Solving Large Markov Decision Processes (depth paper)

Yilan Gu
Department of Computer Science
University of Toronto
Toronto, Ontario, Canada
yilan@cs.toronto.edu

March 30, 2003

Contents

1 Introduction

Markov decision processes (MDPs) [4, 5] are a natural and basic formalism for decision-theoretic planning and learning problems in stochastic domains (e.g., [21, 11, 88, 90, 87]). In the MDP framework, the system environment is modeled as a set of states. An agent performs actions in the environment, and the outcomes of each action result in different states with certain probabilities. Moreover, the agent earns rewards according to states visited and actions taken. Solving the planning problem modeled in MDP is equivalent to looking for a policy (i.e., universal plan) that can achieve maximal long-term expected reward. Basic solution techniques such as value iteration and policy iteration are well-understood [4, 47, 5, 73].

Traditional MDP representation and solution techniques require explicit (flat) state representations, while decision and learning problems often involve a tremendous number of explicit states. In other words, it is impractical to solve real-world problems that generally have very large state spaces with traditional MDP algorithms, since the computational complexity of traditional solution techniques for MDPs are at least polynomial in the size of the state space. Recently, researchers have tried to tackle large state space MDPs from different perspectives. Mainly, two broad classes of abstraction methods have been proposed: (a) temporal abstraction and task decomposition approaches [83, 57, 63, 72, 45, 85]; and (b) state space abstraction via logical representations [20, 10, 13, 16, 15, 30, 38, 58]. Generally speaking, temporal abstraction and task decomposition approaches try to compress the scale of the state space by introducing subtasks or complex actions which allow a broad range of time scales for different actions. For example, [85, 45] introduces *macro-actions*, local policies on subsets of the state space (regions). Based on given macro-actions, the original state space is replaced by a hierarchical model including only the border states which at are boundaries of regions. This significantly reduces the size of the state space and greatly speeds up the computation.

On the other hand, state space abstraction via logical representations aims at avoiding enumerating explicit states by describing the system through state features (or aspects), aggregating 'similar' states, and eliminating certain aspects of the state description to reduce the effective state space during reasoning. For instance, the relational MDP (RMDP) model [38, 58] groups similar ground objects together as different classes, giving fluent (predicate or function) schemata and action schemata defined over object classes instead of using explicit states and actions. This representation not only makes it possible to describe large state-space problems, but is also capable of specifying similar decision-making problems (related MDPs) by using one single RMDP model and obtaining a universal optimal policy for the objects in the same class. State space abstraction methods often involve value function decomposition, while temporal abstraction methods often involve hierarchical subtask decomposition. However, the partition of temporal abstraction and state space abstraction is not so absolute. Some approaches [28, 27, 2, 31] include aspects of both. The MAXQ [28, 27] method, as an example, is mainly a temporal abstraction approach, but it also has the aspects of state space abstraction and value function decomposition.

In this article, we review the conventional specification of MDPs, the classical solutions, and different approaches within the two classes of abstraction methods. In the next section,

we go over fundamentals of specifying MDPs and basic techniques for looking for optimal solutions for planning problems modeled in MDPs. In Section 3, we review the situation calculus, dynamic Bayesian networks (DBNs) and other logical representations. In Section 4, we discuss various temporal abstraction and process decomposition approaches such as options and macro-actions, Markov task decomposition, hierarchical abstract machines (HAMs), and the MAXQ method. Then, Section 5 gives an overview of MDPs represented and solved via logical representations, including factored MDPs (FMDPs), first-order MDPs (FOMDPs), relational MDPs (RMDPs) and models using the Integrated Bayesian Agent Language (IBAL). Finally, we briefly discuss potential future work.

2 Markov Decision Processes

The *Markov decision process* (MDP) framework is adopted as the underlying model [21, 3, 11, 12] in recent research on *decision-theoretic planning* (DTP), an extension of classical artificial intelligence (AI) planning. It is also used widely in other AI branches concerned with acting optimally in stochastic dynamic systems.

