][Hanks and McDermott, 1994] [Kushmerick *et al*., 1995] and 2-stage Temporal Bayesian Networks (2TBNs) [Dean and Kanazawa, 1989]. For details of these approaches, we refer to [Boutilier *et al*., 1999]. Here, we will use a representation, similar in spirit, but focusing on the state-space partitions involved. An MDP $M = \langle Q, A, T, R\rangle$ can be given in factored form by giving a quadruple $\langle F, A, T_F, R_F\rangle$,[6] where the state space Q is given in factored form by the set of state variables F (with no constraining formula). The state-transition distribution of a factored MDP is specified by giving, for each fluent f and action α, a factored conditional probability distribution $T_F(\alpha, f)$ representing the probability that f is true after taking α, given the state in which the action is taken—$T_F(\alpha, f)$ is[7] a partition of the state space, where two states are in the same block if and only if they result in the same probability of setting f to true when α is applied, and the block is labeled with that probability. The unfactored transition probabilities $T(i, \alpha, j)$ can be extracted from this representation as

$$T(i,\alpha, j) = \prod_{\{f|j(f)\}} label_F(i,\alpha, f) \prod_{\{f|\neg j(f)\}} (1 - label_F(i,\alpha, f))$$

where $j(f)$ is true if and only if the fluent f is assigned true by state j, and $label_F(i, \alpha, f)$ gives the label assigned to the block containing state i by $T_F(\alpha, f)$. We note that to use this factored representation, we must have that the post-transition fluent values are independent of each other given the pre-transition state[8], so that the probability of arriving at a given state is the product of the probabilities associated with each fluent value for that

[6] We discuss factored action spaces further in Section 6.1, and synchronic effects in Section 6.3.

[7] By our definition of "factored conditional probability distribution"

[8] Factored representations can also be designed that allow dependence between post-transition fluents (so-called "synchronic effects"). For simplicity of presentation here we disallow such dependence, but we discuss the ramifications of allowing dependence later, in Section 6.3.

Pr(A,B,C,D,E)
= Pr(A) Pr(B | A) Pr(C | B,A)
Pr(D | C,B,A) Pr(E | A,B,C,D,E)
= Pr(A) Pr(B | A) Pr(C | A)
Pr(D | C,B) Pr(E | C)

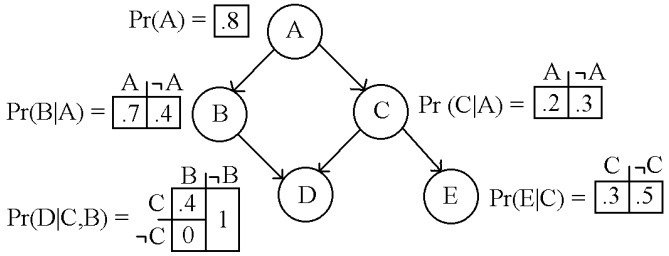

Figure 2. A Bayesian network and decomposition of the joint probability distribution over the variables in the network based on the chain rule and the conditional independencies implied by the network.

state. The reward function R_F of a factored MDP is a factored mapping from states to real numbers—i.e., a labeled factored partition of the state space where each label gives the reward associated with any state in that block. Two states are in the same block of R_F if and only if they yield the same immediate reward.

3 Equivalence Notions for State Space Aggregation

In this section, we discuss state equivalence notions that aim to capture when two states behave identically for all purposes of interest. We first consider some simple definitions and their shortcomings, before defining an appropriate notion. The definitions here are independent of the MDP representation and are inspired by work in concurrent processes that uses unfactored state spaces; our principle contribution is to connect this work to factored state spaces, providing natural algorithms for solving factored MDPs.

3.1 Simple Equivalence Notions for Markov Decision Processes

In this section, we define two simple notions of equivalence between states in an MDP. We argue here that these notions both equate states that we must treat differently, and so are too coarse. The first of these notions is a stochastic generalization of *action-sequence equivalence*, a classic equivalence notion for finite sequential machines [Hartmanis and Stearns, 1966]. Let $F = \langle Q, A, O, T, R\rangle$ and $F' = \langle Q', A, O, T', R'\rangle$ be two FSMs over the same input and output sets. The states i of F and j of F' are action-sequence equivalent if and only if for every input sequence ξ, the same set of output sequences ϕ can be generated under ξ from either state i or state j, *i.e.*,

$$\forall \xi \{ \phi \mid \xi \rightarrow_{F,i} \phi \} = \{ \phi \mid \xi \rightarrow_{F',j} \phi \}.$$

This equivalence notion also naturally applies to two states from the same FSM.

We now generalize this notion, for the stochastic case, to an equivalence notion between states in MDPs. The *distribution over reward sequences* associated with a given MDP assigns to each sequence of actions $\alpha_1, \ldots, \alpha_k$ and starting state q a probability distribution over length k sequences of real values $r_1, \ldots, r_k$. This distribution gives the probability of obtaining the sequence of rewards $r_1, \ldots, r_k$ when starting from state q and performing action sequence $\alpha_1, \ldots, \alpha_k$. Let $M = \langle Q, A, T, R\rangle$ and MDP $M' = \langle Q', A, T',$

$R'\rangle$ be two MDPs with the same action space. The states i of M and j of M' are *action-sequence equivalent* if and only if for every sequence of possible actions $\alpha_1, \ldots, \alpha_n$, for any n, the distributions over reward sequences for i in M and j in M' are the same. Note that this definition applies naturally to two states within the same MDP as well.

FSMs are generally used to map input sequences to output sequences. However, because MDPs are typically used to represent problems in which we seek an effective policy (rather than action sequence), action-sequence equivalence is not an adequate equivalence notion for MDP state aggregation for the purpose of constructing equivalent reduced problems. This is because a policy is able to respond to stochastic events during execution, while a sequence of actions cannot. In particular, two MDP states may be action-sequence equivalent and yet have different values under some policies and even different optimal values. We show an example of such an MDP in Figure 3 where the states i and i' have the same distribution over reward sequences for every action sequence, but i has a better optimal value than i'. This difference in optimal value occurs because policies are able to respond to different states with different actions and thus respond to stochastic transitions based on the state that results. However, action sequences must choose the same sequence of actions no matter which stochastic transitions occur. In the figure, a policy can specify that action α_1 is best in state j_1, while action α_2 is best in state j_2—the policy thus gains an advantage when starting from state i that is not available when starting from state i'. Action sequences, however, must commit to the entire sequence of actions that will be performed at once and thus find states i and i' equally attractive.

The failure of action-sequence equivalence to separate states with different optimal values suggests a second method for determining state equivalence: directly comparing the optimal values of states. We call this notion *optimal value equivalence*. MDP states i and j are optimal value equivalent if and only if they have the same optimal value.

Optimal value equivalence also has substantial shortcomings. States equivalent to each other under optimal value equivalence may have entirely different dynamics with respect to action choices. In general, an optimal policy differentiates such states. In some sense, the fact that the states share the same optimal value may be a "coincidence". As a result, we have no means to calculate equivalence under this notion, short of computing and comparing the optimal values of the states—but since an optimal policy can be found by greedy one-step look-ahead from the optimal values, computing this equivalence relation will be as hard as solving the original MDP. Furthermore, we are interested in aggregating equivalent states in order to generate a reduced MDP. While the equivalence classes under optimal value equivalence can serve as the state space for a reduced model, it is unclear what the effects of an action from such an aggregate state should be—the effects of a single action on different equivalent states might be entirely different. Even if we manage to find a way to adequately define the effects of the actions in this case, it is not clear how to generalize a policy on a reduced model to the original MDP.

Neither of these equivalence relations suffices. However, the desired equivalence relation will be a refinement of both of these: if two states are equivalent, they will be both action sequence equivalent and optimal value equivalent. To see why, consider the proposed use for the equivalence notion, namely to aggregate states defining a smaller

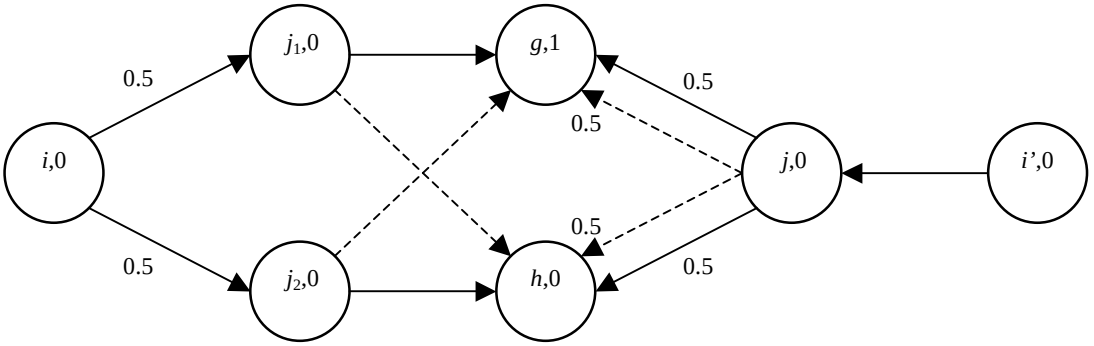

Figure 3. An MDP where action sequence equivalence would find i and i' to be equivalent even though they have different optimal values. Any edge not labeled is deterministic and deterministic self-loops are omitted. Transitions involving action α_1 are shown with a solid edge while those involving α_2 is shown with a dotted edge.

equivalent MDP that we can then solve in order to generalize that solution to the larger original MDP. For the reduced MDP to be well defined, the reward value for all equivalent states must be equal; likewise, the transition distributions for all equivalent states and any action must be equal (at the aggregate level). Thus, the desired equivalence relation should only equate states that are both action sequence and optimal value equivalent (the former is proved by induction on sequence length and the latter by induction on horizon).

3.2 Bisimulation for Non-deterministic Finite Sequential Machines

Bisimulation for FSM states captures more state properties than is possible using action sequence equivalence. Bisimulation for concurrent processes [Park, 1981] generalizes a similar concept for deterministic FSM states from [Hartmanis and Stearns, 1966].

Let $F = \langle Q, A, O, T, R\rangle$ and $F' = \langle Q', A, O, T', R'\rangle$ be two FSMs over the same input and output spaces. A relation $E \subseteq Q \times Q'$ is a *bisimulation* if each $i \in Q$ (and $j \in Q'$) is in some pair in E, and whenever $E(i, j)$ then the following hold for all actions α in A,

1. $R(i) = R'(j)$,
2. for i' in Q s.t. $T(i, \alpha, i')$, there is a j' in Q' s.t. $E(i', j')$ and $T'(j, \alpha, j')$, and conversely,
3. for j' in Q' s.t. $T'(j, \alpha, j')$, there is an i' in Q s.t. $E(i', j')$ and $T(i, \alpha, i')$.

We say two FSM states i and j are *bisimilar* if there is some bisimulation B between their FSMs in which $B(i, j)$ holds. Bisimilarity is an equivalence relation, itself a bisimulation.

The reflexive symmetric transitive closure of any bisimulation between two FSMs, restricted to the state space of either FSM gives an equivalence relation which partitions the state space of that FSM. The bisimulation can be thought of as a one-to-one mapping between the blocks of these two partitions (one for each FSM) where the two blocks are related if and only if some of their members are related. All block members are bisimilar to each other and to all the states in the block related to that block by the bisimulation. Next, an immediate consequence of the theory of bisimulation [Park, 1981].

Theorem 1: FSM states related by a bisimulation are action-sequence equivalent.[9]

We note that optimal-value equivalence is not defined for FSMs.

Aggregation algorithms construct a partition of the state space Q and aggregate the states in each partition block into a single state (creating one aggregate state per partition block) in order to create a smaller FSM with similar properties. When the partition used is due to a bisimulation, the resulting aggregate states are action-sequence equivalent to the corresponding states of the original FSM. The following theorem is a nondeterministic generalization of a similar theorem given in [Hartmanis and Stearns, 1966].

Theorem 2: Given an FSM $F = \langle Q, A, O, T, R \rangle$ and an equivalence relation $E \subseteq Q \times Q$ that is a bisimulation, there is a unique quotient machine F/E and each state i in Q is bisimilar to the state i/E in F/E.[10]

[Hennessy and Milner, 1985] show that bisimulation captures exactly those properties of FSM states which can be described in Hennessy-Milner Modal Logic (HML).[11] We briefly define this logic here as an aside—we do not build on this aspect of bisimulation here. The theorem below states that HML can express exactly those properties that can be used for state aggregation in the factored FSM methods we study. Following [Larson and Skou, 1991],[12] the formulas ψ of HML are given by the syntax:

$$\psi ::= \text{True} \mid \text{False} \mid [\alpha, o]\psi \mid \langle\alpha, o\rangle\psi \mid (\psi_1 \vee \psi_2) \mid (\psi_1 \wedge \psi_2)$$

The satisfaction relation $i \models \psi$ between a state i in an FSM F and a HML formula ψ is defined as usual for modal logics and Kripke models. Thus, $i \models \langle\alpha,o\rangle\psi$ whenever $j \models \psi$ for some j where $T(i, \alpha, j)$ and $R(j) = o$, and dually, $i \models [\alpha,o]\psi$ whenever $T(i, \alpha, j)$ and $R(j) = o$ implies $j \models \psi$.

Theorem 3: [Hennessy and Milner, 1985] Two states i and j of an FSM F are bisimilar just in case they satisfy exactly the same HML formulas.[13]

3.3 Stochastic Bisimulation for Markov Decision Processes

In this section, we define stochastic bisimilarity for MDPs as a generalization of bisimilarity for FSMs, generalizing "output" to "reward" and adding probabilities. Stochastic bisimilarity differs from bisimilarity in that transition behavior similarity must be

[9] For space reasons, we do not repeat the proof of this result.

[10] For space reasons, we do not repeat the proof of this result.

[11] We note that the semantics of concurrent processes work deals with domains that are generally infinite and possibly uncountable. Our presentation for FSMs is thus a specialization of that work to finite state spaces.

[12] HML and the corresponding bisimulation notion are normally defined for sequential machines with no outputs, where the only issue is whether an action sequence is allowed or not. We make the simple generalization to having outputs in order to ease the construction of the MDP analogy and to make the relationship between the literatures more apparent.

[13] For space reasons, we do not repeat the proof of this result.

measured at the equivalence class (or "block") level—bisimilar states must have the same block transition probabilities to each block of "similar" states.

The i/E notation generalizes to any relation $E \subseteq Q \times Q'$. Define i/E be the equivalence class of i under the reflexive, symmetric, transitive closure of E, restricted to Q, when $i \in Q$ (restrict to Q' when $i \in Q'$). The definitions are identical when E is an equivalence relation in $Q \times Q$.

Let $M = \langle Q, A, T, R \rangle$ and $M' = \langle Q', A, T', R' \rangle$ be two MDPs with the same action space, and let $E \subseteq Q \times Q'$ be a relation. We say that E is a *stochastic bisimulation*[14] if each $i \in Q$ (and $j \in Q'$) appears in some pair in E, and, whenever $E(i, j)$, both of the following hold for all actions α in A,

1. $R(i/E)$ and $R'(j/E)$ are well defined and equal to each other.
2. For states i' in Q, and j' in Q' s.t. $E(i', j')$, $T(i, \alpha, i'/E) = T'(j, \alpha, j'/E)$.

See Section 2.2 for the definition of $T(i, \alpha, B)$ for a block B. We say that two MDP states i and j are *stochastically bisimilar* if there is some stochastic bisimulation between their MDPs which relates i and j. Note that these definitions can be applied naturally when the two MDPs are the same. This definition is closely related to the definition of *probabilistic bisimulation* for probabilistic transition systems (MDPs with no utility or reward specified) given in [Larson and Skou, 1991].

Theorem 4: Stochastic bisimilarity restricted to the states of a single MDP is an equivalence relation, and is itself a stochastic bisimulation from that MDP to itself. [15]

A stochastic bisimulation can be viewed as a bijection between corresponding blocks of partitions of the corresponding state spaces. So two MDPs will have a bisimulation between them exactly when there exist partitions of the two state spaces whose blocks can be put into a one-to-one correspondence preserving block transition probabilities and rewards. Stochastic bisimulations that are equivalence relations have several desirable properties as equivalence relations on MDP states.[16]

Theorem 5: Any stochastic bisimulation that is an equivalence relation is a refinement of both optimal value equivalence and action sequence equivalence.

We are interested in state space aggregation and thus primarily in equivalence relations. The following theorem ensures that we can construct an equivalence relation from any bisimulation that is not already an equivalence relation.

---

[14] Stochastic bisimulation is also closely related to the *substitution property* of finite automata developed in [Hartmanis and Stearns, 1966] and the notion of *lumpability* for Markov chains [Kemeny and Snell, 1960].

[15] We note that the proofs of all the theorems presented in this paper, except where omitted and explicitly noted, are left until the appendix for sake of readability.

[16] It is possible to give a stochastic modal logic for those properties of MDP states that are discriminated by stochastic bisimilarity. For an example of a closely related logic that achieves this goal for probabilistic transition systems, see the probabilistic modal logic given in [Larson and Skou, 1991].

Theorem 6: The reflexive, symmetric, transitive closure of any stochastic bisimulation from MDP $M = \langle Q, A, T, R \rangle$ to any MDP, restricted to $Q \times Q$, is an equivalence relation $E \subseteq Q \times Q$ that is a stochastic bisimulation from M to M.

Any stochastic bisimulation used for aggregation preserves the optimal value and action sequence properties as well as the optimal policies of the model:

Theorem 7: Given an MDP $M = \langle Q, A, T, R \rangle$ and an equivalence relation $E \subseteq Q \times Q$ that is a stochastic bisimulation, each state i in Q is stochastically bisimilar to the state i/E in M/E. Moreover, any optimal policy of M/E induces an optimal policy in the original MDP.

It is possible to give a stochastic modal logic, similar to the Hennessy-Milner modal logic above, that captures those properties of MDP states that are discriminated by stochastic bisimilarity (e.g., see [Larson and Skou, 1991] which omits rewards).

4 Model Minimization

Any stochastic bisimulation can be used to perform model reduction by aggregating states that are equivalent under that bisimulation. The definitions ensure that there are natural meanings for the actions on the aggregate states. The coarsest bisimulation (stochastic bisimilarity) gives the smallest model, which we call the "minimal model" of the original MDP. In this section, we investigate how to find bisimulations, and bisimilarity efficiently. We first summarize previous work on computing bisimilarity in FSM models, and then generalize this work to our domain of MDPs.

4.1 Minimizing Finite State Machines with Bisimilarity

Concurrent process theory provides methods for computing the bisimilarity relation on an FSM state space. We summarize one method, and show how to use it to compute a minimal FSM equivalent to the original [Milner, 1990]. Consider FSMs $F = \langle Q, A, O, T, R \rangle$ and $F' = \langle Q', A, O', T', R' \rangle$ and binary relation $E \subseteq Q \times Q'$. Define $H(E)$ to be the set of all pairs (i, j) from $Q \times Q'$ satisfying the following two properties. First, $E(i, j)$ must hold. Second, for every action $\alpha \in A$, each of the following conditions holds:

1. $R(i) = R'(j)$,
2. for i' in Q s.t. $T(i, \alpha, i')$, there is a j' in Q' s.t. $E(i', j')$ and $T'(j, \alpha, j')$, and conversely,
3. for j' in Q' s.t. $T'(j, \alpha, j')$, there is an i' in Q s.t. $E(i', j')$ and $T(i, \alpha, i')$.

We note that $H(E)$ is formed by removing pairs from E that violate the bisimulation constraints relative to E. We can then define a sequence of relations $E_0, E_1, \ldots$ by taking $E_0 = Q \times Q$ and $E_{x+1} = H(E_x)$. Since $E(i, j)$ is required for (i, j) to be in $H(E)$, it is apparent that this sequence will be monotone decreasing, *i.e.*, $E_{x+1} \subseteq E_x$. It also follows that any fixed-point of H is a bisimulation between F and itself. Therefore, by iterating H on an initial (finite) $E = Q \times Q$ we eventually find a fixed-point (which is therefore also a bisimulation). By Theorem 2, this bisimulation can be used in state space aggregation to produce

a quotient model with states that are action sequence equivalent to the original model.

Further analysis has demonstrated that the resulting bisimulation contains every other bisimulation, and is thus the largest[17] bisimulation between F and itself [Milner, 1990]. As a result, this bisimulation is the bisimilarity relation on Q, and produces the smallest quotient model of any bisimulation when used in state space aggregation.

4.2 Minimizing Markov Decision Processes with Stochastic Bisimilarity

We show here how the direct generalization of the techniques described above for computing bisimilarity yields an algorithm for computing stochastic bisimilarity that in turn is the basis for a model minimization algorithm. Given an MDP $M = \langle Q, A, T, R \rangle$, we define an operator I on binary relations $E \subseteq Q \times Q$ similar to H. Let $I(E)$ to be the set of all pairs i, j such that $E(i, j)$, $R(i) = R(j)$, and for every action α in A and state i' in Q,

$$T(i, \alpha, i'/E) = T(j, \alpha, i'/E) .$$

We can again define a decreasing sequence of equivalence relations $E_0 \supseteq E_1 \supseteq \ldots$ by taking $E_0 = Q \times Q$ and $E_{x+1} = I(E_x)$. Again, the definitions immediately imply that any fixed point of I is a stochastic bisimulation between M and itself. Therefore, by iterating I on an initial (finite) $E = Q \times Q$, we are guaranteed to eventually find a fixed point (which is therefore a stochastic bisimulation). Theorem 7 implies that this stochastic bisimulation can be used in state space aggregation to produce a quotient model containing blocks that are both action sequence and optimal value equivalent to the original model.

The resulting stochastic bisimulation contains every other stochastic bisimulation between M and itself, and is thus the largest stochastic bisimulation between M and itself,[18] the stochastic bisimilarity relation on Q. Aggregation using this relation gives a coarser (smaller) aggregate reduced model than with any other bisimulation. Use of this technique for computing bisimilarity for state space aggregation and model reduction provides a straightforward motivation for and derivation of a model minimization algorithm: simply aggregate bisimilar states to form the coarsest equivalent model, the quotient model under bisimilarity.

4.3 Implementing Model Minimization using Block Splitting

We now describe a method for computing stochastic bisimilarity[19] by repeatedly splitting the state space into smaller and smaller blocks, much like the $I(E)$ operation described above. We start by introducing a desired property for partition blocks that can be checked locally (between two blocks) but that when present globally (between all pairs of blocks) ensures that a bisimulation has been found.

We say that a block B is *stable with respect to block C* if and only if every state p

---

[17] Here, by "largest", we are viewing relations as sets of pairs partially ordered by subset.

[18] We can show that if E contains a bisimulation B, then $I(E)$ must still contain that bisimulation—the key step is to show that $T(i, \alpha, i'/E) = T(j, \alpha, i'/E)$ for any i' in Q, any α in A, and any i and j such that $B(i,j)$.

[19] Our algorithm is a stochastic adaptation of an algorithm in [Lee and Yannakakis, 1992] that is related to an algorithm by [Bouajjani *et al.*, 1992]. All of these algorithms derive naturally from the known properties of bisimilarity in concurrent process theory [Milner, 1990].

in B has the same probability $T(p, \alpha, C)$ of being carried into block C for every action α and the block reward $R(B)$ is well defined. We say that B is *stable with respect to equivalence relation* E if B is stable with respect to every block in the partition induced by E. We say that an equivalence relation E is *stable* if every block in the induced partition is stable with respect to E. These definitions immediately imply that any stable equivalence relation is a bisimulation.

The equivalence relation $I(E)$ can be defined in terms of stability as the relation induced by the coarsest partition (among those refining E) containing only blocks that are stable with respect to E. This partition can be found by splitting each block of E into maximal sub-blocks that are stable with respect to E (i.e. stable with respect to each block of E). To make this concrete, we define a split operation that enforces this stability property for a particular pair of blocks.

Let P be a partition of Q, B a block in P, and C a set of states $C \subset Q$. We define a new partition denoted SPLIT(B, C, P) by replacing B with the uniquely determined sub-blocks $\{B_1, \ldots, B_k\}$ such that each B_i is a maximal sub-block of B that is stable with respect to C. Since B_i is stable with respect to C, for any action α and for states p and q from the same block B_i we have that $T(p, \alpha, C) = T(q, \alpha, C)$ and $R(p)=R(q)$. Since the B_i are maximal, for states p and q from different blocks, either $T(p, \alpha, C) \neq T(q, \alpha, C)$ or $R(p) \neq R(q)$.

The SPLIT operation can be used to compute $I(E)$ by repeated splitting of the blocks of the partition induced by E as follows:

```
Let P' = P = the partition induced by E
For each block C in P
    While P' contains a block B for which P' ≠ SPLIT(B,C,P')
      P' = SPLIT(B,C,P') /* blocks added here are stable wrt. C */
                         /* so need not be checked in While test */
I(E) = the equivalence relation represented by P'
```

We refer to this algorithm as the *partition improvement* algorithm, and to iteratively applying partition improvement starting with $\{Q\}$ as *partition iteration*. However, in partition iteration, suppose a block B has been split so that P' contains sub-blocks $B_1, \ldots, B_k$ of B. Now, splitting other blocks C to create stability with respect to B is no longer necessary since, we will be splitting C to create stability with respect to $B_1, \ldots, B_k$ in a later iteration of I. Blocks that are stable with respect to $B_1, \ldots, B_k$ are necessarily stable with respect to B. This analysis leads to the following simpler algorithm, which bypasses computing I iteratively and computes the greatest fixed point of I more directly:

```
Let P = {Q}  /* trivial one block partition */
While P contains block B & C s.t. P ≠ SPLIT(B,C,P)
    P = SPLIT(B,C,P)
Greatest Fixed point of I = the equivalence relation given by P
```

We refer to this algorithm as the *model minimization algorithm*, and we refer to the $P \neq$ SPLIT(B, C, P) check as the *stability check* for blocks B and C. That model minimization computes a fixed point of I follows from the fact that when all blocks of a partition are stable with respect to that partition, the partition is a bisimulation (and thus a fixed point

of I). The following lemma and corollary then imply that either model minimization or partition iteration can be used to compute the greatest fixed point of I.

Lemma 8.1: Given equivalence relation E on Q and states p and q such that $T(p, \alpha, C) \neq T(q, \alpha, C)$ for some action α and block C of E, p and q are not related by any stochastic bisimulation refining E.

Corollary 8.2: Let E be an equivalence relation on Q, B a block in E, and C a union of blocks from E. Every bisimulation on Q that refines E is a refinement of the partition SPLIT(B, C, E).

Theorem 8: Partition iteration and model minimization both compute stochastic bisimilarity.

By repeatedly finding unstable blocks and splitting them, we can thus find the bisimilarity partition in linearly many splits relative to the final partition size (each split increases the partition size, which cannot exceed that of the bisimilarity partition, so there are at most linearly many splits). The model minimization algorithm performs at most quadratically many stability checks:[20] simply check each pair of blocks for stability, splitting each unstable block as it is discovered. The cost of each split operation and each stability check depends heavily on the partition representation and is discussed in detail later in this paper.

We note that this analysis implies that the partition computed by model minimization is the stochastic bisimilarity partition, regardless of which block is selected for splitting at each iteration of the `While` loop. We therefore leave this choice unspecified.

Figure 4.a shows an MDP in factored representation by giving a DBN with the conditional probability tables represented as decision trees, using the representation developed in [Dean and Kanazawa, 1989] and [Boutilier *et al*., 2000]. Figure 4.b shows the immediate-reward partition for this MDP, which is computed by $I(\{Q\})$. There are two blocks in this partition: states in which the reward is one and states in which the reward is zero. Figure 4.c shows the quotient model for the refined partition constructed by the model minimization algorithm. Aggregate states (blocks of the two partitions) are described as formulas involving fluents, *e.g.*, $\neg S_1 \wedge S_2$ is the set of states in which S_1 is false and S_2 is true. A factored SPLIT operation suitable for finding this quotient model without enumerating the underlying state space is described in Section 4.4.

[20] Observe that the stability of a block C with respect to another block B and any action is not affected by splitting blocks other than B and C, so no pair of blocks need to be checked for stability more than once for each action. Also the number of blocks ever considered cannot exceed twice the number of blocks in the final partition, since blocks that are split can be viewed as internal nodes of a tree. Here, the root of the tree is the block of all states, the leaves of the tree are the blocks of the final partition, and the children of any node are the blocks that result from splitting the block at the node. These facts imply the quadratic bound on stability checks.

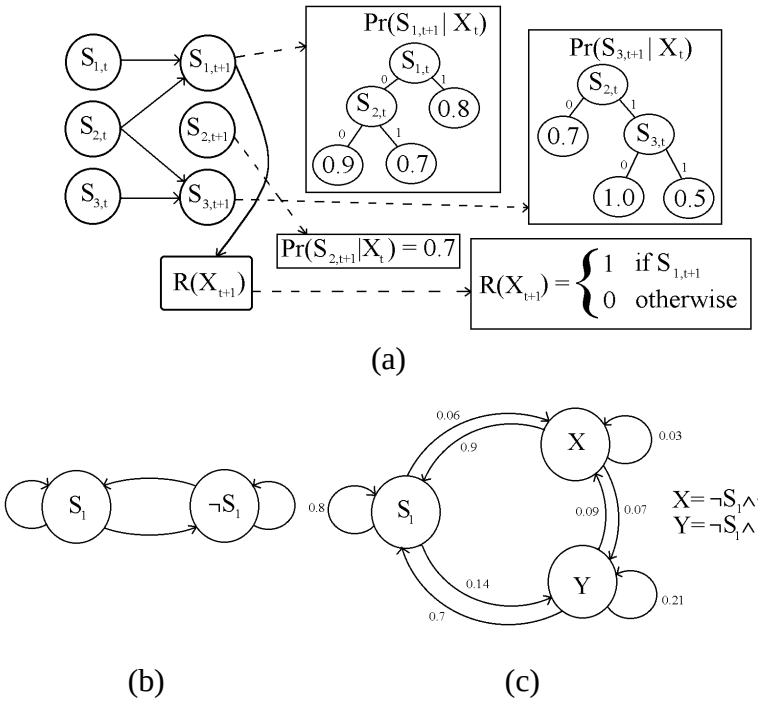

Figure 4. (a) A factored representation of an MDP with three fluents (S_1, S_2, S_3) and only one action. Partitions are shown represented as decision trees. (b) A graph of the immediate reward partition $I(\{Q\})$ of the MDP with nodes representing blocks and arcs representing possible transitions. (c) The quotient model of the coarsest homogeneous partition computed by the model minimization algorithm.

The model-minimization algorithm is given independently of the underlying representation for state-space partitions. However, in order for the algorithm to guarantee finding the target partition, we must have a partition representation sufficiently expressive to represent an arbitrary partition of the state space. Such partition representations may be expensive to manipulate, and may blow up in size. For this reason, partition manipulation operations that do not exactly implement the splitting operation described above can still be of use—typically these splitting operations guarantee that the resulting partition can be represented in a more restrictive partition representation. Such operations can still be adequate for our purposes if, whenever a split is requested, the operation splits "at least as much" as requested.

Formally, we say that a block splitting operation SPLIT* is *adequate* if SPLIT*(*B*, *C*, *P*) is always a refinement of SPLIT(*B*, *C*, *P*). Adequate split operations that can return partitions that are strictly finer than SPLIT are said to be *non-optimal*. The minimization algorithm, with SPLIT replaced by an adequate SPLIT*, is a *model reduction algorithm.* Note that non-optimal SPLIT* operations may be cheaper to implement than SPLIT, even though they "split more" than SPLIT. One natural way to define an

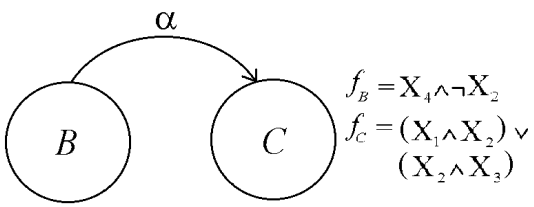

Figure 5. Block formulas used to illustrate splitting for factored representations.

adequate but non-optimal SPLIT* operation is to base the definition on a partition representation that can represent only some possible partitions. In this case, SPLIT* is defined as a coarsest representable refinement of the optimal partition computed by SPLIT. (For many natural representations, e.g., fluentwise partitions, this coarsest refinement is unique.) As shown by the following theorem, the model reduction algorithm remains sound.

Theorem 9: Model reduction returns a stochastic bisimulation.

Corollary 9.1: The optimal policy for the quotient model produced by model reduction induces an optimal policy for the original MDP.

This theorem guarantees us that model reduction will still find an equivalent reduced model. However, we may lose the property that the resulting partition is independent of the order in which we chose to split blocks (i.e., which block is split by the main `while` loop when a choice is present). This property must be proven anew for each SPLIT* operation that is considered, if the property is desired. A theorem and corollary similar to Theorem 9 and Corollary 9.1 can be proven with analogous techniques for partition iteration using an adequate SPLIT* operation.

Some published techniques that operate on implicit representations resemble minimization with adequate but non-optimal splitting operations. We describe some of these techniques and the connection to minimization later, but first we examine the details of our algorithm for a particular factored representation.

4.4 Factored Block Splitting

This subsection describes a method for implementing the SPLIT operation on partitions given a factored representation of the MDP dynamics. The method and factored representation are provided to make concrete the operations involved, not to assert that either the method or the representation is particularly distinguished. Using this splitting method, our model minimization algorithm can construct a reduced model without explicitly enumerating states. The later part of this section gives a detailed example.

We now introduce notation to set up a running example for this section. Let Q be the set of all states, and P a partition of Q. For any block B of states, let f_B be the formula used to represent B. Given blocks B and C in P, we are interested in splitting B to obtain a set of sub-blocks that are stable with respect to C. We replace B with the resulting stable sub-blocks to obtain a refinement of P called P'. Figure 5 depicts the basic objects for our example. We start by focusing on a particular, but arbitrary, action α, and then generalize to multiple actions by computing the intersection of the partitions for each action.

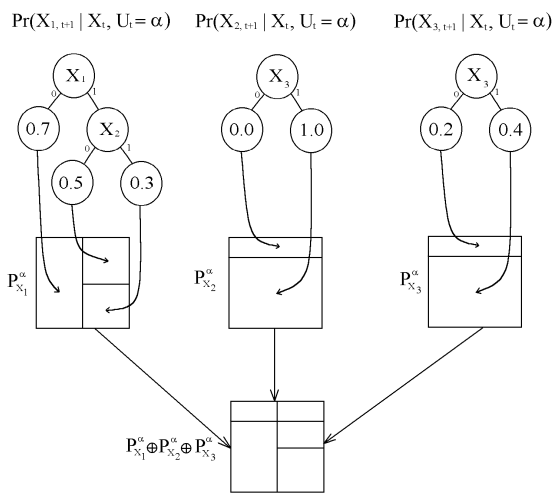

Figure 6. Partial action dynamics for α: decision trees describing the effects of α on the fluents in f_C, and the induced partitions of the state space.

We assume that the state-transition distribution for action α is in factored form—for each fluent, there is a decision tree specifying the conditional probability distribution over the value of the fluent at time *t*, given the state at time *t*–1. Figure 6 illustrates the decision trees for our running example; we only show the decision trees for the three fluents in f_C. In our example trees, the distribution over values is given by a single probability (that of "true"), because there are only two possible values. Note that these decision trees are labeled, factored partitions of the state space. The leaves of the tree correspond to the blocks of the partition—each block is specified by the values assigned to the fluents on the path from the root to the corresponding leaf. These blocks are then labeled with the probability distribution at the corresponding decision-tree leaf.

Each fluent has a decision tree describing its behavior under action α. Consider a subset *F'* of the fluents. We obtain a partition that we refer to as the *partition determining the transition distribution for F' under* α, as follows. The blocks of the partition are given by the intersection of the |*F'*| partitions described by the decision trees for fluents in *F'*. There is a one-to-one correspondence between blocks in the new partition and sets of blocks from the |*F'*| partitions (one from each) with non-empty intersections. We label each block of this new "overlaid" partition with the product of the distribution labels on the blocks in the corresponding set of blocks. This partition is then a refinement of the partition under α for any of the fluents in *F'*. States in the same block of this overlaid partition have the same probability of transitioning (under action α) to any block of the partition Fluentwise(*F'*) defined in Section 2.3. Here as elsewhere in our discussion, we simultaneously treat states as elements of *Q* that can be contained in a block of a state space partition, and as assignments of values to fluents that can satisfy the formula associated with a given block of a partition.

We denote the labeled partition for fluent X_i under action α as $P^{\alpha}_{X_1}$. For example, the decision tree for X_1 shown in Figure 6 gives us

$$P^{\alpha}_{X_1} = \{B_1, B_2, B_3\},$$

where the formulas associated with the blocks of $P^{\alpha}_{X_1}$ are

$$f_{B_1} = \neg X_1 \qquad f_{B_2} = X_1 \wedge \neg X_2 \qquad f_{B_3} = X_1 \wedge X_2$$

The probability distribution for X_1 under action α for the blocks of $P^{\alpha}_{X_1}$ is given by

$$\Pr(X_{1,t+1} \mid X_t, U_t = \alpha) = \begin{cases} 0.7 & X_t \in B_1 \\ 0.5 & X_t \in B_2 \\ 0.3 & X_t \in B_3 \end{cases}.$$

Note that we can group all leaves of the decision tree for a given fluent that share the same probability distribution label into a single block in the partition for the fluent. For example, if the probability distribution for X_1 at the leaf for both blocks B_1 and B_2 in $P^{\alpha}_{X_1}$ were 0.7, then we would group all the states in blocks B_1 and B_2 into a block B', giving

$$P^{\alpha}_{X_1} = \{B', B_3\}, \qquad f_{B'} = (\neg X_1) \vee (X_1 \wedge \neg X_2), \qquad f_{B_3} = X_1 \wedge X_2, \text{ and}$$

$$\Pr(X_{1,t+1} \mid X_t, U_t = \alpha) = \begin{cases} 0.7 & X_t \in B' \\ 0.3 & X_t \in B_3 \end{cases}.$$

For each fluent X_i, the partition $P^{\alpha}_{X_i}$ groups states that behave the same under action α with regards to X_i. However, what we want is to group states in B that behave the same under action α with respect to C. Since C is specified using a formula f_C, we need only concern ourselves with fluents mentioned in f_C, as the other fluents do not influence whether or not we end up in C. If we take the intersection of all the partitions for each of the fluents mentioned in f_C, we obtain the coarsest partition that is a refinement of all those fluent partitions. This partition distinguishes between states with different probabilities of ending up in C. We can then restrict the partition to the block B to obtain the sub-blocks of B where states in the same sub-block all have the same probability of ending up in C after taking action α. Therefore, if Fluents(f_C) is the set of all fluents appearing in f_C, the partition determining the transition distribution for Fluents(f_C) under α makes all the necessary state distinctions.

The procedure Block-split() shown in Figure 7 computes the coarsest partition of B that is a refinement of all the partitions associated with the fluents in f_C and the action α. It does so by first computing the coarsest partition of Q, which we will denote P_Q, with this property, and then intersecting each block in this partition with B. (In terms of representing blocks as formulas, intersection is just conjunction.) Applying this to our ongoing example gives the following partitions:

$$P^{\alpha}_{X_1} = \{ X_1 \wedge X_2,\ X_1 \wedge \neg X_2,\ \neg X_1 \} \qquad P^{\alpha}_{X_2} = \{X_3, \neg X_3\} \qquad P^{\alpha}_{X_3} = \{X_3, \neg X_3\}$$

$$P_Q = \{\ X_1 \wedge X_2 \wedge X_3,\quad X_1 \wedge X_2 \wedge \neg X_3,\quad X_1 \wedge \neg X_2 \wedge X_3,\quad X_1 \wedge \neg X_2 \wedge \neg X_3,\quad \neg X_1 \wedge X_3,\quad \neg X_1 \wedge \neg X_3 \}$$

Intersecting each block of P_Q with f_B (eliminating empty blocks) computes the final partition of B given by

$$\{ X_1 \wedge \neg X_2 \wedge X_3 \wedge X_4,\quad X_1 \wedge \neg X_2 \wedge \neg X_3 \wedge X_4,\quad \neg X_1 \wedge \neg X_2 \wedge X_3 \wedge X_4,\quad \neg X_1 \wedge \neg X_2 \wedge \neg X_3 \wedge X_4\}.$$

```
Block-split(B, C, α)
    return { f_B ∧ f ∧ f_R | f ∈ Partition-determining(Fluents(f_C), α),
                             f_R ∈ Reward partition,
                             and f_B ∧ f ∧ f_R is satisfiable };

Partition-determining(F, α)     /*  the partition determining the
                                    fluents in F */
    if F = ∅ then return {true};
    for some X ∈ F,
      return { f ∧ f_B' | B' ∈ P_X^α,
                          f ∈ Partition-determining(F–{X}, α), and
                          f ∧ f_B' is satisfiable};
```

Figure 7. Procedure for partitioning block B with respect to block C and action α

This procedure runs, in the worst case, in time exponential in the number of fluents mentioned in f_C.[21] As with most factored MDP algorithms, in the worst case, the factoring gains us no computational advantage.