2.1 General Problem Description

In this subsection, we first review the classical models, and next we talk about classical algorithms for looking for optimal solutions for MDPs.

2.1.1 Model Description

An MDP is a tuple $< \mathcal{S}, \mathcal{A}, T, R >$, where every component is described as follows.

- *State space* $\mathcal{S}$ is a set of distinct states, which describe the system properties. How one defines states can vary with particular applications. It is common to assume that the states capture all information relevant to the agent's decision-making process. Generally, the number of states can be finite or infinite, discrete or continuous. In this article, unless otherwise stated, we will focus on finite discrete models.

- *Action set* $\mathcal{A}$ is a set of actions that can be performed by the agent controlling the dynamic system. Again, the set of actions can be finite or infinite; and, unless otherwise stated, we will focus on finite action sets in this article.

- *Transition function* $T : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0, 1]$ captures the fact that actions affect the system stochastically. In particular, $T(s, a, s') = Pr(s'|s, a)$ denotes the probability of transition to state s' given that action a is executed in current state s. Note that the transition function exhibits the *Markov property* which says that the probability of transition to some state s' in next step depends only on the current state and action, and therefore is independent of previous states and actions given the current state.

- *Reward function* $R : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{R}$ is a mapping from state-action pairs to real numbers. Intuitively, $R(s, a)$ is the reward that the agent would get after performing action a in

state s.[1]

2.1.2 Time Horizon and Discount Factor

The objective considered for MDPs is to maximize the expected cumulative reward gathered by the agent acting over some time frame. This time frame is normally measured in *stages*, where the occurrence of an event, such as the performance of a stochastic action, constitutes one stage. The *horizon* specifies how long (for how many stages) the agent will act. It can be finite or infinite. *Finite horizon* means that the agent executes only finitely many steps, while *infinite horizon* means that the agent may execute the actions forever in the system. In this article, unless otherwise stated, we will focus on infinite horizon. According to different formalizations of the expected reward gathered by the agent over time, there are two types of objective criteria: *discount* infinite horizon and *average* infinite horizon. In the discount infinite horizon, a *discount factor* $\gamma \in [0, 1)$ is introduced to indicate how rewards earned in different time stages should be weighted. Intuitively, future rewards count for less. They will be scaled down by γ^k after k stages from any starting point, and the overall expected reward is the sum of these discounted rewards over the infinite horizon. In the average infinite horizon, the expected reward is counted as an average over stages of the sum of the rewards gathered by the agent at each stage from the very beginning. Again, unless otherwise stated, we will focus on discounted infinite horizon in this article.

2.1.3 Observations, Policies and Value Function

In any planning or learning problem modeled in MDPs, an agent needs to choose actions (i.e., make decisions) based on *observing* at which state the agent currently is. Hence, a *policy* is defined as a plan that specifies which action the agent should execute at the current stage based on the information it has observed. Policies can be *stationary* or *non-stationary*, *deterministic* or *non-deterministic*. Intuitively, if the choices of actions are independent of the stage, this policy is stationary; otherwise, it is non-stationary. If for any current state, there is one any only one action can be chosen, this policy is deterministic, otherwise it is non-deterministic. In this article, unless otherwise stated, the policies we consider are stationary deterministic policies, which are mappings from state spaces to action spaces.

Every policy is associated with a *value function*. Let $V^\pi : \mathcal{S} \to \mathcal{R}$ be the value function, i.e., an expected cumulative reward function, for a fixed policy π. V^π depends on the state in which the process starts. In a discounted infinite horizon MDP,

$$V^\pi(s) \;=\; E_\pi\{\sum_{t=0}^{\infty} \gamma^t R(s_t) | s_0 = s\}.$$

It satisfies the following Bellman equation [4]

$$V^\pi(s) \;=\; R(s, \pi(s)) + \gamma \sum_{t \in \mathcal{S}} T(s, \pi(s), t) V^\pi(t). \tag{1}$$

[1]Sometimes the reward function may be defined in a different way as a mapping $R : \mathcal{S} \to \mathcal{R}$, and $R(s)$ which represents the reward that the agent would get when it is in state s. Or, $R : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathcal{R}$ represents that the reward is a function of a starting state, a performed action and an ending state.