One adequate but non-optimal splitting operation that works on the factored representation is defined in terms of the procedure Block-split() as

$$\text{SPLIT}^*(B, C, P) = (P - \{B\}) \cup (\cap_{\alpha \in A} \text{Block-split}(B, C, \alpha)).$$

We refer to SPLIT* defined in this manner as S-SPLIT, abbreviation "structure-based splitting". Structure-based splitting the exact transition probabilities assigned to blocks of states. This splitting method splits two states if there is *any* way of setting the quantifying parameters that would require splitting the states. S-SPLIT is non-optimal because it cannot exploit "coincidences" in the quantifying parameters to aggregate "structurally" different states.

In order to implement an optimal split, we need to do a little more work. Specifically, we have to combine blocks of Block-split(B, C, α) that have the same probability of ending up in C. Situations where we must combine such blocks in order to be optimal arise when an action, taken in different states from B, affects the fluents in f_C differently, but "coincidentally" has the same overall probability of ending up in block C from the different source states. For example, suppose action α, taken in state p in B, has a 0.5 probability of setting fluent X_1, and always sets fluent X_2; however, when α is taken in state q in B, it has a 0.5 probability of setting fluent X_2, and always sets fluent X_1. If block C has formula $X_1 \wedge X_2$ both state p and state q have a 0.5 probability of transitioning to block C under action α. However, p and q must be in separate blocks for each of the fluents in the formula $X_1 \wedge X_2$, since α affects both X_1 and X_2 differently at p than at q—hence, Block-split() will partition p and q into different blocks, even though they behave

[21] The order in which the fluents are handled can dramatically affect the run time of Partition-determining() if inconsistent formulas are identified and eliminated on each recursive call.

Block		P(X_1)	P(X_2)	P(X_3)	P(C_1)	P(C_2)	P(C_3)	P(f_C)
B_1	$X_1 \wedge \neg X_2 \wedge X_3 \wedge X_4$	0.5	1.0	0.02	0.010	0.490	0.010	0.510
B_2	$X_1 \wedge \neg X_2 \wedge \neg X_3 \wedge X_4$	0.5	0.6	0.50	0.150	0.150	0.150	0.450
B_3	$\neg X_1 \wedge \neg X_2 \wedge X_3 \wedge X_4$	0.7	1.0	0.02	0.014	0.686	0.006	0.706
B_4	$\neg X_1 \wedge \neg X_2 \wedge \neg X_3 \wedge X_4$	0.7	0.6	0.50	0.210	0.210	0.090	0.510

Figure 8: Optimal Split Computations for the ongoing example. We show, for each B_i, the probability P(X_i) of setting each fluent X_i in fluents(C), when acting in B_i. The right four columns use these values to compute the probability P(C_i) of landing in each block C_i of Fluentwise(C), and then the probability P(f_C) of landing in C itself, in each case when acting in each B_i.

the same with respect to C. To compute the coarsening of Block-split(B, C, α) required to obtain optimal splitting, we first consider a particular partition of the block C.

The partition of C that we use in computing an optimal split of B is the fluentwise[22] partition Fluentwise(Fluents(C)), restricted to C. This partition has a block for each assignment to the fluents in Fluents(C) consistent with f_C. We denote this partition as Fluentwise(C). In our example, $f_C = (X_1 \wedge X_2) \vee (X_2 \wedge X_3)$ so Fluentwise(C) = $\{X_1 \wedge X_2 \wedge X_3,\ X_1 \wedge X_2 \wedge \neg X_3,\ \neg X_1 \wedge X_2 \wedge X_3\}$ which we shall call C_1, C_2, and C_3, respectively.

The probability of transition from $B_j \in$ Block-split(B, C, α) to $C_i \in$ Fluentwise(C) is defined as

$$\Pr(X_{t+1} \in C_j \mid X_t \in B_i\ ,\ U_t = \alpha) = \Pr(X_{t+1} \in C_j \mid X_t = p,\ U_t = \alpha),$$

where p is an arbitrary state in B_i. The choice of p does not affect the value of $\Pr(X_{t+1} \in C_j \mid X_t \in B_i,\ U_t = \alpha)$ by the design of Block-split(). We can compute these probabilities by multiplying the appropriate entries from the probability distributions for the fluents in f_C and thus induce a labeling for the blocks of the partition returned by Block-split(),

$$\Pr(X_{t+1} \in C \mid X_t \in B_i,\ U_t = \alpha) \ = \sum_{C_j \in \mathit{Fluentwise}(C)} \Pr(X_{t+1} \in C_j \mid\ X_t \in B_j,\ U_t = \alpha).$$

To compute the optimal split, we group together those blocks in $\bigcap_{\alpha \in A}$ Block-split(B, C, α) that have the same block transition distributions, *i.e.*, B_i, $B_j \in \bigcap_{\alpha \in A}$ Block-split(B, C, α) are in the same block of SPLIT(B, C, P) if and only if

$$\Pr(X_{t+1} \in C \mid X_t \in B_i,\ U_t = \alpha) = \Pr(X_{t+1} \in C \mid X_t \in B_j,\ U_t = \alpha), \text{ for all } \alpha.$$

Once again, we note that in the worst case, the additional work added to compute an optimal split with this method is exponential in the original MDP representation size because Fluentwise(C) would have to be enumerated explicitly. To complete our example, we show these calculations in Figure 8, the final column of which indicates that we can combine the blocks labeled B_1 and B_4, since they both have the same probability of transitioning to block C. As a result, we obtain the following partition of B: $\{ X_1 \wedge \neg X_2 \wedge \neg X_3$

[22] See Section 2.3.

$\wedge X_4$, $\neg X_1 \wedge \neg X_2 \wedge X_3 \wedge X_4$, $(X_1 \wedge \neg X_2 \wedge X_3 \wedge X_4) \vee (\neg X_1 \wedge \neg X_2 \wedge \neg X_3 \wedge X_4)$ }.

4.5 Hardness of Model Minimization with Factored Partitions

The difficulty of optimal splitting is implied by the following complexity result.

Definition 1: The *bounded-size model-minimization decision problem* is:

Given a number k represented in unary notation and a factored MDP M with a minimal model of k or fewer states, determine whether the minimal model of M has exactly k states.

Theorem 10: The bounded-size model-minimization problem is NP-hard.

It is worth noting that the different non-optimal SPLIT* operations make different tradeoffs between ease of computation and amount of reduction that can be achieved in the reduced model. Also, some non-optimal SPLIT* definitions guarantee that the resulting partition can be represented compactly, as we will see in Section 4.6.

Theorem 10 shows that model minimization will be expensive in the worst case, regardless of how it is computed, even when small models exist. In addition, since our original algorithm presentation in [Dean and Givan, 1997] it has been shown that the factored-stability test required for the particular algorithm we present (and implicit in computing SPLIT) is also quite expensive to compute, being $\text{coNP}^{\text{C=P}}$-hard [Goldsmith and Sloan, 2000].[23] This result does not directly imply hardness for the bounded-size model minimization problem (i.e. Theorem 10), because there could be other algorithms for addressing that problem without using SPLIT.

4.6 Non-optimal Block Splitting for Improved Effectiveness

We discuss three different non-optimal block splitting approaches and the interaction between these approaches and our choice of partition representation as well as the consequent improvement in effectiveness. The optimal SPLIT defined above requires a general-purpose partition representation to represent the partitions encountered during model reduction—e.g. the DNF representation discussed in Section 2.3. Each of the alternative non-optimal SPLIT* approaches can guarantee that the resulting partition is representable with a less expressive but more compact representation, as discussed below.

We motivate our non-optimal splitting approaches by noting that the optimal factored SPLIT operation described in Section 4.4 has two phases, each of which can independently take time exponential in the input size. The first phase computes Block-split(B, C, α) for each action α, and uses it to refine B, defining the partition S-SPLIT(B, C, P). The second phase coarsens this partition, aggregating blocks that are "coincidentally" alike for the particular quantifying parameters (transition probabilities and rewards) in the

[23] [Goldsmith and Sloan, 2000] also show that the complexity of performing a test for an approximate version of stability, ε-stability, for an arbitrary partition is coNP^{PP}-complete. (ε-stability, is a relaxed form of stability defined in [Dean *et al.*, 1997]).

model. Our non-optimal splitting methods address each of these exponential phases, allowing polynomial-time computation of the partition resulting from that phase.

The first non-optimal approach we discuss guarantees a fluentwise-representable partition—recall from Section 2.3 that a fluentwise partition can be represented as a subset of the fluents where the blocks of the partition correspond to the distinct truth assignments to that subset of fluents. We define the "fluentwise split" F-SPLIT(*B, C, P*) to be the coarsest refinement of SPLIT(*B, C, P*) that is fluentwise representable. F-SPLIT(*B, C, P*) is the fluentwise partition described by the set of all fluents *X* such that there are two states differing only on *X* that fall in different blocks of SPLIT(*B, C, P*). Equivalently, F-SPLIT(*B, C, P*) is the fluentwise partition described by the set of all fluents *X* that are present in every DNF description of SPLIT(*B, C, P*). As with SPLIT(B,C,P), the function F-SPLIT(B,C,P) can be computed in two phases. The first phase intersects partitions from the action definitions, returning the coarsest fluentwise refinement of the result. The second phase combines blocks in the resulting partition (due to "coincidences"), and again takes the coarsest fluentwise refinement, to yield the desired partition. The first phase can be carried out efficiently in polynomial time in the size of the output, but the second phase appears to require time possibly exponential in its output size, because it appears to require enumerating the blocks of the first-phase output.

To avoid the exponential time required in the second phase to detect "coincidences" that depend on the quantifying parameters, we need to define a "structural" notion of block stability—one that ignores the quantifying parameters. Because our factored representation defines transition probabilities one fluent at a time, we will define structural stability in a similar fluentwise manner.

We say that a block *B* of a partition *P* is *fluentwise stable with respect to fluent X* if and only if for every action α, *B* is a subset of some block of the partition $T_F(\alpha, X)$. The block *B* is termed *fluentwise stable with respect to block C* if *B* is fluentwise stable with respect to every fluent mentioned in <u>every</u> DNF formula describing block *C*. We call a partition *P fluentwise stable* if every block in the partition is fluentwise stable with respect to every other block in the partition. It is straightforward to show that the "structural split" S-SPLIT(*B, C, P*), as defined above in Section 4.4, is the coarsest refinement of SPLIT(*B, C, P*) for which each sub-block of *B* is fluentwise stable with respect to *C*.

The operation S-SPLIT is adequate and is computed using Block-split() for each action, as described in Section 4.4, assuming that each block formula in the input partition representation is simplified (in the sense that any fluent mentioned must be mentioned to represent the block). This assumption holds for blocks represented as conjunctions of literals, as in decision-tree partitions. Under this assumption S-SPLIT can be computed in time polynomial in the size of its input formulas plus the number of new blocks introduced (which may be exponential in the input size). Analysis of S-SPLIT guarantees that if each input block is describable by a conjunction of literals then so are the blocks of the output partition, ensuring that the inputs are conjunctions of literals, if each partition in the original factored MDP definition is so represented (e.g. if decision

tree partitions are used to define the MDP[24]), as long as all block splitting is done with S-SPLIT. This guarantee allows model reduction with S-SPLIT to use this simpler representation of partitions. With S-SPLIT the result of reduction is also not order-dependent, unlike some adequate, non-optimal splits (see Section 4.3).

Theorem 11: Given a partition *P*, there is a unique coarsest fluentwise-stable stochastic bisimulation refining *P*. Iterating S-SPLIT using model reduction or partition iteration starting from *P* computes this bisimulation regardless of the order of block splitting.

To avoid exponential model-reduction time even when the resulting model is exponentially large, we can combine the above two concepts. We call the resulting "fluentwise structural" split FS-SPLIT(*B*, *C*, *P*). FS-SPLIT(*B*, *C*, *P*) computes the coarsest fluentwise-representable refinement of SPLIT(*B*, *C*, *P*) such that each sub-block of *B* is fluentwise stable with respect to *C*. The split operation FS-SPLIT is adequate and computable in time polynomial in the size of *M*, even for factored *M*, and the resulting partition is again independent of the order of splitting.

Theorem 12: Given a partition *P*, there is a unique coarsest stochastic bisimulation refining *P* even under the restriction that the partition be both fluentwise stable and fluentwise representable. Iterating FS-SPLIT using model reduction or partition iteration starting from *P* computes this bisimulation regardless of the order of block splitting.

A variant of S-SPLIT that is closer to the optimal SPLIT can be derived by observing that there is no need to split a block *B* to achieve fluentwise stability relative to a destination block *C* when the block *B* has a zero probability of transitioning to the block *C*. This refinement does not affect FS-SPLIT due to the bias towards splitting of the "fluentwise" partition representation used, but adding this refinement does change S-SPLIT. The resulting split operation, which we call R-SPLIT, is significant in that it is implicit in the previously published factored MDP algorithms in [Boutilier *et al*., 2000].

We define the *regression region* for a block *B* to be the block containing those states *i* such that $T(i, \alpha, B)$ is non-zero. A block *B* is said to be *regression stable with respect to block C* if *B* is either entirely contained in the regression region of *C* and *B* is fluentwise stable with respect to *C* or *B* does not overlap the regression region of *C*. The "regression" splitting operation R-SPLIT(*B*, *C*, *P*) is the coarsest refinement of SPLIT(*B*, *C*, *P*) such that each sub-block of *B* is regression stable with respect to *C*. We say a partition *P* is regression stable if every block of *P* is regression stable with respect to every other block of *P*. R-SPLIT can be calculated using a modification of the Block-split function, given in Figure 7. For each action α, replacing the call Partition-determining(Fluents(*C*), α) with the call Regression-determining(f_C, α), invoking the pseudo-code shown in Figure 9. We note that R-SPLIT, unlike S-SPLIT, depends on the

[24] It is worth noting that decisions trees as used in this paper are less expressive than the disjoint conjunctions of literals representation. That is to say there exist sets of disjoint conjunctions of literals that represent partitions not representable with decision trees, e.g. $\{A\wedge\neg B, B\wedge\neg C, C\wedge\neg A, A\wedge B\wedge C, \neg A\wedge\neg B\wedge\neg C\}$.

```
Regression-determining(f_C, α)
  P_C = { b | b ∈ Partition-determining(Fluents(f_C),α) and
            Pr(f_C true in next state | current state in b) > 0 }
  Q_0 = Q - ⋃_{b∈P_C} b    /* states with zero trans. probability to C */
  Return {Q_0} ∪ P_C
```

Figure 9. Function used in computing R-SPLIT.

specific transition probabilities (i.e. whether each is zero or not), not just the partitions used in defining *T*. Given a partition (and factored MDP) using only blocks described by conjunctions of literals, $I_{\text{R-SPLIT}}$ returns another such partition[25]. Unlike S-SPLIT, we do not have a method for computing R-SPLIT in worst-case polynomial-time in the number of blocks in the output partition (similarly, the corresponding algorithms in [Boutilier *et al*., 2000], as discussed below in Section 5, are not polynomial in the output size).

Theorem 13: Given a partition *P*, there exists a unique coarsest regression-stable stochastic bisimulation refining *P*.

It turns out that this target partition can be computed by iterating R-SPLIT, as expected, but that the partition found may depend on the order in which splitting is done unless we restrict the starting partition representation, as follows.

Theorem 14: Let *M* be a factored MDP with all partition blocks represented as conjunctions of literals. Given a starting partition *P* also so represented, iterating R-SPLIT using partition iteration computes the coarsest regression-stable stochastic bisimulation refining *P*, regardless of the order in which blocks are selected for splitting.

5 Existing Algorithms

We briefly describe several existing algorithms that operate on factored representations, and relate these algorithms to model reduction/minimization. Our model minimization and reduction methods provide a means for automatically converting a factored MDP into a familiar explicit MDP by aggregation. The resulting explicit MDP can then be manipulated with traditional solution algorithms, and the resulting solutions induce corresponding solutions in the original factored MDP. In this process, the aggregation analysis is completely separate from the later value or policy computations.

Previous work by [Boutilier *et al*., 2000] gives algorithms that interleave value and policy computations with aggregation computations, by giving factored forms of the traditional MDP solution methods. This interleaved approach has advantages in some cases where the minimal model is too expensive to compute, because exploiting value

[25] However, single calls to R-SPLIT can return partitions not representable with conjunctions of literals. $I_{\text{R–SPLIT}}$ cannot—this difference is surprising and is a consequence of the fact that every state must transition somewhere and thus be in some regression region. See the proof of Lemma 16.1 for more detail.

computations based on partial minimization may make it possible to avoid full minimization (e.g. sometimes value-based algorithms can compute the minimal model for just the optimal policy without computing the full minimal model).

Here we argue that two previously published methods, state space abstraction and structured successive approximation (SSA), can be alternatively viewed as model reduction followed by traditional MDP solution [Boutilier and Dearden, 1994]. Model reduction provides an explication of the state equivalence properties being computed by these techniques, as well as a description of the techniques that separates the partition manipulation from the value computation (relying on traditional techniques for the latter).

We then discuss two other previous methods [Boutilier *et al*., 2000], structured policy iteration (SPI) and structured value iteration (SVI), that can obtain advantages over direct model reduction due to the interleaving of value computations with partition manipulation. Finally, we discuss connections between model minimization and a previously published factored POMDP solution technique, and relate our work to the SPUDD system [Hoey *et al*. 1999]. There is other related work on factored MDP solution that we do not analyze here, e.g., [Baum and Nicholson, 1998] [Koller and Parr, 2000].

5.1 State-Space Abstraction

State-space abstraction [Boutilier and Dearden, 1994] is a means of solving a factored MDP by generating an equivalent reduced MDP formed by determining which fluents values are necessarily irrelevant to the solution. As presented by [Boutilier and Dearden, 1994] the method handles synchronic effects[26,27]—here we address the restriction of that method to factored MDPs represented without synchronic effects. Inclusion of synchronous effects does not increase expressiveness, but may result in a polynomial reduction in the size of the representation [Littman, 1997]. We discuss the extension of our minimization technique to handle synchronic effects in Section 6.3. Pseudo-code for the state-aggregation portion of state-space abstraction is given in Figure 10. Throughout the code, the inferred partition of the state space is fluentwise representable and is maintained as a set of fluents—where every truth assignment to the set is a block of the partition. The method for selecting the fluents determining the partition is described in [Boutilier and Dearden, 1994] as finding the "relevant fluents"—this selection is performed by the procedure Add-relevant.

Here we show that the method in the pseudo-code for determining fluent relevance is effectively a fluentwise-stability check; exactly the check performed by FS-SPLIT. Fluents are added to the set of relevant fluents whenever the current partition is not fluentwise stable (for lack of those fluents). We note that one difference between Add-relevant and FS-SPLIT is that Add-relevant effectively checks the stability of all blocks in the current partition simultaneously rather than just one block; in fact,

[26] See footnote 8 on page 2.

[27] The representation given in [Boutilier and Dearden '94] does not explicitly mention handling synchronic effects. Synchronic effects are achieved in that representation when the "synchronized variables" are included in the same *aspect* when the action is described.

```
State-Space-Abs()
  F_R = Fluents(R)
  do
     F_IR = F_R
     F_R = Add-relevant(F_IR)
  while (F_IR ≠ F_R)
      return F_R
```

```
Add-relevant(F)
      Return F ∪ ⋃_{f∈F, a∈A} Fluents(T_{a,f})
```

Figure 10. Pseudo-code for the aggregation portion of the state space abstraction algorithm, following [Boutilier and Dearden, 1994]. The reward partition is given by R, the action space by A, and the transition distributions by T ($T_{a,f}$ is a partition of the state space where states in the same block have equal probability of setting fluent f under action a). Each partition is represented using a decision tree. Given such a tree t, Fluents(t) gives the set of fluents used in any test in the tree. The F variables are fluentwise-representable state-space partitions represented in a compact form as a set of fluents.

Add-relevant computes the same partition as the iterative use of FS-SPLIT in partition improvement. We write $I_{\text{FS-SPLIT}}(P)$ for the partition returned by the partition improvement method of section 4.3, with SPLIT replaced by FS-SPLIT for splitting and block stability checking—we note the $I_{\text{FS-SPLIT}}(P)$ refines $I(P)$ and that by Theorem 12 we reach a bisimulation by iterating $I_{\text{FS-SPLIT}}(P)$ to a fixed point.

Lemma 15.1: Given a fluentwise partition P and a minimal tree-represented factored MDP M, the partition computed by Add-relevant(P) is the partition $I_{\text{FS-SPLIT}}(P)$.

As a result, we conclude that iterating Add-relevant, as in state-space abstraction, is equivalent to iterating FS-SPLIT as in model reduction.

Theorem 15: Given a minimal tree-represented MDP, model reduction using FS−SPLIT yields the same partition that state-space abstraction yields, and does so in polynomial-time in the MDP representation size.

[Boutilier and Dearden, 1994] also describe a method of approximation by limiting the fluents that are considered relevant to the reward partition—this idea can also be captured in the model reduction framework using ideas like those in section 6.2.

5.2 Structured Stochastic Dynamic Programming—Overview

Policy iteration is a well-known technique for finding an optimal policy for an explicitly represented MDP by evaluating the value at each state of a fixed policy and using those values to compute a locally better policy. Iterating this process leads to an optimal policy [Puterman, 1994]. In explicit MDPs, the evaluation of each fixed policy can be done with another well-known algorithm called *successive approximation*, which computes the *n*-step-to-go value function for the policy for each *n*—converging quickly to the infinite-horizon value function for the policy. A related technique, *value iteration*, computes the *n*-step-to-go value function for the optimal policy directly, for each *n.* Both successive approximation and value iteration converge in the infinite limit to the true value function, and a stopping criterion can be designed to indicate when the estimated

values are within some given tolerance [Puterman, 1994].

[Boutilier *et al.*, 2000] describe variants of policy iteration, successive approximation, and value iteration designed to work on factored MDP representations, called *structured policy iteration* (*SPI*), *structured successive approximation* (*SSA*), and *structured value iteration* (*SVI*), respectively. As we discuss in detail below, SSA can be understood as a variant of model reduction using the regression splitting operation R-SPLIT described in Section 4.6. Single iterations of SPI can also be understood in this manner: the policy improvement phase can be described using a variant of model reduction, so that SPI can be viewed as iterating policy improvement and SSA, each a model reduction.

These methods can be viewed as performing partition manipulation simultaneously with value and/or optimal policy computation—here we will indicate the connection between model reduction and the partition manipulations performed by these algorithms. If model reduction is used, the value and/or policy computations are performed on the aggregate model after reduction, using standard explicit-model techniques. We note that removing the value computations from these algorithms yields substantially simpler code; however, computing value functions and policies during reduction allows their use "anytime" even if reduction is too expensive to complete. The interleaved value computations also allow the aggregation of states that are not equivalent dynamically under all actions. The guarantee is only that the value will be the same for the optimal[28] actions (which will still remain optimal) but the aggregated model may not be equivalent to the original model for other actions. Determining which actions are optimal to enable this extra aggregation requires maintaining information about state values.

SVI is closely similar to model reduction when the tree simplification phase, discussed below, is omitted; tree simplification is generally made possible by the interleaved value computations, and can result in significant savings. Each iteration of SPI is understood using model reduction restricted to the current policy; however, the full iterative procedure is quite different from model reduction followed by explicit policy iteration. Informally, this is because SPI performs aggregation relative to the different specific policies encountered, whereas model minimization or reduction aggregates relative to all policies (states must be separated if they differ under *any* policy).

With both policy and value iteration, model reduction has an advantage in cases where the MDP parameters (but not the tree structure, i.e., the partitions) may change frequently, as in some machine learning settings where the parameters are being learned, for example. In such cases, the reduced model does not change when the parameters change[29], so no re-aggregation needs to be done upon parameter change. This observation suggests omitting tree simplification from SVI in such cases.

Another example where model reduction has an advantage over SPI/SVI arises

[28] Here, "optimal" refers to being the optimal initial action in a finite horizon policy, where the horizon is extended on each iteration of the method.

[29] Assuming an appropriate split operation is used (R-SPLIT or S-SPLIT, for example). If R-SPLIT is being used, the given "structure" must indicate which parameters are zero and which are non-zero. We note that exact model minimization (as opposed to reduction) produces a result that can depend heavily on the model parameters, not just on the structure of parameter dependency.

with "exogenous events". [Boutilier *et al*., 2000] mentions the difficulty in capturing "exogenous events" such as external user requests in the SPI/SVI approach—such requests have the effect of changing the parameters of the reward function, but not the structure, and typically require re-computing the entire solution when using SPI/SVI. In contrast, the model reduction approach does not require any new partition manipulation upon changing the reward parameters, since the reduced model is unchanged; only the explicit reduced-model solution needs to be re-computed, by traditional methods. One contribution of our work is in explicating the SPI/SVI methods by separating analysis of the value computations from analysis of the partition manipulations, as well as connecting the latter to the literature on concurrent processes and automata theory.

Although the value computations included in SPI and SVI differentiate these methods from model reduction, our methods can still be used to explicate the partition manipulations performed by these algorithms. In particular, using the model-reduction form of SSA we construct a model-reduction presentation of SPI below. Following [Boutilier *et al*., 2000], throughout this section we assume that all factored MDPs are represented using decision trees for the partitions involved in defining the reward and action-transition functions. Moreover, we assume that these trees are *minimal* in the following sense: if a fluent appears in a tree, then the tree could not be modified by simply deleting that fluent (and replacing it with either sub-tree) without changing the function represented by the tree. Minimality in this sense is easy to enforce, and without minimality, the algorithms in [Boutilier *et al*., 2000] may do more splitting than our methods.

5.3 Structured Stochastic Dynamic Programming—Details

Partial pseudo-code for the SSA, SPI, and SVI algorithms is shown in Figure 11. Here we show only the partition-manipulation aspects of the algorithms, and only briefly indicate the somewhat complex associated value computations. We provide pseudo-code for these algorithms for reference and for grounding our theorems below, but a full appreciation of this section requires familiarity with [Boutilier *et al*., 2000].

We begin our analysis of the connection between SSA/SVI/SPI and model reduction by showing that the partition computed by the function PRegress is closely related to the partition computed by the function Regression-determining presented earlier, in Figure 9. Regression-determining computes factored block splitting.

Lemma 16.1: Let V be a tree-represented value function, where P_V is the partition given by the tree. Let α be an action, and for any block C of P_V, let Φ_C denote the conjunction of literals describing C. We then have the following.

> The partition computed by PRegress(V, α) is the intersection over all blocks C of P_V of Regression-determining(Φ_C, α).

The key subroutines Regress-policy and Regress-action compute factored state-space partitions identical to those computed by the I operator (see section 4.2) under the following assumptions: first, the only actions available are those under consideration (ei-

```
PRegress(V, a)
  If (V.Tree = single leaf)
    P.Tree = single leaf (represents {Q})
    P.Label = Maps Q to {}
    Return P
  x = Fluent-Tested-at(Root(V.Tree))
  Px.Tree = Px|a.Tree
  For each xi in Val(x)
    Vxi = SubTree(V, xi)
    Pxi = PRegress(Vxi, a)
  Split each block B in Px.Tree by:
    T = ∩Trees({Pxi|Pr(xi in
Px|a.Label(B))>0})
    Px.Tree = Replace(B,T.Tree,Px.Tree)
  Maintain Px.Label as set of distributions
  over single fluent values

  Return Px

Regress-action(V, a)
  PV,a = PRegress(V, a)
  QV,a.Tree = ∩Trees({R,PV,a })
  Label each block of QV,a.Tree by computing
  the Q value using PV,a.Label, V, and R

  Return QV,a

Regress-policy(V, π)
  QV,π.Tree = π.Tree
  For each action a
    QV,a = Regress-action(V, a)
  For each block B of π.Tree
    a = π.Label(B)
    QV,π.Tree = Replace(B, QV,a.Tree, QV,π.Tree)
    Label new blocks of QV,π from QV,a.Label
  Return QV,π
```

```
SSA(π)
  V0,π = R, k = 0
  Until (similar(Vk,π, Vk-1,π))
    Vk+1,π = Regress-policy(Vk,π, π)
    k = k+1
  Return Vk,π

SPI(π')
  While (π' ≠ π)
    π = π'
    Vπ = SSA(π)
    For each action a
      QVπ,a = Regress-action(Vπ, a)
      π'.Tree=
∩Trees({QVπ,a.Tree,π'.Tree})
    π'.Label = λb.argmaxa(QVπ,a(b))
    π' = Simplify-tree(π')
  Return π and Vπ

SVI()
  V0 = R, k = 0
  Until (similar(Vk, Vk-1)
    Vk+1.Tree = Vk.Tree
    For each action a
      QVk,a = Regress-action(Vk,a)
      Vk+1.Tree = ∩Trees({QVk,a.Tree,
                          Vk+1.Tree})
    Vk+1.Label = λb.max(QVk,a(b))
    Vk+1 = Simplify-tree(Vk+1)
    k = k+1
  π.Tree = Vk.Tree
  π.Label = λb.argmaxa(QVk,a(b))
  π = Simplify-tree(π)
  Return π and Vk
```

Figure 11. Partial pseudo-code for the SSA, SPI, and SVI algorithms, following [Boutilier *et al.*, 2000]. Boxed italicized comments refer to omitted code. Mappings over the state space are represented with decision trees as labeled factored state-space partitions—if M is such a mapping then M.Tree gives the partition description as a tree, M.Label gives the labeling as a mapping from the blocks of M.Tree to the range of the mapping, and M(b) gives the value of mapping M on any state *i* in block b (this value must be independent of *i*). Examples of such mappings are Q-functions (Q), value functions (V), policies (π), and factored MDP parameters (the reward function, R, and the effects of action a on fluent x, $P_{x|a}$). The function $\cap$Trees takes a set of trees and returns a decision tree representing the intersection of the corresponding partitions. The function Replace(B,P_1,P_2) replaces block B in state-space partition P_2 with the blocks of B $\cap$ P_1, returning the resulting partition (each partition is again represented as a tree). Simplify-tree() repeatedly removes tests where all the branches lead to identical sub-trees.

ther the single action specified for Regress-action, or the actions specified by the policy for Regress-policy); and second, to improve effectiveness and stay within the decision-tree representation, all block-splitting is done with the structural split operation R-SPLIT. Regress-policy also forcibly splits apart states that select different actions, even if those actions behave identically (see the line $Q_{V,\pi}$.Tree = π.Tree in Regress-policy).

To formalize these ideas, we need a method of enforcing the first assumption concerning the available actions. For a fixed policy π and MDP *M*, we define the π-*restricted*

MDP M_π to be the MDP M modified to have only one action that at each state q has the same transition behavior as $\pi(q)$ in M. To model the restriction to a single action α in Regress-action, we consider the policy π_α that maps every state to action α, and then use M_{π_α} to restrict to that single action everywhere.

We now define $I_{\text{R-SPLIT}}(P)$ to be the partition returned by partition improvement using R-SPLIT for splitting, so we can state the following results describing Regress-action and Regress-policy.

Lemma 16.2: Given action α and value function V, Regress-action(V, α) on MDP M intersected with V.tree gives the partition computed by $I_{\text{R-SPLIT}}(V.\text{Tree})$ on MDP M_{π_α}.

Lemma 16.3: Given policy π and value function V, Regress-policy(V, π) on MDP M intersected with V.tree gives the partition computed by $I_{\text{R-SPLIT}}(V.\text{Tree})$ on MDP M_π intersected with π.Tree.

Given a policy π, structured successive approximation (SSA) repeatedly applies Regress-policy($\cdot$,π) starting from the reward partition, until a fixed point is reached. Noting that Regress-policy just computes $I_{\text{R-SPLIT}}$, SSA is shown to compute the same partition of the state space as partition iteration on the π-restricted MDP using R-SPLIT, starting from the π-induced partition of the state space.

Theorem 16: For any tree-represented MDP M and policy π, SSA(π) produces the same resulting partition as partition iteration on M_π using R-SPLIT starting from the partition π.Tree.

We note that it follows from Theorem 11 that the resulting partition is a bisimulation, so that traditional value computation methods can be used on the resulting aggregate model to compute a factored value function for M.

Policy iteration requires the computation of values to select the policy at each iteration—as a result, model reduction (which does not compute state values, but only aggregations) cannot be viewed alone as performing policy iteration. Here we analyze structured policy iteration as a combination of model reduction, traditional explicit-model techniques, and tree simplification.

Each iteration of structured policy iteration improves the policy π in two steps, analogous to explicit policy iteration: first, the policy π is evaluated using SSA, and then an improved policy is found relative to π using "structured policy improvement" (which is implemented by calls to $\cap$Trees and Simplify-tree in the pseudo-code). The first of these steps is equivalent to model reduction on M_π followed by traditional value iteration, as just discussed, yielding a factored value function for π.

Given this value function V_π, policy improvement is conducted as follows. The central "for" loop in the SPI pseudo-code intersects the partitions returned by Regress-action(V_π, α) for the different actions α. Noting we have shown that Regress-action computes the $I_{\text{R-SPLIT}}$ operation on the partition for V_π in M_{π_α}, we show here that this "for"

loop computes the $I_{\text{R-SPLIT}}$ operation for M itself. Once this operation is used to compute the partition, policy improvement concludes by doing a greedy look-ahead to compute the block labels (actions) and then simplifying the resulting tree.

Theorem 17: The policy improvement "for" loop in SPI computes $I_{\text{R-SPLIT}}(V_\pi.\text{Tree})$.

Therefore, each SPI iteration is equivalent to using model reduction and explicit value iteration to evaluate the policy π, and then partition improvement ($I_{\text{R-SPLIT}}$) followed by a greedy look-ahead and tree-simplification to compute a new policy π. We note that this is *not* the most natural way to use model reduction to perform policy iteration—that would be to reduce the entire model to a reduced model using R-SPLIT, and then conduct explicit policy iteration on the resulting reduced model. The trade-off between SPI and this more direct approach is discussed at the beginning of section 5: SPI benefits in many cases by doing value computations that allow tree simplification, but model reduction is useful in settings where the aggregation cannot depend on the model parameters but only the model structure (i.e. the parameters may change).

To conclude our discussion of structured stochastic dynamic programming, we turn to structured value iteration, or SVI. Perhaps, the most natural way to use model reduction to perform value iteration would be to compute a reduced model (say using S-SPLIT) and then perform explicit value iteration on that model. It turns out that SVI computes exactly this reduced model (while simultaneously performing value computations) if we omit the tree simplification step (Simplify-tree). This can be seen by noting that the "for" loop in SVI computes $I_{\text{R-SPLIT}}$, just as the "for" loop in SPI does—in this case, SVI iterates this computation starting from the reward function (using the "until" loop) until the reduced model is computed[30], after which SVI is just performing standard value iteration on that model. We conclude that, without tree simplification, SVI is essentially equivalent to model reduction using R-SPLIT followed by value iteration. Adding tree simplification to SVI has advantages and disadvantages similar to the tree simplification in SPI, as discussed above. If desired, SVI with tree simplification can be modeled using partition improvement with appropriate value function labeling alternated with tree simplification.

5.4 Partially Observable MDPs

The simplest way of using model-reduction techniques to solve partially observable MDPs (POMDPs) is to apply the model-minimization algorithm to the underlying fully observable MDP using an initial partition that distinguishes on the basis of both reward and observation model. The reduced model can then be solved using a standard POMDP algorithm [Monahan, 1982][Littman, 1994][Cassandra *et al.*, 1997][Zhang and Zhang, 2001]. We conjecture that the factored POMDP algorithm described in [Boutilier and Poole, 1996] can be analyzed using model reduction in a manner similar to the analysis of SVI presented above.

[30] We assume the "Similar(V, V')" test in SVI returns "false" if the corresponding partitions are different.

5.5 SPUDD

More recent work improving structured dynamic programming, e.g. SPUDD [Hoey *et al*. 1999], has primarily been concerned with changing the underlying representation from decision trees to decision diagrams. Since our algorithm is developed independently of the representation, model reduction is well defined for partitions represented as decision diagrams—no extension is needed. Rather than repeating all the analytic results shown above for structured dynamic programming again, for decision diagrams, we instead note that similar analytical results can be developed, comparing model minimization to SPUDD. We expect that empirical comparisons similar to those shown below can be obtained as well, but we do not yet have a decision diagram implementation.

6 Extensions and Related Work

6.1 Action Equivalence for Large Action Spaces

We have extended the notion of stochastic bisimilarity to include equivalence between actions that behave identically [Dean *et al.*, 1998]. Intuitively, two actions that have identical definitions can be collapsed into one. More than this though, once a state space equivalence relation has been selected, two actions that have different definitions may behave the same, once groups of equivalent states are aggregated. We wish to define the partition of the action space that results from a stochastic bisimulation using this intuition. Given an MDP $M = \langle Q, A, T, R\rangle$ and a relation $E \subseteq Q \times Q$, we say that two actions α_1 and α_2 are *dynamically bisimilar* with respect to E if for every two states $i, j \in Q$ we have that $T(i, \alpha_1, j/E) = T(i, \alpha_2, j/E)$. Given this equivalence relation on actions, we can then define a *dynamic quotient MDP* that aggregates both the state and action space. Given an MDP $M = \langle Q, A, T, R\rangle$ and a bisimulation $E \subseteq Q \times Q$, the dynamic quotient MDP $M/(E,D)$, where D is the dynamic bisimilarity relation with respect to E, is defined to be the machine $\langle Q/E, A/D, T', R'\rangle$ such that $T'(i/E, \alpha/D, j/E) = T(i, \alpha, j/E)$ and $R'(i/E) = R(i)$ where the choice of i and j does not affect T or R because E is a bisimulation, and the choice of α does not affect T by the definition of D.

One approach to computing a dynamic quotient MDP is to first compute a stochastic bisimulation and then compute the dynamic bisimilarity relation with respect to that bisimulation. However, this approach fails to exploit the possible reductions in the action space (by equivalence) during the construction of the stochastic bisimulation. Specifically, the iterative construction of the stochastic bisimilarity relation described in this paper requires, at each iteration, a computation for each action. If the action space can be grouped into exponentially fewer equivalence classes of actions, this "per action" computation can be replaced by a "per equivalence class of actions" computation, with possible exponential time savings. All of this assumes we can cheaply compute the dynamic bisimilarity relation D, which will depend entirely on the representation used for the MDP and the relation E. We do not consider this issue here, but in [Dean *et al.*, 1998] we present representations for MDPs that allow the effective computation of dynamic bisimilarity for many MDPs, and give an algorithm that exploits dynamic bisimilarity to

achieve possibly exponential savings in runtime (over that from model reduction alone).

6.2 Approximate Model Minimization

One of the foci of this paper has been to translate some of the efficiency of representing an MDP in a compact form into efficiency in computing an optimal policy for that MDP. The resulting computational savings can be explained in terms of finding a bisimulation over the state space, and using the corresponding partition to induce a smaller MDP that is equivalent to the original MDP in a well-defined sense. The reduced MDP states correspond to groups of states from the original MDP that behave the same under all policies, and thus the original and reduced MDP yield the same optimal policies and state values. Despite reducing the MDP with this approach, the resulting minimal model in many cases may still be exponentially larger than the original compact MDP representation—implying that in some cases the computational cost of solving the reduced MDP is still rather daunting.

One approach to overcoming this computational cost is to relax the definition of equivalence on states. This relaxation can be done by allowing the aggregation of states into the same "equivalence" class even though their transition probabilities to other blocks are different, so long as they are approximately the same (i.e., within ε of each other, for some parameter ε). We call the resulting partition an ε-stable partition—such a partition generally induces an aggregate MDP that is much smaller than the exact minimal model. Use of this approach does have its drawbacks: the reduced model is not equivalent to the original MDP, but only approximately equivalent. Solutions resulting from approximate model minimization thus may not be optimal but will typically be approximately optimal. For further information on how to carry out approximate model minimization/reduction see [Dean *et al*., 1997].

6.3 Handling Synchronic Effects

We first extend our representation of a factored MDP $M = \langle F, A, T_F, R_F \rangle$ given in Section 2.3 to represent synchronic effects (correlations between the effects of an action on different fluents). We change only the definition of T_F from our factored representation without synchronic effects. As before, the state space Q is given by the set of state fluents F. Following Bayesian belief network practice, the fluents F are now ordered as $f_1, \ldots, f_n$—the distribution describing the effects of an action on a fluent f_i will be allowed to depend on the post-action values of fluents f_j for j less than i, and the compactness of the resulting representation will in general depend heavily on the ordering chosen.