V^π for any fixed policy π is the unique solution to its Bellman equation. There are several ways to compute π. For example, we can use *successive approximation* starting with any V_0^π, computing the *n-stage-to-go* function V_n^π as

$$V_n^\pi \;=\; R(s,\pi(s)) + \gamma \sum_{t\in\mathcal{S}} T(s,\pi(s),t) V_{n-1}^\pi(t),$$

and obtaining V^π when $n \to \infty$ (Practically, the iteration will terminate once it satisfies $|V_n^\pi - V_{n-1}^\pi| < \delta$ for some small positive real number δ). Or, assume that there are n states in the system, we can compute V^π by solving the following linear system for unknown vector $X = (V^\pi(s_1), \cdots, V^\pi(s_n))^T$:

$$X \;=\; R + \gamma\, P\, X,$$

where $R = (R(s_1,\pi(s_1)), \cdots, R(s_n,\pi(s_n)))^T$ and $\mathcal{P}$ is the transition matrix with elements $p_{i,j} = T(s_i,\pi(s_i),s_j)$ for any $1 \le i,j \le n$.

The goal for any given MDP is to find a policy achieving as much expected reward as we can under any situation. Formally, we are looking for an *optimal* policy π^* such that $V^{\pi^*}(s) \ge V^\pi(s)$ for any $s \in \mathcal{S}$ and any policy π. It has been proved that any optimal solution $V^*(s) = V^{\pi^*}(s)$ of the value function for some π^* satisfies the following Bellman optimality equation

$$V^*(s) \;=\; \max_{a\in\mathcal{A}} \left[R(s,a) + \gamma \sum_{t\in\mathcal{S}} T(s,a,t) V^*(t) \right]. \tag{2}$$

2.2 Classical Algorithms (I): Dynamic Programming

To achieve the goal of finding an optimal policy for an MDP, the use of *dynamic programming* (DP) algorithms is proposed. The key idea of DP is the use of value functions to organize and generate good policies. There are two classical DP algorithms: policy iteration and value iteration.

2.2.1 Policy Iteration

Policy iteration begins with an arbitrary policy π_0, then iteratively computes π_{i+1} from π_i ($i \ge 0$). Each iteration has two steps, *policy evaluation* and *policy improvement*.

1. We first evaluate the policy π_i, i.e., compute the value function $V^{\pi_i}(s)$ for each $s \in \mathcal{S}$.
2. Next, we improve the policy, i.e., for each $s \in \mathcal{S}$, find an action a^* which maximizes

$$Q_{i+1}^{\pi_i}(s,a) \;\stackrel{def}{=}\; R(s,a) + \gamma \sum_{t\in\mathcal{S}} T(s,a,t) V^{\pi_i}(t).$$

   If $Q_{i+1}^{\pi_i}(s,a^*) > V^{\pi_i}(s)$, let $\pi_{i+1}(s) = a^*$; otherwise, let $\pi_{i+1}(s) = \pi_i(s)$.

The algorithm's iteration continues until $\pi_{j+1}(s) = \pi_j(s)$ at some iteration j for every state $s \in \mathcal{S}$. For a finite MDP in which the state space and action space are finite, this process must converge to an optimal value function with optimal policy in a finite number of iterations, since the number of policies for this MDP is finite and each policy is guaranteed to be a strict improvement over the previous one (unless it is already optimal). Policy iteration often converges in very few iterations.