We assume that a "parent" relationship is defined for the fluents, as in a Bayesian network, such that for each fluent f_i, Parents(f_i, α) is a set of fluents earlier in the ordering $f_1, \ldots, f_n$ such that the value of f_i after taking action α is independent of the post-action value of any other fluent f_j, given post-action values for Parents(f_i, α). We then define the Ancestors(f_i, α) to give the set of fluents that are transitively parents of f_i for action α, along with f_i itself. The state-transition distribution of a factored MDP is now specified by giving a factored partition $T_F(\alpha, f_i)$ of Q for each fluent f_i and action α, where each parti-

tion block is labeled with a factored joint probability distribution over Ancestors(f_i, α) giving the probability that each assignment to Ancestors(f_i, α) will result when taking α from the labeled block. The distributions $T_F(\alpha, f_i)$ must obey a consistency constraint: for each action α and fluents f and f' such that $f' \in$ Parents(f, α), the distribution $T_F(\alpha, f')$ must be the same as the distribution $T_F(\alpha, f)$ marginalized to the fluents Ancestors(f'). One way to achieve this consistency is to represent each factored conditional probability distribution as a product (as in a Bayesian network), such that the distribution for a fluent includes every factor used in the product of any of that fluent's parents[31] (i.e., the Bayesian network for fluent f contains the Bayesian networks for the parents of f).

Given this representation for a synchronous-effect factored MDP, model reduction using S-SPLIT, F-SPLIT, or FS-SPLIT can be carried out just as specified above. This is because these split methods do not depend on the partition labels in the action descriptions, but only on the partitions themselves. Exact splitting with SPLIT requires using the joint probability distribution labels to combine blocks that are "coincidentally" alike after S-SPLIT. This combination is similar in spirit to that described for the independent action effects case near the end of section 4.4, and we leave this generalization as an exercise for the reader. Model reduction using R-SPLIT requires adding only an inference algorithm for determining whether a joint probability distribution assigns a probability of zero to a given formula—for the key case of distributions in chain-rule product form (i.e. like Bayesian networks) and formulas that are conjunctions of literals, such algorithms are generally well known (e.g., [Pearl, 1988]).

6.4 Other Related Work

The basic idea of computing reduced equivalent models has its origins in automata theory [Hartmanis and Stearns, 1966] and stochastic processes [Kemeny and Snell, 1960]. Our work can also be viewed as a stochastic generalization of recent work in computer-aided verification via model checking [Burch *et al.*, 1994][Lee and Yannakakis, 1992]. In addition, the goals of our work are similar to goals of [Dietterich and Flann, 1995], which presents an online learning method using a factored representation to learn about blocks of states, using a regression operator similar to our block splitting operation.

The approximation of an optimal policy discussed in the last section is just one of many approximation approaches. [Boutilier and Dearden, 1996] gives approximate versions of SPI and SVI by sometimes allowing states with similar, but different, values to be aggregated into the same leaf of a value-function tree. This additional aggregation is achieved by pruning value trees, replacing sub-trees whose values differ by at most ε by leaves whose label may be either an average value for the sub-tree or a range of values

[31] Conversion to this factored MDP representation from the more familiar (and very similar) dynamic Bayesian networks with synchronous effects is straightforward [Littman, 1997], but may involve an exponential growth in size in computing the required state-space partitions. It is possible to design a similar labeled-partition representation that avoids this growth, but applying model minimization appears to require the exponentially larger representation. The synchronous-effect methods presented in [Boutilier *et al.*, 2000] also encounter exponential size growth when "summing out post-action influences."

subsuming all the values of the sub-tree. [Koller and Parr, 2000] propose a very different factored value function representation—value functions are represented as a linear combination of the factored value functions used here, and a policy iteration method is given for this decomposed value function method. Note that this representation can assign exponentially many different values over the state space with a polynomial-size decomposition, unlike our labeled factored partitions or the more familiar decision-tree representations for value functions. Large state spaces have also been dealt with approximately by trajectory sampling in [Kearns *et al*., 1999], and elsewhere.

7 Empirical Investigation

We have explored the theory of model minimization; here we provide some data on its performance on simple synthetic domains. We have constructed a non-optimized implementation using DNF formulas to represent blocks—using S-SPLIT to construct a reduced, equivalent model. We used this implementation to conduct experiments on the Linear, Expon, and Coffee domains used in the previous evaluation of structured dynamic programming [Boutilier *et al*., 2000], and compare the reduced-model sizes found by our technique to the size of the value-function representation produced by structured dynamic programming (SVI, in particular). We use the number of leaves in the decision-tree value function produced by SVI as a measure of the size of the representation.

We now briefly describe these domains.[32] The Linear domains, Linear3 through Linear9, have between three and nine ordered state fluents, respectively. For each state fluent, the domain provides an action that sets that fluent to "true" while setting all fluents later in the order to "false". Reward is obtained only when all state fluents are "true". The Linear domains were designed to show the strengths of the structured–dynamic-programming algorithms—due to our similarity to these approaches, we expected to see good results on these domains. The Expon domains, Expon3 through Expon9, are similar to the Linear domains, except that the action corresponding to each state fluent sets that fluent to "true", if all later fluents are "true", and sets it "false" otherwise. (In either case, as before, it sets all later fluents to "false".) To reach reward, these actions must be used to "count" through the binary representations of the states, so we expect every state to behave uniquely. The Expon domains were designed to exploit the weaknesses of structured dynamic programming, so, we expected little reduction in state space size.

The Coffee domain has six state fluents (has-user-coffee, has-robot-coffee, wet, raining, have-umbrella, and location) and four actions (move, give-coffee, buy-coffee, and get-umbrella). The move action has a 0.9 chance of moving the robot between the of-

[32] Complete details of the Coffee domain and some Linear and Expon domains can be found online at http://www.cs.ubc.ca/spider/jhoey/spudd/spudd.html. Note that this website also contains some "factory" domains. We do not include these domains in our tests because both our model reduction implementation (using S-SPLIT) and the web-available SPUDD implementation are unable to solve them exactly in the available memory. Approximation methods could be added to either approach in order to handle the factory domains, however these approximation techniques will not be discussed further here. (We note that SPUDD has such approximation built in, and can analyze the factory domains using it.)

Domain	#Of Fluents	State Space Size	# Of SVI Leaves	Minimal Model Size	Ratio
Linear3	3	8	4	4	0.500
Linear4	4	16	5	5	0.313
Linear5	5	32	6	6	0.188
Linear6	6	64	7	7	0.110
Linear7	7	128	8	8	0.063
Linear8	8	256	9	9	0.036
Linear9	9	512	10	10	0.020
Expon3	3	8	8	8	1.000
Expon4	4	16	16	16	1.000
Expon5	5	32	32	32	1.000
Expon6	6	64	64	64	1.000
Expon7	7	128	128	128	1.000
Expon8	8	256	256	256	1.000
Expon9	9	512	512	512	1.000
Coffee	6	64	18	21	0.329

Figure 12. Results from the experiments. For each domain, we give the name, number of fluents defining the state space, number of states in state space, number of leaves in the value tree after running SVI, number of blocks in the reduced model, and state-space compression ratio from aggregation using this reduced model, respectively.

fice and store locations, with an 0.9 chance of getting the robot wet if it is raining, unless the robot has the umbrella, which reduces that chance to 0.1. If the robot is in the office with coffee, the give-coffee action has a 0.8 chance of giving the user coffee and an (independent) 0.9 chance of the robot losing the coffee. If the robot is at the store, give-coffee has a 0.8 chance of the robot losing the coffee, with no chance of providing any to the user. Buy-coffee has a 0.9 chance of getting the robot coffee, if the robot is in the store. Get-umbrella has a 0.9 chance of getting the robot the umbrella, when in the office. There is a large reward if the user has coffee and a small one if the robot is not wet.

These domains are, of course, much smaller than what will typically be seen in real applications, but they illustrate the range of possible results from our technique, and allow for a comparison to other current approaches, in particular to structured dynamic programming. The results obtained in our experiments are shown in Figure 12, and are as expected—the Linear domains show a linear increase in the size of the reduced model with respect the number of variables (i.e., an exponential amount of compression), whereas the Expon domains show no reduction in model size, and remain exponential in the number of variables. Structured dynamic programming, specifically SVI, performs identically (on both Linear and Expon domains), showing that we are indeed factoring that method into a model-reduction phase, followed by any traditional solution technique.

The Coffee domain shows a substantial savings, with the reduced MDP being about a third the size of the original, and very similar in size to the SVI-produced value function, but not identical. The difference in the Coffee domain results from model-

reduction refusing to aggregate states when they differ in the dynamics of *any* action, even a non-optimal action. In this case, when the user has coffee and the robot is dry, the robot need only avoid going outside to stay dry, so that no other state variables affect the value (just has-user-coffee and wet). However, sub-optimal actions need to know more of the state to determine the chance that the robot gets wet, e.g., whether it is raining—this results in four states in the reduced model that correspond to one SVI value-function leaf.

Overall, these results are comparable to those obtained by structured dynamic programming, which is expected since those algorithms can be viewed as a form of model reduction. Further investigation into the use of model minimization and comparable techniques in real applications is needed in order to verify what exactly are the drawbacks of such approaches when applied in practice.

8 Conclusion

We present the method of model minimization for MDPs and its use in analyzing and understanding existing algorithms. In order to develop this method of analysis we have shown that equivalence notions used in concurrent process theory to compare processes for equivalence have a direct application to the theory of MDPs. In particular, the notion of a bisimulation between two processes (formalized above in a limited way as FSMs) directly generalizes to a useful equivalence notion for MDP states. Moreover, concurrent process theory provides theoretical tools that can be used to automatically compute bisimulations between FSMs—these tools also immediately generalize to compute MDP state equivalence. We also develop methods to carry out this computation for MDPs represented in factored form. By adding a straightforward notion of action equivalence relative to a bisimulation, we can also use the notion of bisimulation to aggregate a large action space. These methods also lend themselves naturally to approximation, as we have discussed elsewhere in [Dean *et al*., 1997].

9 References

[Baum and Nicholson, 1998] Baum, J. and Nicholson, A. E. 1998. Dynamic non-uniform abstractions for approximate planning in large structured stochastic domains. In *Pacific Rim International Conference on Artificial Intelligence 1998*. 570-598.

[Bellman, 1957] Bellman, R. 1957. *Dynamic Programming*. Princeton University Press, Princeton, NJ.

[Bouajjani *et al.*, 1992] Bouajjani, A.; Fernandez, J.-C.; Halbwachs, N.; Raymond, P.; and Ratel, C. 1992. Minimal state graph generation. *Science of Computer Programming* 18:247-269.

[Boutilier and Poole, 1996] Boutilier, C. and Poole, D. 1996. Computing optimal policies for partially observable decision processes using compact representations. In *Proceedings of the Third European Workshop on Planning*. 1168-1175.

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

[Boutilier and Dearden, 1996] Boutilier, Craig and Dearden, Richard 1996. Approximat-

ing value trees in structured dynamic programming. In *Proceedings of the Thirteenth International Conference on Machine Learning*. 54-62. Bari, Italy.

[Boutilier *et al.*, 1995b] Boutilier, Craig; Dearden, Richard; and Goldszmidt, Moises 1995b. Exploiting structure in policy construction. In *Proceedings International Joint Conference on Artificial Intelligence 14.* IJCAII. 1104-1111.

[Boutilier *et al*., 1999] Boutilier, Craig; Dean, Thomas; and Hanks, Steve 1999. Decision theoretic planning: Structural assumptions and computation leverage. *Journal of Artificial Intelligence Research* 11:1-94.

[Boutilier *et al*., 2000] Boutilier, Craig; Dearden, Richard; and Goldszmidt, Moisés 2000. Stochastic dynamic programming with factored representations. *Artificial Intelligence* 121(1-2): 49-107.

[Burch *et al.*, 1994] Burch, Jerry; Clarke, Edmond M.; Long, David; McMillan, Kenneth L.; and Dill, David L. 1994. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer Aided Design* 13(4):401-424.

[Cassandra *et al.*, 1997] Cassandra, Anthony; Littman, Michael L. and Zhang, Nevin L. 1997. Incremental pruning: A simple, fast, exact algorithm for partially observable Markov Decision Processes. In *13th Conference on Uncertainty in Artificial Intelligence*. 54-61.

[Dean and Givan, 1997] Dean, Thomas and Givan, Robert 1997. Model minimization in Markov decision processes. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence.* AAAI. 106-111.

[Dean and Kanazawa, 1989] Dean, T. and Kanazawa, K. 1989. A model for reasoning about persistence and causation. *Computer Intelligence* 5(3):142-150.

[Dean *et al*., 1997] Dean, Thomas ; Givan, Robert ; and Leach, Sonia. Model reduction techniques for computing approximately optimal solutions for Markov decision processes. In *13th Conference on Uncertainty in Artificial Intelligence*. 106-111.

[Dean *et al.*, 1998] Dean, Thomas; Givan, Robert; and Kim, Kee-Eung 1998. Solving planning problems with large state and action spaces. In *The Fourth International Conference on Artificial Intelligence Planning Systems.* 102-110.

[Dearden and Boutilier, 1997] Dearden, Richard and Boutilier, Craig 1997. Abstraction and approximate decision theroretic planning. *Artificial Intelligence* 89(1):219-283.

[Dietterich and Flann, 1995] Dietterich, T. G., and Flann, N. S. 1995. Explanation-based learning and reinforcement learning: A unified view. In *Proceedings Twelfth International Conference on Machine Learning*, 176-184.

[Draper *et al.*, 1994] Draper, Denise; Hanks, Steve; and Weld Daniel S. 1994. Probabilistic planning with information gathering and contingent execution. In *Artificial Intelligence Planning Systems 1994*. 31-36.

[Goldsmith and Sloan, 2000] Goldsmith, Judy and Sloan, Robert 2000. The complexity of model aggregation. In *Proc. AI and Planning Systems*, April, 2000. 122-129.

[Hanks, 1990] Hanks, Steve 1990. Projecting plans for uncertain worlds. Ph.D. thesis 756, Yale University, Department of Computer Science, New Haven, CT.

[Hanks and McDermott, 1994] Hanks, S and McDermott, D. V. 1994. Modeling a dynamic and uncertain world I: Symbolic and probabilistic reasoning about change. *Artificial Intelligence*, 66(1), 1-55.

[Hartmanis and Stearns, 1966] Hartmanis, J. and Stearns, R. E. 1966. *Algebraic Structure Theory of Sequential Machines.* Prentice-Hall, Englewood Cliffs, N.J.

[Hennessy and Milner, 1985] Hennessy, M. and Milner, R. 1985. Algebraic laws for nondeterminism and concurrency. In *Journal of the Association for Computing Machinery 137.*

[Hoey *et al*. 1999] Hoey, Jesse; Aubin, Robert St.; Hu, Alan J.; and Boutilier, Craig 1999. SPUDD: Stochastic Planning using Decision Diagrams, In *15th Conference on Uncertainty in Artificial Intelligence. 279--28*.

[Howard, 1960] Howard, R. A. 1960. *Dynamic Programming and Markov Processes*. Cambridge, Mass. The MIT Press.

[Kearns *et al*., 1999] Kearns, Michael; Mansour, Yishay; and Ng, Andrew 1999. A sparse sampling algorithm for near-optimal planning in large Markov decision processes. In *16th Conference on Uncertainty in Artificial Intelligence*. 21-30.

[Kemeny and Snell, 1960] Kemeny, J. G. and Snell, J. L. 1960. *Finite Markov Chains.* D. Van Nostrand, New York.

[Koller and Parr, 2000] Koller, Daphne and Ron, Parr 2000. Policy iteration for factored MDPs. In *16th Conference on Uncertainty in Artificial Intelligence*. 326--334.

[Kushmerick *et al*., 1995] Kushmerick, Nicholas; Hanks, Steve; and Weld, Daniel 1995. An algorithm for probabilistic planning. *Artificial Intelligence* 76(1-2).

[Larson and Skou, 1991] Larsen, K. G. and Skou, A. 1991. Bisimulation through probabilistic testing. In *Information and Computation 94(1):128*.

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

[Lin and Dean, 1995] Lin, Shieu-Hong and Dean, Thomas 1995. Generating optimal policies for high-level plans with conditional branches and loops. In *Proceedings of the Third European Workshop on Planning.* 205-218.

[Littman, 1994] Littman, Michael 1994. The Witness algorithm: Solving partially observable Markov descision processes. Brown University Department of Computer Science Technical Report CS-94-40.

[Littman *et al*., 1995] Littman, Michael; Dean, Thomas; and Kaelbling, Leslie 1995. On the complexity of solving Markov decision problems. In *Eleventh Conference on Uncertainty in Artificial Intelligence*. 394-402.

[Littman, 1997] Littman, Michael 1997. Probabilistic propositional planning: Reoresentations and complexity. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*. 748-754.

[McAllester and Rosenblitt, 1991] McAllester, David and Rosenblitt, David 1991. Systematic nonlinear planning. In In *Proceedings of the Eighth National Conference on Artificial Intelligence*. 634-639.

[Milner, 1989] Milner, R. 1989. Communication and Concurrency. Series in *Computer Science*. Prentice-Hall International.

[Milner, 1990] Milner, R. 1990. Operational and Algebraic Semantics of Concurrent Processes. In Leeuwen J. van (Ed.), *Handbook on Theoretical Computer Science* 1201-1242. Amsterdam: Elsevier Science Publishers B. V.

[Monahan, 1982] Monahan, G. E. 1982. A survey of partially observable Markov decision processes: Theory, models, and algorithms. *Management Science* 28(1):1-16.

[Park, 1981] Park, D.M. 1981. Concurrency on automata and infinite sequences. In P. Deussen, ed. *Conference on Theoretical Computer Science*, volume 104 of *Lecture Notes in Computer Science.* Springer Verlag.

[Pearl, 1988] Pearl, J. 1988. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, San Francisco, California.

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

[Zhang and Zhang, 2001] Zhang, Nevin L. and Zhang, Weihong 2001. Speeding up the convergence of value iteration in partially observable Markov decision processes. In *Journal of Artificial Intelligence Research* . Vol. 14.

Appendix

Lemma 4.1: The reflexive symmetric transitive closure of a stochastic bisimulation between any two MDPs $M = \langle Q, A, T, R\rangle$ and $M' = \langle Q', A, T', R'\rangle$ restricted to $Q \times Q$ is itself a stochastic bisimulation between M and itself.

Proof: Let E_1 be a stochastic bisimulation between the MDPs $M = \langle Q, A, T, R\rangle$ and $M' = \langle Q', A, T', R'\rangle$ and let E_2 be the reflexive symmetric transitive closure of E_1 restricted to $Q \times Q$. We show that E_2 is a stochastic bisimulation.

Consider i and j in Q such that $E_2(i, j)$. We note that the definition of E_2 as the reflexive symmetric transitive closure of E_1 ensures that there is a (possibly empty) path of arcs in E_1, ignoring arc direction, from i to j. Likewise there must be a path of arcs in E_1 between any two states in i/E_2 or any two states in j/E_2. A simple induction on the length of an arbitrary path of E_1 arcs shows that any two states related by such a path have the same R (or R') values, because $E_1(i', j')$ implies $R(i') = R'(j')$. It follows that $R(i/E_2)$ and $R(j/E_2)$ are well defined and equal, as desired in showing that E_2 is a bisimulation.

To show the transition-model properties that imply that E_2 is a bisimulation, we first note that the sets i/E_1 and i/E_2 (and likewise j/E_1 and j/E_2) are identical by definition. We must show that for any $i' \in Q$ and $j' \in Q$ such that $E_2(i', j')$, the block transition probabilities $T(i, \alpha, i'/E_2)$ and $T(j, \alpha, j'/E_2)$ are equal. As just observed, it suffices to show that $T(i, \alpha, i'/E_1)$ and $T(j, \alpha, j'/E_1)$ are equal. This follows by induction on the sum of the length of the shortest E_1 path from i' to j' and the length of the shortest E_1 path from i to j (ignoring arc direction)—this induction iterates the fact that for any action α, any $x, x' \in Q$ and $y, y' \in Q'$, $E_1(x, y)$ and $E_1(x', y')$ together imply that $T(x, \alpha, x'/E_1)$ and $T'(y, \alpha, y'/E_1)$ are equal, since E_1 is a bisimulation.■

Lemma 4.2: The union of two stochastic bisimulations between the same pair of MDPs is also a stochastic bisimulation between those MDPs.

Proof: Let E_1 and E_2 be two stochastic bisimulations between the same pair of MDPs $M = \langle Q, A, T, R\rangle$ and $M' = \langle Q', A, T', R'\rangle$ and let E be the union of those stochastic bisimulations (i.e. the union of the sets of pairs of states related by those bisimulations). We now show that E is a stochastic bisimulation. We write E^* for the reflexive symmetric transitive closure of E. Consider $i \in Q$ and $j \in Q'$ such that $E(i, j)$.

That $R(i/E)$ and $R'(j/E)$ are well defined and equal to each other is implied by the following assertion. For any i' and j' in $Q \cup Q'$ such that $E^*(i', j')$, $R''(i') = R''(j')$,

where R'' is defined on $Q \cup Q'$ by R on Q and by R' on Q'. This assertion is shown by induction on the length of the B-path between i' and j', iterating the fact that either $E_1(i'', j'')$ or $E_2(i'', j'')$ implies $R(i'') = R'(j'')$ because E_1 and E_2 are both bisimulations.

We now argue that $E(i, j)$ and $E(i', j')$ together imply that $T(i, \alpha, i'/E) = T'(j, \alpha, j'/E)$, for $i, i' \in Q$ and $j, j' \in Q'$. Without loss of generality, by symmetry, we assume that $E_1(i, j)$. It is easy to show that the equivalence classes of E^* are formed by unioning sets of the form k/E_1 for different k in $Q \cup Q'$. Thus the class i'/E is the disjoint union of sets $i'_1/E_1, \ldots, i'_{n_1}/E_1$, and likewise j'/E is the disjoint union of sets $j'_1/E_1, \ldots, j'_{n_2}/E_1$. We now show how to select, for each block i'_m/E_1, a corresponding block $j'_{m'}/E_1$ such that $T(i, \alpha, i'_m/E_1) = T'(j, \alpha, j'_{m'}/E_1)$, with no block $j'_{m'}/E_1$ being selected twice; from this we can conclude that $T(i, \alpha, i'/E) \pi T'(j, \alpha, j'/E)$. A symmetric argument then shows $T(i, \alpha, i'/E) \pi T'(j, \alpha, j'/E)$, so we can conclude $T(i, \alpha, i'/E) = T'(j, \alpha, j'/E)$ as desired. The block $j'_{m'}/E_1$ can be selected by finding any state j'' such that $E_1(i'_m, j'')$; it is not hard to then show that the block j''/E must be $j'_{m'}/E_1$ for some m' but will not be selected for any other m.■

Theorem 4: Stochastic bisimilarity restricted to the states of a single MDP is an equivalence relation, and is itself a stochastic bisimulation from that MDP to itself.

Proof: First, we prove that there exists a maximal stochastic bisimulation from an MDP M to itself—it follows that this relation is stochastic bisimilarity, which is thus a bisimulation. Since there are only finitely many unique binary relations that can be defined over the states of an MDP, we can enumerate those that are stochastic bisimulations on M as $B_1, \ldots, B_m$. We construct the maximal stochastic bisimulation in the following manner, starting with $E_1 = B_1$, and taking $E_i = E_{i-1} \cup B_i$, this leads us to E_m which is the maximal stochastic bisimulation. In order to prove this, we need to show that E_m contains all other stochastic bisimulations, and that it is itself a stochastic bisimulation. E_m contains all other stochastic bisimulations, since it contains all the B_i by its construction. We show that E_m is a stochastic bisimulation by induction on the index. As a base case, E_1 is a stochastic bisimulation, since it is B_1, which is a stochastic bisimulation. For the inductive case, the union $E_{i-1} \cup B_i$ yields E_i, which is a stochastic bisimulation by Lemma 4.2.

All that remains to prove the theorem is to show that E_m when restricted to the states of a single MDP is an equivalence relation—this follows immediately from Lemma 4.1 because if the reflexive symmetric transitive closure of E_m is a bisimulation it must be contained in E_m and thus must be E_m. ■

Theorem 5: Any stochastic bisimulation that is an equivalence relation is a refinement of both optimal value equivalence and action sequence equivalence.

Proof: Throughout this proof we will use states i and j as stochastically bisimilar states from an MDP $M = \langle Q, A, T, R \rangle$. We show optimal value equivalence of i and j by showing, using induction on m, that i and j have the same optimal discounted value at every finite horizon m. We define m-horizon optimal discounted value function in the following manner for all states s and all non-negative integers m,

$$v_m(s) = R(s) + \max_\alpha \gamma \Sigma_{k \in Q} [T(s, \alpha, k)\, v_{m-1}(k)]$$

where γ is the discount factor, and we take $v_{-1}(s)$ to be 0 for all states s.

For the base case take $m = 0$. In this case the value function for any state is just the reward for that state, $v_0(s) = R(s)$. Since states i and j are stochastically bisimilar we know that $R(i) = R(j)$, and so that $v_0(i) = v_0(j)$, as desired. For the inductive case, we define the m-horizon Q-value[33] for any state s, action α, and non-negative integer m, by

$$q_m(s, \alpha) = R(s) + \gamma \Sigma_{k \in Q} [T(s, \alpha, k)\, v_{m-1}(k)].$$

Let E be stochastic bisimilarity. Using the induction hypothesis, we have for any action α,

$$\begin{aligned} q_m(i, \alpha) &= R(i) + \gamma \Sigma_{k \in Q} [T(i, \alpha, k)\, v_{m-1}(k)] \\ &= R(i) + \gamma \Sigma_{b \in Q/E} [T(i, \alpha, b)\, v_{m-1}(b)] \\ &= R(j) + \gamma \Sigma_{b' \in Q/E} [T(j, \alpha, b')\, v_{m-1}(b')] \\ &= R(j) + \gamma \Sigma_{k \in Q} [T(j, \alpha, k)\, v_{m-1}(k)] = q_m(j, \alpha). \end{aligned}$$

Since for any state s, $v_m(s) = \max_\alpha q_m(s, \alpha)$, it follows that $v_m(i) = v_m(j)$, as desired.

We now show that i and j are action-sequence equivalent by induction on the length m of the action sequence—we show that for any action sequence $\alpha_1, \ldots, \alpha_m$, the distribution over sequences of rewards attained by following $\alpha_1, \ldots, \alpha_m$ is the same for i and j. We take $\phi_{s,m}(\vec{\alpha})$ to be a random variable ranging over reward sequences of length m, with the distribution generated from starting state s following action sequence $\vec{\alpha}$.

For the base case, we take $m = 0$ and consider the empty sequence ε of actions. Here, the reward sequence $\phi_{s,0}(\varepsilon)$ is deterministically the empty sequence—implying that $\phi_{i,0}(\varepsilon)$ and $\phi_{j,0}(\varepsilon)$ have identical distributions as desired.

For the inductive case, consider action sequence $\vec{\alpha} = \alpha_1, \ldots, \alpha_m$. We note that for any state s, we have that $\Pr(\phi_{s,m}(\vec{\alpha}) = r_1, \ldots, r_m)$ is equal to

$$\Pr(\phi_{s,1}(\alpha_1) = r_1) \Sigma_{b \in Q/E}\ T(s, \alpha_1, b)\ \Pr(\phi_{b,m-1}(\alpha_2, \ldots, \alpha_m) = r_2, \ldots, r_m),$$

where $\phi_{b,n}(\vec{\alpha})$ is defined to be $\phi_{s,n}(\vec{\alpha})$ for some state $s \in b$, and the choice of s does not affect the value of $\phi_{b,n}(\vec{\alpha})$ for $n < m$ by the induction hypothesis. We apply this equation for s equal to i and for s equal to j, and show that the right-hand sides are equal in the two cases. First, we note that the probability that $\phi_{s,1}(\alpha_1)$ equals r_1 in the above equation is either zero or one, depending on whether $R(s)$ is r_1, and that $R(i) = R(j)$ since i and j are stochastically bisimilar. Then, the fact that $T(i, \alpha_1, b) = T(j, \alpha_1, b)$ for each block b (because E is a bisimulation) gives $\Pr(\phi_{i,m}(\vec{\alpha}) = r_1, \ldots, r_m) = \Pr(\phi_{j,m}(\vec{\alpha}) = r_1, \ldots, r_m)$, concluding the inductive case. Thus, stochastic bisimilarity refines action-sequence equivalence.■

Theorem 6: The reflexive, symmetric, transitive closure of any stochastic bisimulation from MDP $M = \langle Q, A, T, R \rangle$ to any MDP, restricted to $Q \times Q$, is an equivalence relation $E \subseteq Q \times Q$ that is a stochastic bisimulation from M to M.

Proof: This follows directly from Lemma 4.1, along with the fact that restricting an

[33] Our use of the standard terminology "Q-function" does not imply any connection to the state space Q.

equivalence relation to a sub-domain preserves the equivalence relation property.■

Theorem 7: Given an MDP $M = \langle Q, A, T, R\rangle$ and an equivalence relation $E \subseteq Q \times Q$ that is a stochastic bisimulation, each state i in Q is stochastically bisimilar to the state i/E in M/E. Moreover, any optimal policy of M/E induces an optimal policy in the original MDP.

Proof: Let $M/E = \langle Q/E, A, T', R'\rangle$. First we prove that any $i \in Q$ is stochastically bisimilar to i/E in M/E. Let Z be the relation over $Q \times Q/E$ that contains only the pairs $(i, i/E)$ for each $i \in Q$. We show Z is a stochastic bisimulation from M to M/E.

Select $i \in Q$ and $j \in Q/E$ such that $Z(i, j)$. We note that i/Z equals both i/E and j, and that j/Z is the set $\{j\}$. It follows that the rewards $R(i/Z)$ and $R'(j/Z)$ are well defined and equal, since E is a stochastic bisimulation. Now select action $\alpha \in A$, state $i' \in Q$ and state $j' \in Q/E$ such that $Z(i', j')$. Noting that $T(i, \alpha, i'/Z) = T(i, \alpha, i'/E) = T'(i/E, \alpha, i'/E) = T'(j, \alpha, j') = T'(j, \alpha, j'/Z)$, we conclude Z is a bisimulation and therefore that i and i/E are stochastically bisimilar.

As above in the proof of Theorem 5, define the Q-value for any state s and action α by giving $q(s, \alpha)$ as the sum of $R(s)$ and $\gamma \Sigma_{k \in Q} [T(s, \alpha, k)\, v^*(k)]$. We now show that any optimal action for state i/E in Q/E is an optimal action for state i in Q. To show this, we show that the Q-value for arbitrary action α in state i/E is the same as the Q-value for α in state i. We conclude

$$\begin{aligned} q(i/E, \alpha) &= R'(i/E) + \gamma \Sigma_{j/E \in Q/E}\, T'(i/E, \alpha, j/E)\, v^*(j/E) \\ &= R(i) + \gamma \Sigma_{j/E \in Q/E}\, T(i, \alpha, j/E)\, v^*(j/E) \\ &= R(i) + \gamma \Sigma_{s \in Q} T(i, \alpha, s)\, v^*(s) = q(i, \alpha) \text{ in } M. \end{aligned}$$

The second line follows via the definition of M/E, with for R' and T', and the third line via the definition of block transition probability and the equality of values within a block (implicit in the proof of Theorem 5). This Q-value equivalence yields our theorem. ■

Lemma 8.1: Given equivalence relation E on Q and states p and q such that $T(p, \alpha, C) \neq T(q, \alpha, C)$ for some action α and block C of E, p and q are not related by any stochastic bisimulation refining E.

Proof: Suppose not. Let B and C denote blocks of the partition of Q induced by E, let α be any action in A, and let p and q denote states in block B such that $T(p, \alpha, C) \neq T(q, \alpha, C)$. Let E' be a stochastic bisimulation refining E such that p and q are in the same block in E'. Let $\{C_1, \ldots, C_k\}$ be the set of blocks in E' that refine C. Because E' is a stochastic bisimulation, for each block D in E', $T(p, \alpha, D) = T(q, \alpha, D)$. Summing this fact over all the blocks C_i we derive the following equation, contradicting $T(p, \alpha, C) \neq T(q, \alpha, C)$:

$$T(p, \alpha, C) = \Sigma_{1 \leq i \leq k} T(p, \alpha, C_i) = \Sigma_{1 \leq i \leq k} T(q, \alpha, C_i) = T(q, \alpha, C).\blacksquare$$

Corollary 8.2: Let E be an equivalence relation on Q, B a block in E, and C a union of blocks from E. Every bisimulation on Q that refines E is a refinement of the partition

SPLIT(B, C, E).

Proof: Let E be an equivalence relation on Q, B a block in E, and C be the union of blocks C_1 thru C_n from E. Let E' be a stochastic bisimulation that refines E. Note that SPLIT(B, C, E) will only split states i and j if either $R(i) \neq R(j)$ or $T(i, \alpha, C) \neq T(j, \alpha, C)$, by definition. But if $R(i) \neq R(j)$ then $i / E' \neq j / E'$ since E' is a stochastic bisimulation. And if $T(i, \alpha, C) \neq T(j, \alpha, C)$, then there must be some k such that $T(i, \alpha, C_k) \neq T(j, \alpha, C_k)$, because for any state s, $T(s, \alpha, C) = \Sigma_{1 \leq m \leq n} T(s, \alpha, C_m)$. Therefore, we can conclude by Lemma 8.1 that $i / E' \neq j / E'$. ■

Theorem 8: Partition iteration and model minimization both compute stochastic bisimilarity.

Proof: Partition iteration and model minimization both terminate with a partition P for which SPLIT(B, C, P) = P for any blocks B and C in P. SPLIT(B, C, P) will split any block B containing a pair of states i and j for which either $R(i) \neq R(j)$ or $T(i, \alpha, C) \neq T(j, \alpha, C)$. So any partition returned by partition iteration or model minimization must be a stochastic bisimulation. Since both model minimization and partition iteration start from the trivial $\{Q\}$ partition, and each only refines the partition by applying the SPLIT operator to blocks in the partition, we can conclude by Corollary 8.2 that each partition encountered, including the resulting partition, must contain stochastic bisimilarity. The resulting partition, being a stochastic bisimulation, must be stochastic bisimilarity. ■

Theorem 9: Model reduction returns a stochastic bisimulation.

Proof: Since model reduction always splits at least as much as model minimization, due to the definition of stability, it must be the case that the partition returned by model reduction is a refinement of the partition returned by the model minimization algorithm, i.e., stochastic bisimilarity according to Theorem 8. Any such relation has the reward properties required of stochastic bisimulations. The transition-model properties follow immediately from the stability of all blocks in the resulting partition, which is a consequence of the exit test of the final "while" loop in the algorithm. ■

Corollary 9.1: The optimal policy for the quotient model produced by model reduction induces an optimal policy for the original MDP.

Proof: Follows directly from Theorem 7 and Theorem 9.■

Theorem 10: The bounded-size model-minimization problem is NP-hard.

Proof: We proceed by reducing 3CNF satisfiability[34] to the bounded-size model minimization problem. Let F be a formula in 3CNF involving n variables X_1, …, X_n and m clauses, with $L_{j,i}$ denoting the jth literal in the ith clause, for $1 \leq j \leq 3$ and $1 \leq i \leq m$; every literal is of the form X_p or $\neg X_p$ for some $1 \leq p \leq n$.

[34] The 3CNF formula is a non-empty set of clauses, each a disjunction of exactly three literals.

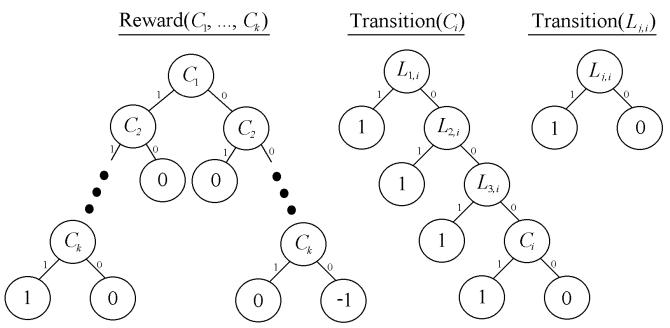

Figure 13. Reward and transition functions for 3CNF reduction. Each is a real-valued labeled partition, as described in Section 2.3.2, representing a factored MDP as discussed in Section 2.3.3.

We construct a factored MDP M for use in minimization as follows. The set of fluents factoring the state-space is the union of $\{X_p \mid 1 \le p \le n\}$ and $\{C_i \mid 1 \le i \le m\}$ where the X_p are called *variable fluents*, and the C_i are called *clause fluents* and will be associated by construction with the clauses in F. Below we often refer to the n X_p fluents (and the corresponding n variables in F) indirectly by referring to the $3m$ literals $L_{j,i}$. We now describe the reward and state-transition functions, which are shown in Figure 13. There is only one action in M. The single action is only capable of changing the truth value of the C_i fluents—fluent C_i is set to be true if one of the $L_{j,i}$ is true, otherwise C_i retains its previous value. So, after its first application, the action is a no-op, since it deterministically sets the C_i values according to the $L_{j,i}$ values, which do not change. There are three possible rewards, 1, -1, and 0, which are associated, respectively, with the block of states where all of the C_i are true, the block where all the C_i are false, and the remaining block.

Each state in the MDP M specifies values for all the X_p and C_i variables. As a result, each state can be viewed as specifying a truth-assignment to the $L_{j,i}$ variables, i.e., a potential model for the formula F given as input to the satisfiability problem. Each state, also specifies values for the C_i variables. It is important to note that there is one state in the state space for each way of setting all the X_p and C_i variables. Suppose the formula F is satisfiable. Consider a state setting all C_i variables false, and setting the X_p variables according to a satisfying assignment for F. Observe that there will be an action transition in the MDP from this state to a state where all the C_i variables are true. If the formula F is not satisfiable, then there will be no state where such a transition is possible. We now analyze the minimal model of the MDP M, and leverage these observations to determine the satisfiability of F from only the number of blocks in the minimal model—specifically, from whether or not there is a block where all C_i variables are false from which there is a transition possible to a block where all C_i variables are true.

Figure 15 shows several formulas that will be useful in describing the minimal model for M. Using these formulas, the reward function can be described by labeling the partition $\{C, U, \neg C \wedge \neg U\}$—this partition is the result of $I(\{Q\})$, and is shown in square boxes in Figure 14. Model minimization will start with and further refine this partition, as discussed below. The formula F is satisfiable if and only if there is a path[35] from some

[35] We note that the "path" here will always be of length one due to the dynamics of our action.

$$F = (L_{1.1} \vee L_{2.1} \vee L_{3.1}) \wedge \ldots \wedge (L_{1.m} \vee L_{2.m} \vee L_{3.m})$$
$$G = (L_{1.1} \vee L_{2.1} \vee L_{3.1}) \vee \ldots \vee (L_{1.m} \vee L_{2.m} \vee L_{3.m})$$
$$H = (L_{1.1} \vee L_{2.1} \vee L_{3.1} \vee C_1) \wedge \ldots \wedge (L_{1.m} \vee L_{2.m} \vee L_{3.m} \vee C_m)$$
$$C = C_1 \wedge \ldots \wedge C_m$$
$$U = \neg C_1 \wedge \ldots \wedge \neg C_m$$

Figure 15. Formulas for describing the partitions used in the 3CNF reduction. Including the original formula *F*.

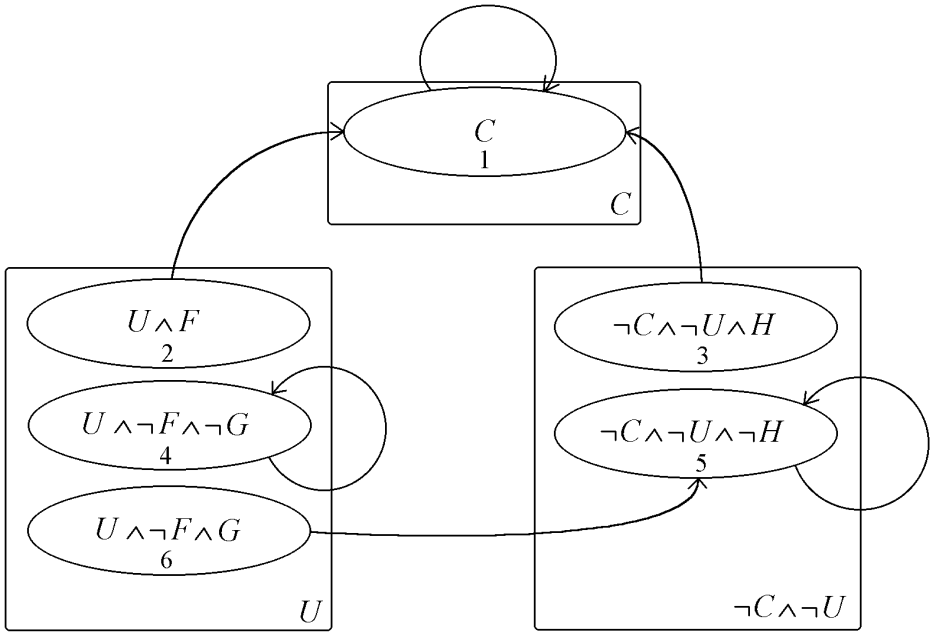

Figure 14. Initial and minimal stable partitions for the 3CNF reduction.

state in the block *U* to the block *C*, which is true if and only if the sub-block $U \wedge F$ is non-empty.

The numbered oval blocks in Figure 14 shows the final partition resulting from model minimization, except that some of the oval blocks shown may in fact be empty. To check that this partition is in fact stochastic bisimilarity on *M*, note the following: the blocks have uniform and well-defined rewards (see the square blocks); the transitions shown are deterministic and uniform within each block; and any two states in different blocks differ either on their immediate reward or on their reward at the next state.