2.2.2 Value Iteration

One drawback to policy iteration is that each iteration involves policy evaluation, which may itself be a protracted iterative computation requiring multiple sweeps through the state space. Value iteration algorithm, in which the policy evaluation step of policy iteration is truncated in some way without losing the convergence guarantees, begins with arbitrary V_0, and then for each iteration, given that V_{n-1} is computed, we compute the *n-stage-to-go Q-function* $Q_n(s, a)$ and *n-stage-to-go value function* V_n according to the following equations:

$$\begin{aligned} Q_n(s,a) &= R(s,a) + \gamma \sum_{t \in \mathcal{S}} T(s,a,t) V_{n-1}(t), \\ V_n(s) &= \max_{a \in \mathcal{A}} Q_n(s,a). \end{aligned}$$

As $n \to \infty$, V_n converges linearly to the optimal value function V^*. After some finite number of iterations k, the choice of maximizing action for each state s forms an optimal policy and V_k approximates the value. In fact, value iteration is obtained simply by turning the optimal Bellman equation Eq. 2 into an update rule.

2.2.3 Other Analogous Algorithms

Asynchronous dynamic programming (asynchronous DP) algorithms [5, 6] are important tools for on-line approaches to solve MDPs. These algorithms are variants of classical value or policy iteration algorithms, which do not require the value function to be constructed, or policy to be improved for each state in each iteration. As long as every state is updated sufficiently often, the convergence of the value function can be assured. It is clear that these algorithms are very useful when dealing with large state space MDPs, since they are not required to cover all state space. Puterman and Shin (1978) [74] also proposed another DP algorithm called *modified policy iteration*, which provides a middle ground between value iteration and policy iteration. The algorithm is exactly like policy iteration except that the policy evaluation step is carried out via using a small number of successive approximation steps instead of evaluating the existing policy exactly.

2.3 Classical Algorithms (II): State-based Search Methods

The DP algorithms (except the asynchronous DP) must examine the entire state space during each iteration. In this subsection, we review *decision tree search algorithms*, state-based search techniques for solving MDPs. The algorithms are advantageous when initial states or goal states are known: they restrict the attention of looking for optimal policies

for the MDPs to those states that are *reachable* from initial states or *accessible* to the goal states [2]. A detailed survey for this class of techniques can be found in [10].

2.3.1 Decision Tree Search Algorithms

We first see how a decision tree search algorithm works for finite-horizon MDPs. Given a finite-horizon MDP $< \mathcal{S}, \mathcal{A}, T, R >$ with horizon T and an initial state s_{init}, a *decision tree rooted at* s_{init} shown in following (Figure 1) is constructed as follows: each action applicable at s_{init} forms level 1 of the tree, and the resulting states s' with positive probability when applying any of those actions at s_{init} forms level 2, with an arc from action a to s' labeled with transition probability $T(s_{init}, a, s')$; the actions applicable at the states at level 2 form level 3, and so on, until the tree is grown to depth $2T$. The optimal T-stage value function and the optimal policy can be easily computed by using the decision tree. The detailed algorithm is that we label every node in the tree with values computed as follows: the value of each action node is the *expected value* of its successor states in the tree, and the value of each state node s is the sum of $R(s, a)$ and the maximum value of its successor action nodes, and the $(T - t)$-stage-to-go value $V^*_{T-t}(s)$ is the labeled value of state s reachable from s_{init} in level $2t$, and the maximizing actions along the tree form the optimal policy.

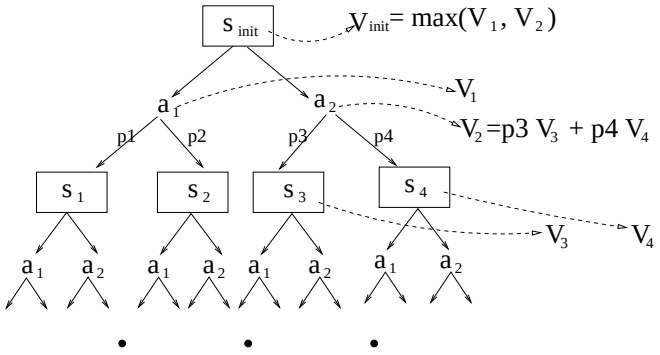


Figure 1: The initial stages of a decision tree for evaluating action choices at s_{init}.