Once an appropriate *k* for the bounded-size model minimization problem is selected, the problem will be answered "yes" if and only if the block $U \wedge F$ (block 2) is non-empty, and thus if and only if *F* is satisfiable, as desired. Selecting an appropriate *k* to achieve this property requires our reduction to determine which blocks in Figure 14 are

non-empty. We note that block 1 is always non-empty, and non-emptiness for block 2 implies non-emptiness for block 3 (simply set the $L_{j,i}$ to satisfy F and set some but not all C_j to get a member of block 3)—therefore checking whether all three of blocks 1 through 3 are non-empty is sufficient. Thus the appropriate value of k is β+3 where β is the number of non-empty blocks among blocks 4, 5, and 6. It remains to show that non-emptiness of blocks 4, 5, and 6 can be determined in polynomial time, by analysis of F.

We note that the validity of F can easily be checked: F is valid if and only if every clause C in F there exists a literal L such that both L and $\neg L$ appear in C. If F is valid, then H is also valid, and then blocks 4, 5, and 6 are all empty. If F is not valid, then $\neg F$ is satisfiable, implying the existence of at least one clause r in F that is falsifiable. The assignment to the X_i fluents that makes r false extended with all the C_i fluents true except C_r will be a member of block 6, and thus block 6 is non-empty when F is not valid. The formula $\neg F \wedge G$ is satisfiable if and only if $\neg F$ is satisfiable and has at least two clauses that do not share all their literals—this provides an emptiness test of block 5, as U can always be satisfied independently of F and G. The formula $\neg G$ is satisfiable if and only if no variable appears in F in both positive and negative form. Since $\neg G$ implies $\neg F$, determining the satisfiability of $\neg G$ determines the emptiness block 4. All of these emptiness determinations can be made in polynomial time in the size of F. ■

Lemma 11.1: Given equivalence relation E on Q, block B in E, block C a union of any non-empty set of blocks in E, and states p and q in B, if p and q do not fall in the same block of S-SPLIT(B, C, E) then p and q are not related in any fluentwise-stable partition refining E.

Proof: Suppose p and q are in different blocks of S-SPLIT(B, C, E). This implies that p and q fall into different blocks of $T_F(\alpha, f)$ for some action α and fluent f, where f is necessary to describe the block C. This implies that there are two states that differ only on their value of f, one that is in block C, and one that is not. Furthermore, any partition that distinguishes between these states, including any refinement of C, must also use f to do so. Any refinement of E contains a refinement of C, since E contains a refinement of C. Thus, the fluent f is necessary to describe at least one block in any refinement of E. It follows that p and q, being in different blocks of $T_F(\alpha, f)$ for some action α, cannot belong to the same block of any fluentwise stable refinement of E. ■

Theorem 11: Given a partition P, there is a unique coarsest fluentwise-stable stochastic bisimulation refining P. Iterating S-SPLIT using model reduction or partition iteration starting from P computes this bisimulation regardless of the order of block splitting.

Proof: The existence of a coarsest fluentwise-stable stochastic bisimulation refining P is guaranteed since the partition $\{\{q\} | \, q \in Q\}$ is a fluentwise-stable stochastic bisimulation refining P, and there are only finitely many partitions. Uniqueness of the coarsest such partition is proven by contradiction. Assume two distinct partitions E_1 and E_2 are both coarsest fluentwise-stable stochastic bisimulations refining P. Construct the new partition E refining P that equates any two states equated by either E_1 or E_2, as follows: E is the

symmetric transitive closure of E_1 union E_2, where the partitions are viewed as equivalence relations represented as sets of pairs. We note that this partition E is a coarsening of both E_1 and E_2 and thus any fluent necessary to represent any block in E must be necessary to represent at least one block E_1 and at least one block in E_2 (see proof of Lemma 11.1). This ensures that any two states related by either E_1 or E_2 must be in the same block of $T_F(\alpha, f)$ for any fluent required to define a block of E—since any such fluent is required to define a block of E_1 and a block of E_2, and both E_1 and E_2 are fluentwise stable. Then a simple induction shows that since any two states related by E are connected by a path made from E_1/E_2 arcs (i.e., a path of arcs drawn from E_1 union E_2), any two such states must be in the same block of $T_F(\alpha, f)$ for any fluent required to define a block of E. So E is fluentwise stable. Also, by Lemma 4.1 and Lemma 4.2, E must be a stochastic bisimulation. Therefore, E is a fluentwise-stable, stochastic bisimulation that is a coarsening of both E_1 and E_2, contradicting our assumption.

That iterating S-SPLIT using model reduction or partition iteration finds the coarsest fluentwise stable stochastic bisimulation follows directly from Lemma 11.1.■

Lemma 12.1: Given equivalence relation E on Q, block B in E, action α, and states p and q in B, if p and q do not fall in the same block of FS-SPLIT(B, C, E), where C is the union of any set of blocks in E, then p and q are not related in any fluentwise stable partition refining E that is also fluentwise representable.

Proof: First we note that S-SPLIT(B, C, E) is a coarsening of the unique coarsest *fluentwise-stable* partition E' refining E (by Lemma 11.1). It follows that any fluent needed to represent S-SPLIT(B, C, E) is also needed to represent E'. FS-SPLIT(B, C, E) is the fluentwise partition given by the set of fluents required to represent S-SPLIT(B, C, E), and thus must be a coarsening of any fluentwise partition including all the fluents needed to represent E'. But any representation of any fluentwise-stable partition refining E must use all the fluents needed in every representation of E', since E' is the coarsest such partition. So any fluentwise representation of a fluentwise-stable refinement of E must include all the fluents used in FS-SPLIT(B, C, E), and thus must separate p and q, as desired. ■

Theorem 12: Given a partition P, there is a unique coarsest stochastic bisimulation refining P even under the restriction that the partition be both fluentwise stable and fluentwise representable. Iterating FS-SPLIT using model reduction or partition iteration starting from P computes this bisimulation regardless of the order of block splitting.

Proof: Let E be the unique coarsest fluentwise-stable stochastic bisimulation refining P, which we know exists, by Theorem 11. The fluentwise-representable partition E' containing just those fluents required for representing E is a fluentwise-stable partition that is fluentwise representable—this follows because E' refines E without requiring any new fluents for representation. Since our choice of E guarantees that every fluentwise-stable partition refines E, every fluent needed to represent E is needed to represent any fluentwise-stable partition; therefore all such fluents must be included in any fluentwise representation of any fluentwise-stable partition. Thus E' is a unique coarsest fluentwise stable

stochastic bisimulation even under the restriction that it also be fluentwise representable. That iterating FS-SPLIT, using either model reduction or partition iteration, yields this partition follows directly from Lemma 12.1.■

Lemma 13.1: Suppose we are given an equivalence relation E on Q, where the blocks of both E and the partitions representing the factored MDP are represented as conjunctions of literals. Then, any states p and q that do not fall in the same block of $I_{\text{R-SPLIT}}(E)$ are not in the same block of any regression-stable partition refining E.

Proof: Suppose p and q fall into different blocks of $I_{\text{R-SPLIT}}(B, C, E)$. By the definition of R-SPLIT, p and q must fall into different blocks of $T_F(\alpha, f)$ for some action α and fluent f necessary to describe some block C of E and either p or q must be in the regression region of C. Without loss of generality, let p be in the regression region of C. Consider a regression-stable refinement E' of E—we show that p and q fall into different blocks of E'. Since p is in the regression region of C, p must be in the regression region of some sub-block C' of C in E'. Furthermore, because C is represented as a conjunction of literals, every fluent required to describe block C must be required to describe any sub-block of C— in particular, the fluent f is required to describe the block C'. Now we have that p is in the regression region of C', description of which requires the fluent f (for which p and q fall into different blocks of $T_F(\alpha, f)$). It follows that p and q must be separated by R-SPLIT(B', C', E') for any block B' of E' containing both p and q; thus, there can be no such B' in the regression-stable E', and p and q fall into different blocks in E', as desired. ■

Theorem 13: Given a partition P, there exists a unique coarsest regression-stable stochastic bisimulation refining P.

Proof: The existence of a coarsest regression-stable stochastic bisimulation refining P is guaranteed since the partition $\{\{q\} | q \in Q\}$ is a regression-stable stochastic bisimulation refining P, and there are only finitely many partitions. Suppose for contradiction that two distinct partitions E_1 and E_2 are both coarsest regression-stable stochastic bisimulations refining P. Construct the new partition E refining P that equates any two states equated by either E_1 or E_2, as follows: E is the symmetric transitive closure of E_1 union E_2, where the partitions are viewed as equivalence relations represented as sets of pairs. We note that this partition E is a coarsening of both E_1 and E_2 and thus any fluent necessary to represent any block in E must be necessary to represent at least one block E_1 and at least one block in E_2 (see proof of Lemma 11.1). This ensures that any two states related by either E_1 or E_2 must be in the same block of $T_F(\alpha, f)$ for any fluent required to define any block of E containing either state in its regression region—since any such fluent is required to define such a block of E_1 and such a block of E_2 and both E_1 and E_2 are regression stable. But then a simple induction shows that since any two states related by E are connected by a path of E_1/E_2 arcs, any two such states must be in the same block of $T_F(\alpha, f)$ for any fluent required to define any block of E containing either state in its regression region. So E is regression stable. In addition, by Lemma 4.1 and Lemma 4.2, E must also be a sto-

chastic bisimulation. Therefore, E is a regression-stable stochastic bisimulation that is a coarsening of both E_1 and E_2, which contradicts our assumption. ■

Lemma 14.1: Given a non-empty block B represented by a conjunction of literals Φ, each fluent f is mentioned in Φ if and only if f appears in every formula describing B.

Proof: (If) If f appears in every formula describing block B, then since Φ describes B, Φ must mention f. (Only if) Since Φ describes a non-empty block B it follows that Φ is satisfiable. Let v be a truth assignment to all the fluents such that v satisfies Φ, and let v' be v with the value of fluent f negated for some fluent f appearing in π . Since Φ describes B and is a conjunction of literals that mentions f, we know that v' is not in block B. Furthermore, since both v and v' satisfy all the same formulas that do not contain f, but B contains v and not v', any description of B must contain f. ■

Lemma 14.2: If every block in a partition P is representable with a conjunction of literals, every block of $I_{\text{R-SPLIT}}(P)$ is also so representable, under the assumption that the blocks in the partitions describing the MDP are also so representable.

Proof: Let C be a block of states. We define the "required fluents" of C, Req-Fluents(C), to be the set of fluents that are mentioned in every DNF block formula that describes block C. We define Determines(C) to be the intersection of the T_F partitions for each $F \in$ Req-Fluents(C). Note that any partition of the form Determines($\cdot$) is made up only of blocks representable by conjunctions of literals (given our assumptions about the MDP).

Let Regress(C) to be the partition Determines(C) modified so that any blocks B such that $T(B, \alpha, C) = 0$ for every action α are aggregated into a single block. Let S be a set of blocks. We use Determines(S) and Regress(S) to denote the intersection over members e of S of Determines(e) and Regress(e), respectively. Let s be a state. We define Reachable(s) to be the set of blocks C of P such that $T(s, \alpha, C) \neq 0$. For any block B, let Reachable(B) is the set of all blocks C such that some state s in B has $T(s, \alpha, C) \neq 0$. We prove that Regress(P) intersected with P and R is the same partition as $I_{\text{R-SPLIT}}(P)$, and that every block B of Regress(P) is an element of Determines(Reachable(B)). Thus, any block of $I_{\text{R-SPLIT}}(P)$ can be represented as a conjunction of literals, since it is the intersection of blocks from P, R, and a Determines($\cdot$) partition, where each block is representable as a conjunction of literals.

We now show that Regress(P) intersected with P and R is the same partition as $I_{\text{R-SPLIT}}(P)$. Let s and t be states from the same block of Regress(P) $\cap$ P $\cap$ R. Since s and t are in the same block of P, to be in different blocks of $I_{\text{R-SPLIT}}(P)$ they must fall in different blocks of some call to R-SPLIT(B', C, P) for some C in P where s and t are both in B'. States s and t are in different blocks of R-SPLIT(B', C, P) only if either s and t have different reward or are in different blocks of T_F for some F in Req-Fluents(C) and either s or t has a non-zero probability of transitioning to C. Since s and t are in the same block of R they must have the same reward, and since they are in the same block of Regress(C) they must either both have zero probability of transition to C or be in the same block of T_F for all F in Req-Fluents(C). So, s and t are in the same block of $I_{\text{R-SPLIT}}(P)$, and thus

that Regress(P) intersected with P and R refines $I_{\text{R-SPLIT}}(P)$.

Now consider s and t from the same block B of $I_{\text{R-SPLIT}}(P)$. Since $I_{\text{R-SPLIT}}(P)$ always refines both P and R, s and t must be the same blocks of P and R. We know that block B is not split by any call of the form R-SPLIT(B, C, P) for any $C \in P$, implying that either $T(B, \alpha, C) = 0$ for all α or every state in B falls in the same block of T_F for all F in Req-Fluents(C). Since s and t are both in B, they must be in the same block of Regress(C) for any $C \in P$, and therefore in the same block of Regress(P), Being in the same blocks of the partitions Regress(P), P, and R means s and t are in the same block of the intersection of those partitions and thus $I_{\text{R-SPLIT}}(P)$ refines Regress(P) intersected with P and R. Since Regress(P) intersected with P and R refines $I_{\text{R-SPLIT}}(P)$ and $I_{\text{R-SPLIT}}(P)$ refines Regress(P) intersected with P and R they must be the same partition.

It remains to show that any block B of Regress(P) is an element of Determines(Reachable(B)). Consider states s and t in B of Regress(P). For all blocks C of P, s and t are in the same block of Regress(C). So, by definition of Regress(C), whenever $T(B, \alpha, C) > 0$, s and t are in the same block of Determines(C). The set of blocks C from P where $T(B, \alpha, C) > 0$ is just Reachable(B), so s and t are in the same block, called B', of Determines(Reachable(B)). So, block B refines a block of Determines(Reachable(B)).

We now consider state $s' \in B'$ and show that $s' \in B$, to conclude that $B' = B$, completing our proof. We consider any state s in B, and show that s and s' fall into the same block of Regress(C) for every block C of P. It suffices to show that Reachable(s) equals Reachable(s') and that s and s' fall into the same block of T_F for any F in Req-Fluents(C) for C in Reachable(s). Consider C in Reachable(s). Note that any such C is a member of Reachable(B) and our choice of B' as the block of Determines(Reachable(B)) containing B implies that s' and s are in the same block of T_F for all fluents in Req-Fluents(C). It remains to show that Reachable(s) equals Reachable(s').

As just argued, s and s' fall into the same block of T_F for any fluent F in Req-Fluents(C) for any C in Reachable(s). This implies that Reachable(s) is a subset of Reachable(s'). The fact that Reachable(s') is a subset of Reachable(s) can be argued as follows. As just shown, s and s' fall in the same block of T_F for any F in Req-Fluents(C) for any C in Reachable(B). This implies that the transition probability from s or s' to any such C is the same. But since these probabilities must sum to one (because s can *only* reach blocks C that are reachable from B, as s is in B), s' cannot transition to any block C' not in Reachable(B), and hence Reachable(s') is a subset of Reachable(s), as desired.■

Theorem 14: Let M be a factored MDP with all partition blocks represented as conjunctions of literals. Given a starting partition P also so represented, iterating R-SPLIT using partition iteration computes the coarsest regression-stable stochastic bisimulation refining P, regardless of the order in which blocks are selected for splitting.

Proof: Lemma 14.2 implies that every block in the partition resulting from the application of the $I_{\text{R-SPLIT}}$ operation has a formula that is a conjunction of literals. Lemma 13.1 then implies that iterating $I_{\text{R-SPLIT}}$ using partition iteration returns the coarsest regression-stable stochastic bisimulation, which by Theorem 13 is unique. ■

Lemma 15.1: Given a fluentwise partition P and a minimal tree-represented factored MDP M, the partition computed by Add-relevant(P) is the partition $I_{\text{FS-SPLIT}}(P)$.

Proof: Let P' be the fluentwise partition returned by Add-relevant. Partition P' is fluentwise representable. Also, P' is a refinement of P since P' (as a set of fluents) contains P. We define fluentwise stability with respect to a partition to mean fluentwise stability with respect to every block of that partition. Below we show that any fluentwise partition omitting any fluent in P' is not fluentwise stable with respect to P, and that P' is fluentwise stable with respect to P. Thus, P' is the coarsest fluentwise-representable partition refining P that is fluentwise stable with respect to P, i.e. $I_{\text{FS-SPLIT}}(P)$, as desired.

For a partition to be fluentwise stable with respect to a fluent means that every pair of states in the same block of the partition must have the same probability distribution over that fluent for every action. If a fluent f' is tested in a minimal tree representation of the effect of some action α on some fluent f then any fluentwise partition omitting f' is not fluentwise stable with respect to f; two states differing only on f' must differ in their probability of setting f when taking α. To be fluentwise stable with respect to P, a partition must be stable with respect to all the fluents in P (as a set of fluents), because describing any block in P with a formula requires all fluents in P. It follows that Add-relevant constructs P' by adding to P only those fluents that cannot be omitted from a partition that is fluentwise stable with respect to P, as desired.

The independence assumptions implicit in the factored representation of the MDP M ensure that any fluent f' not mentioned in the tree for action α and fluent f has no effect on the probability of setting f when taking α. Specifically, states differing only on f' have the same probability of setting f when taking α. Partition P' contains every fluent in any tree defining the effect of any action on any fluent in P, so that two states in the same block can only differ on fluents not mentioned in any such tree. It follows that any two states in the same block of P' have the same probability of setting any fluent in P, and thus that P' is fluentwise stable with respect to P.■

Theorem 15: Given a minimal tree-represented MDP, model reduction using FS−SPLIT yields the same partition that state-space abstraction yields, and does so in polynomial-time in the MDP representation size.

Proof: Since state space abstraction iterates Add-relevant starting with the fluentwise partition Fluents(R) until convergence, it follows directly from 0 that state-space abstraction and iterating FS-SPLIT starting with Fluents(R) find the same partition. A simple analysis of partition iteration shows that the first iteration returns Fluents(R) when using FS-SPLIT. Theorem 12 then implies that model reduction using FS-SPLIT and state space abstraction yield the same partition.

We now show the polynomial-time complexity claim. To obtain this complexity, the basic method must be optimized to make only the linearly many FS-SPLIT calls described below, avoiding unnecessary calls, as follows. When the partition P is fluentwise represented, the partition P' = FS-SPLIT(B, C, P) does not depend on the choice of B or

C from *P* because the same set of fluents are used in every block formula. Thus, *P'* will not be further split by calls of the form FS-SPLIT(*B'*, *C*, *P'*), where *B'* is a block of *P'* and *C* is a block of *P*. This observation implies that partition iteration can compute $I_{\text{FS-SPLIT}}(P)$ with only one call to FS-SPLIT(*B*, *C*, *P*), using any blocks *B* and *C* in the partition *P*. We further note that each call to FS-SPLIT increases the number of fluents in the partition representation, except the last call, which does not change the partition. Thus only linearly many calls to FS-SPLIT can be made during partition iteration. We can conclude that partition iteration terminates in polynomial time, by showing that each call to FS-SPLIT terminates in polynomial time.

Consider the call FS-SPLIT(*B*, *C*, *P*), where *B* and *C* are blocks of *P*, and *P* is a fluentwise-represented partition. Every fluent in the set defining *P* is present in every DNF formula defining any block of *P*. So, for any choice of *B* and *C*, the resulting partition must be fluentwise partition represented by the set of fluents that appear anywhere in the trees defining the effects of the actions on fluents in *P*, together with any fluents in *P* or appearing in the tree defining *R*. This set is computable in time polynomial in the size of those trees plus the number of fluents in *P*. ■

Lemma 16.1: Let *V* be a tree-represented value function, where P_V is the partition given by the tree. Let α be an action, and for any block *C* of P_V, let Φ_C denote the conjunction of literals describing *C*. We then have the following.

> The partition computed by PRegress(*V*, α) is the intersection over all blocks *C* of P_V of Regression-determining(Φ_C, α).

Proof: We use induction on the depth of the tree for *V*. As the base case suppose *V* is a leaf node. Here, PRegress returns the partition {Q}, and Regression-determining(true, α) also returns {Q}. In the inductive case, let fluent *f* be the fluent tested at the root of the tree, and assume the lemma for the sub-trees. Let $P_\cap$ be the intersection over all blocks $C \in P_V$ of Regression-determining(Φ_C, α). We show that $P_\cap$ is PRegress(*V*, α).

We start by noting that the partition returned by PRegress(*V*, α) is built by refining $T_F(\alpha, f)$ using Replace. Since *f* is at the root of *V*.Tree every formula describing any block *C* of P_V includes *f*. In particular, the conjunction describing any block of *V* must contain *f*, since it is on every path from the root of *V* to a leaf, and so *f* must be in Fluents(Φ_C)fp. Thus for every block *C* of P_V the call to Partition-determining(Fluents(Φ_C), α) must be fluentwise stable with respect to *f*, since all states in the same block of Partition-determining(Fluents(Φ_C), α) must be in the same block of $T_F(\alpha, f')$ for any fluent *f'* in Fluents(Φ_C). Consider the partition variable P_C in the pseudo-code for Regression-determining(Φ_C, α), after it is assigned. Since every block in P_C is a block from Partition-determining(Fluents(Φ_C), α), any such block must be fluentwise stable with respect to *f*. We note that the union of all blocks in P_C is the regression region for *C*, as defined in section 4.6. It follows that every state in the regression region for a block *C* is in a fluentwise-stable block (with respect to *f*) in the partition Regression-determining(Φ_C, α), and thus in any partition refining this partition. Every state must be carried to some state under action α, so every state is in the regression region for some block of P_V. So $P_\cap$ must

be fluentwise stable with respect to f; so $P_\cap$ refines $T_F(\alpha, f)$.

We now analyze the refinement of $T_F(\alpha, f)$ returned by PRegress and show that this refinement is the same as the refinement of $T_F(\alpha, f)$ by $P_\cap$. PRegress(V, α) is computed by replacing each block B of $T_F(\alpha, f)$ with the intersection of all the partitions resulting from relevant recursive calls to PRegress, restricted to block B. A recursive call on an immediate sub-tree of V is *relevant* to block B if the value of f leading to that sub-tree has a non-zero probability after taking action α in block B (this probability is uniform throughout B). By the induction hypothesis, each sub-tree partition is the intersection of Regression-determining($\Phi_{C'}$, α) for all blocks C' of the partition represented by the sub-tree. Each such block C' becomes a block C of P_V when restricted to the value of f determining the branch for the sub-tree containing C'—the formula Φ_C will be $\Phi_{C'}$ conjoined with the appropriate literal for f. The refinement of B in PRegress(V, α) is therefore the intersection of Regression-determining($\Phi_{C'}$, α) for all the blocks C' of all sub-trees relevant to B, restricted to B.

Consider a block B of $T_F(\alpha, f)$ and two states i and j from B in different blocks of PRegress(V, α). Our analysis of PRegress(V, α) just above implies that i and j must be in different blocks of Regression-determining($\Phi_{C'}$, α) for some block C' of a sub-tree of V relevant to B. Let C be the block formed by restricting C' to the relevant value of f. Any state in B has a non-zero block transition probability to C' if and only if that state also has a non-zero block transition probability to C—this follows from the definition of "relevant". From this, one can show that i and j are also in different blocks of Regression-determining(Φ_C, α). It follows that $P_\cap$ refines PRegress(V, α).

Now consider a block C of P_V and the corresponding formula Φ_C, and any two states i and j in the same block of PRegress(V, α). For any fluent f' in Φ_C, we have either that Φ_C is always false after performing α whether starting from i or from j, or that both states have an equal probability of transitioning to a state where f' is true after performing α. This implies that states i and j are either both in block Q_0 or both in block of P_C, respectively, in the pseudo-code for Regression-determining(Φ_C, α). We conclude that any two states in the same block of PRegress(V, α) must also be in the same block of Regression-determining(Φ_C, α) for any block C of P_V—thus PRegress(V, α) must refine $P_\cap$. Since PRegress(V, α) and $P_\cap$ refine each other, they must be equal, as desired. ■

Lemma 16.2: Given action α and value function V, Regress-action(V, α) on MDP M intersected with V.tree gives the partition computed by $I_{\text{R-SPLIT}}(V.\text{Tree})$ on MDP M_{π_α}.

Proof: We say that a partition P' is a *regression of P for MDP M* if P' = R-SPLIT(B, C, P') for any blocks B of P' and C of P, where R-SPLIT is computed relative to M. It is not hard to show that the coarsest regression of P refining P for any M is $I_{\text{R-SPLIT}}(P)$ for M. Let P be the partition V.Tree, and let P' be $P \cap$ Regress-action(V, α) on MDP M. We show that P' is the coarsest regression of P refining P for M_{π_α}, to conclude that $P' = I_{\text{R-SPLIT}}(P)$ relative to M_{π_α}.

Since P' is formed by intersection with P, P' refines P. We show that P' is a regression of P relative to M_{π_α}. Let i and j be any two states in the same block B of P'.

Then we need to show that *i* and *j* are in the same block of R-SPLIT(*B*, *C*, *P'*) relative to M_{π_α} for any block *C* of *P*. We note that Regress-Action(*V*, α) uses partition intersection with the reward partition to return a partition that refines the *R* partition. Thus, the states *i* and *j* must have the same reward. States *i* and *j* must also belong to the same block of Regression-determining(Φ_C, α) for any block *C* of *P* by Lemma 16.1 since Regress-action(*V*, α) returns a refinement of PRegress(*V*, α). We can then see that states *i* and *j* must belong to the same block of Block-split(*B*, *C*, α) (as computed by the code of Figure 7 with Partition-determining replaced by Regression-determining to compute R-SPLIT, as discussed in section 4.6)—and thus to the same block of R-SPLIT(*B*, *C*, *P'*) for M_{π_α}, as desired. It follows that *P'* is a regression of *P* for M_{π_α}.

We now argue that *P'* is the coarsest regression of *P* for M_{π_α}. Suppose not, and consider such coarser regression *P"*, and consider states *i* and *j* in the same block of *P"* but in different blocks of *P'*. Note, based on the pseudo-code for Regress-action in Figure 11, that if *i* and *j* are in different blocks of *P'* then they must either be in different blocks of *P*, have different rewards, or (using Lemma 16.1) be in different blocks of Regression-determining(Φ_C, α) for some block *C* of *P*. In each of these cases, we can show that the block *B* of *P"* containing *i* and *j* is split to separate *i* and *j* into different blocks of R-SPLIT(*B*, *C*, *P'*) for some block *C* of *P*, contradicting our assumption about *P"*.■

Lemma 16.3: Given policy π and value function *V*, Regress-policy(*V*, π) on MDP *M* intersected with *V*.tree gives the partition computed by $I_{\text{R-SPLIT}}$(*V*.Tree) on MDP M_π intersected with π.Tree.

Proof: Regress-policy(*V*, π) returns the partition that refines π.Tree by intersecting each block *b* of π.Tree with Regress-action(*V*, α_b) where α_b is the action labeling block *b*, i.e., π.Label(*b*). Let *M'* be the MDP *M* extended by adding a new action α' defined to so that for each state *s*, α' behaves identically to π(*s*) in *M*. Then Regress-policy(*V*, π) in *M* gives the same partition as Regress-action(*V*, α') in *M'* intersected with π.Tree. Applying Lemma 16.2 gives that Regress-policy(*V*, π) intersected with *V*.tree is the same partition as $I_{\text{R-SPLIT}}$(*V*.Tree) for MDP $M'_{\pi_{\alpha'}}$ intersected with π.Tree. To complete the proof, we note that $M'_{\pi_{\alpha'}} = M_\pi$ by the construction of α' and M_π.■

Lemma 16.4: Given tree-represented value functions V_1 and V_2, with corresponding partitions V_1.Tree refining V_2.Tree, we have all of the following monotonicity properties:

1. PRegress(V_1, α) refines PRegress(V_2, α) for any action α,
2. Regress-action(V_1, α) refines Regress-action(V_2, α) for any action α,
3. Regress-policy(V_1, π) refines Regress-policy(V_2, π) for any policy π, and
4. $I_{\text{R-SPLIT}}$(V_1.Tree) refines $I_{\text{R-SPLIT}}$(V_2.Tree).

Proof: We first show some properties of PRegress and of partitions represented as trees that will be useful for proving that PRegress(V_1, α) refines PRegress(V_2, α) for any action α. It follows from Lemma 16.1 that the partition returned by PRegress(*V*, α) for a tree-

represented value function V depends on only the blocks of V.Tree and not on the structure of the tree itself. Another useful property is that for any value function V' which refines V, both represented as trees, we can change the structure of V' to have the same root variable as V without changing the represented partition. We will prove this property by construction. Let X be the root variable of V.Tree, we first note that X must be used to describe any block of V'.Tree because of the following three facts. (1) Every block formula for any block of V.Tree or V'.Tree is a conjunction of literals. (2) Every block of V.Tree mentions X. (3) Every block of V'.Tree is a sub-block of a block of V.Tree. For each value x of X, let the tree τ_x be the tree V'.Tree, with every sub-tree that has root variable X replaced by the immediate sub-tree of that sub-tree corresponding to x. Now construct a tree τ with the root node labeled with X, and the sub-tree for each value x of X being τ_x. Noting that X must occur on every root-to-leaf path in V'.Tree, it is easy to show that the τ represents the same partition as V'.Tree, but has the same root variable as V.Tree.

We now prove by induction on the height of V_2.Tree that PRegress(V_1, α) refines PRegress(V_2, α) for any action α. For the base case, consider a value function V_2 consisting of a single leaf node. In this case PRegress(V_2, α) returns $\{Q\}$, which is refined by every partition so the property is trivially true. In the inductive case, first modify V_1.Tree so that it has the same root variable as V_2.Tree without changing the partition represented, as just described in the previous paragraph. Examining the pseudo-code for PRegress, given in Figure 11, we note that the same X is selected by the calls PRegress(V_1, α) and PRegress(V_2, α) for any action α, and therefore the assignment P_x.Tree = $P_{x|\alpha}$.Tree assigns the same starting tree for both calls. We now observe that Subtree(V_1, x) refines Subtree(V_2, x) for every value x of X, since V_1 refines V_2, and that the height of Subtree(V_2, x) is less than the height of V_2.Tree. Therefore, by the induction hypothesis we have that every P_{xi} in the call PRegress(V_1, α) refines the corresponding P_{xi} in the call PRegress(V_2, α).

Let T_1 be the T calculated to replace block B of P_x.Tree in the call PRegress(V_1, α) and let T_2 be the T calculated to replace B in the call PRegress(V_2, α). We now show that T_1 refines T_2. For states p and q to be in the same block of T_1 they must be in the same block of P_{xi} in the call PRegress(V_1, α) for each x_i such that Pr($X = x_i$) in the distribution $P_{x|a}$.Label(B) is greater than zero. Therefore, since P_{xi} in the call PRegress(V_1, α) refines the corresponding P_{xi} in the call PRegress(V_2, α), p and q must be in the same block of P_{xi} in the call PRegress(V_2, α) for each x_i such that Pr($X = x_i$) in the distribution $P_{x|a}$.Label(B) is greater than zero. Since these P_{xi} are intersected to obtain T_2, p and q must be in the same block of T_2 proving that T_1 refines T_2 when replacing any block B. Part 1 of the lemma follows. The second and third parts of the lemma follow directly from the first and second, respectively, along with an examination of the pseudo-code in Figure 11 and basic properties of intersection on partitions relative to the partition refinement relation.

To prove the last part of the lemma, that $I_{\text{R-SPLIT}}(V_1.\text{Tree})$ refines $I_{\text{R-SPLIT}}(V_2.\text{Tree})$, we show that any two states in the same block of $I_{\text{R-SPLIT}}(V_1.\text{Tree})$ are in the same block of $I_{\text{R-SPLIT}}(V_2.\text{Tree})$. Let p and q be two states from the same block B_1 of $I_{\text{R-SPLIT}}(V_1.\text{Tree})$. This means that p and q must be in the same block of V_1.Tree and in the same block of R-SPLIT(B_1, B_1', $I_{\text{R-SPLIT}}(V_1.\text{Tree})$) for any block B_1' of V_1.Tree.

In order to show that p and q are in the same block of $I_{\text{R-SPLIT}}(V_2.\text{Tree})$ we show that they are in the same block of V_2.Tree and in the same block of R-SPLIT(B_2, B_2', P) for any block B_2' of V_2.Tree, any block B_2 containing both p and q, and any partition P containing block B_2. Since V_1.Tree refines V_2.Tree, the fact that p and q are in the same block of V_1.Tree directly implies that they are in the same block of V_2.Tree. For any $B_2' \in V_2$.Tree, consider the set π of blocks $\{B_1' \mid B_1' \in V_1.\text{Tree}, B_1' \subseteq B_2'\}$. Note that since p and q are in the same block of $I_{\text{R-SPLIT}}(V_1.\text{Tree})$, they must agree on the probability of transition to any block in V_1.Tree. Let p and q both be in block B_2 and B_2 be a block of partition P. If for every member B_1' of π, the probability of transitioning from both p and q to B_1' under an action α is zero, then p and q are in the same block of R-SPLIT(B_2, B_2', P) since their probabilities of transitioning to B_2' are both zero and thus. Now consider for some member B_1' of the set, the probability of transitioning from either p or q to B_1' is non-zero under some action α. Then, since p and q are in the same block of $I_{\text{R-SPLIT}}(V_1.\text{Tree})$, they must be in the same block of $T_F(\alpha, f)$ for every fluent f needed to describe block B_1'. Since B_1' is a sub-block of B_2' and both B_1' and B_2' can be represented as a conjunction of literals every fluent needed for B_2' is needed for B_1'. Therefore, p and q must be in the same block of $T_F(\alpha, f)$ for every fluent f needed to describe block B_2' and thus be in the same block of R-SPLIT(B_2, B_2', P). Using one of these two cases for each action, we get that p and q are in the same block of R-SPLIT(B, B_2', P), whenever p and q are both in block B and B is in P.■

Theorem 16: For any tree-represented MDP M and policy π, SSA(π) produces the same resulting partition as partition iteration on M_π using R-SPLIT starting from the partition π.Tree.

We first show that SSA(π) and partition iteration of M_π using R-SPLIT, starting from the partition π.Tree, written $\text{PI}_{\text{R-SPLIT}}(\pi.\text{Tree}, M_\pi)$, compute the same partition. We notate the sequence of partitions produced by partition iteration as follows: $P_0 = \pi.\text{Tree}$, and $P_{i+1} = I_{\text{R-SPLIT}}(P_i)$. The partition $\text{PI}_{\text{R-SPLIT}}(\pi.\text{Tree}, M_\pi)$ equals P_m, for all m greater than the number t_P of iterations to convergence. Likewise, denote the sequence of factored value functions produced by SSA as follows: $V_0 = R$, and $V_{i+1} = \text{Regress-policy}(V_i, \pi)$. Likewise, the partition SSA(π) equals V_m, for all m greater than the number t_V of iterations to convergence. By induction on the number n of iterations of partition iteration, we show that V_{n+1}.Tree refines P_n and P_{n+1} refines V_n.Tree, for all $n > 0$, and conclude that SSA(π).Tree equals $\text{MR}_{\text{R-SPLIT}}(\pi.\text{Tree}, M_\pi)$, as desired, by considering $n > \max(t_P, t_V)$.

For the base case, consider n equal to 1. Since, by inspection, Regress-policy($\cdot$,π).Tree always refines the partition π.Tree, for any policy π, we know that V_1.Tree refines P_0. Likewise, since, by inspection, the partition $I_{\text{R-SPLIT}}(P)$ always refines the reward partition R.Tree, for any partition P, we know that P_1 refines V_0.Tree. For the inductive case, we first show that P_{n+1} refines V_n.Tree. By a nested induction on n, we can show that P_{n+1} refines P_0, using the fact that $I_{\text{R-SPLIT}}(P)$ refines P, for any P. Thus,

(1) $$P_{n+1} = P_{n+1} \cap P_0 = I_{\text{R-SPLIT}}(P_n) \cap \pi.\text{Tree}.$$

But P_n refines V_{n-1} by the induction hypothesis, so Lemma 16.4 implies that $I_{\text{R-SPLIT}}(P_n)$ refines $I_{\text{R-SPLIT}}(V_{n-1})$. Together with equation (1), this implies that P_{n+1} refines $I_{\text{R-SPLIT}}(V_{n-1}) \cap \pi.\text{Tree}$. By applying Lemma 16.3, we derive that P_{n+1} refines Regress-policy(V_{n-1}, π) $\cap$ V_{n-1}.Tree, which is just $V_n \cap V_{n-1}$, by definition. It is straightforward to show by a nested induction on n that V_n refines V_{n-1}, using Lemma 16.4, so we conclude that P_{n+1} refines V_n.

That V_{n+1}.Tree refines P_n is proven similarly: first, $V_{n+1} = V_{n+1} \cap V_n$ = Regress-policy(V_n, π) $\cap$ V_n. Applying Lemma 16.3, we have $V_{n+1} = I_{\text{R-SPLIT}}(V_n) \cap \pi.\text{Tree}$. But V_n refines P_{n-1} by the induction hypothesis, so Lemma 16.4 implies that $I_{\text{R-SPLIT}}(V_n)$ refines $I_{\text{R-SPLIT}}(P_{n-1})$. With $V_{n+1} = I_{\text{R-SPLIT}}(V_n) \cap \pi.\text{Tree}$, we have that V_{n+1} refines $I_{\text{R-SPLIT}}(P_{n-1}) \cap \pi.\text{Tree}$, which is just P_n since $I_{\text{R-SPLIT}}(P_{n-1}) \cap \pi.\text{Tree} = P_n \cap P_0 = P_n$.■

Theorem 17: The policy improvement "for" loop in SPI computes $I_{\text{R-SPLIT}}(V_\pi.\text{Tree})$.

Proof: Let b be a block in π.Tree. We note that V_π in SPI is a factored value function computed by SSA, and so V_π must be a fixed-point of Regress-policy($\cdot, \pi$). This implies that V_π.Tree must refine π.Tree, and, by examining the Regress-policy pseudo-code, that blocks b' in V_π that refine b are also in Regress-action(V_π, $\pi(b)$). Combine these to get that Regress-action(V_π, $\pi(b)$) refines $\{\neg b\} \cup \{b' \mid b' \in V_\pi \wedge b' \subseteq b\}$. We also have

(2) $$\bigcap_{\alpha \in A} \text{Regress-action}(V_\pi, \alpha) \text{ refines } \pi.\text{Tree} \cap V_\pi.\text{Tree},$$

since b was arbitrary. Given that partition intersection is associative and commutative, the policy improvement "for" loop in SPI can be seen to iterate over the actions to compute

$$\pi.\text{Tree} \cap \bigcap_{\alpha \in A} \text{Regress-action}(V_\pi, \alpha).\text{Tree}.$$

Equation (2) then implies that the computed partition is

(3) $$\bigcap_{\alpha \in A} \left(\text{Regress-action}(V_\pi, \alpha) \cap V_\pi.\text{Tree}\right), \text{ which is } \bigcap_{\alpha \in A} I_{\text{R-SPLIT}}(V_\pi.\text{Tree}) \text{ in } M_{\pi_\alpha},$$

by applying Lemma 16.2 to each of the partitions in the intersection. It is possible to show that for value function V and MDP M' with action space A',

(4) $$I_{\text{R-SPLIT}}(V) \text{ in MDP } M' = \bigcap_{\alpha \in A'} \bigcup_{B \in V.\text{Tree}} \bigcap_{C \in V.\text{Tree}} \text{Block-split}(B, C, \alpha) \text{ in } M',$$

where the intersections are partition intersections[36], and the union is a simple set union, treating the partitions as sets of blocks (the union combines partitions of disjoint sets to get a partition of the union of those disjoint sets). Applying this to each of the terms in the intersection in Equation (3), noting that the only action available in M_{π_α} is α yields

[36] The resulting partition has a block for each pair of blocks in the partitions being intersected, representing the intersection of those two blocks, with empty blocks in the result removed.

$$\bigcap_{\alpha \in A} \bigcap_{\alpha' \in \{\alpha\}} \bigcup_{B \in V_\pi.\text{Tree}} \bigcap_{C \in V_\pi.\text{Tree}} \text{Block-split}(B, C, \alpha') \text{ in } M_{\pi_\alpha}$$

for the partition. Simplifying and noting Block-split(B, C, α) is the same in M_{π_α} and M,

(5) $$\bigcap_{\alpha \in A} \bigcup_{B \in V_\pi.\text{Tree}} \bigcap_{C \in V_\pi.\text{Tree}} \text{Block-split}(B, C, \alpha) \text{ in } M$$

is the computed partition. Finally, applying Equation (4) gives $I_{\text{R-SPLIT}}(V_\pi.\text{Tree})$ in M.■