In fact, the algorithm of computing value functions based a decision tree can be viewed as value iteration with a specific initial state.

However, there still exists difficulty in this approach. The branching factor of a decision tree for stochastic problems is generally very large. Moreover, we have to construct the whole decision tree for finite-horizon MDPs to ensure the optimal value is computed and the optimal policy is constructed. For infinite-horizon MDPs, instead of constructing a tree of fixed number of levels, we need to construct trees deep enough to ensure the optimality, but to determine whether the trees are sufficiently deep is problematic.

[2] Intuitively, the reachability of a state s means that the agent is possible to be in state s via executing some sequence of actions from the initial states; and, the accessibility of a state s' means that the agent is possible to be in some goal state via executing some sequence of actions from the state s'.

2.3.2 Real-time Dynamic Programming

Real-time dynamic programming (RTDP) [3] is one way to around above difficulties with decision trees when solving large MDP problems. Different from conventional DP mentioned above, RTDP is on-line asynchronous dynamic programming which integrates state-based search with execution of an approximately optimal policy. Similar to decision tree evaluation, suppose the agent finds itself in some particular state s_{init}. It can build a partial search tree to some depth and can be halted due to time pressure or some assessment by the agent. The *rollback procedure*, whereby values at the leaves of the tree are first computed and then values at successively higher levels are determined, is applied to this partial tree when a decision to act must be made. The key advantage is that the actual outcome of the execution of the action can be used to prune the tree branches rooted at those unrealized outcomes, and the subtrees rooted at the realized states can then be expanded further.

The rollback procedure of estimating value functions is also investigated in [24, 25, 7, 49]. The algorithm of Hansen and Zilbestein [44] can be viewed as a variant of the methods that integrate search with execution, in which stationary policies can be extracted during search process.

3 Structured Representations of MDPs

As we mentioned before, MDPs are natural models for decision-theoretic planning (DTP) in stochastic systems; but, the classical MDP framework requires explicit state and action enumeration, which grow exponentially with the number of state variables (or domain features). On the other hand, most AI problems including DTP can be viewed in terms of logical propositions, random variables, objects and relations, etc. Logical representations often can specify a system in very compact way. Hence, in recent years, many researchers (e.g., [43, 10, 46, 15]) attempt to represent and solve large state space MDPs by investigating the domain features of given stochastic systems, to represent and solve the MDPs by using high-level representation languages. In this section, we first introduce several propositional representations commonly used to model dynamic systems such as PSTRIPS, dynamic Bayesian networks, etc. Then, we concentrate on discussing the situation calculus, a first-order language that we are most interested in. They are the foundations for abstracting large state spaces of MDPs in a logical way.

3.1 Propositional Logical Representations of Dynamic Systems

In this subsection, we will discuss probabilistic STRIPS, dynamic Bayesian Networks, and some other propositional representations of dynamic stochastic systems.

3.1.1 Probabilistic STRIPS

Probabilistic STRIPS (PSTRIPS) [43, 55] is a probabilistic extension of STRIPS (the STanford Research Institute Planning System) [32]. In the PSTRIPS representation framework, the system features are represented as random variables, and the set of explicit states corresponds to the cross products of the values of all variables. The stochastic actions in the

system often affect only a small number of variables. PSTRIPS represents the dynamics in a very compact way by describing only how particular variables change under an action: each distinct outcome of any stochastic action is described by a list of consistent literals ("change list") associated with certain probability, and there may exist preconditions for different changes and probabilities.

Here is a simple example. Suppose a robot is asked to deliver coffee to some professor, in which the stochastic action is represented as $delCoff$. Several boolean variables are introduced to represent the environment: $Office$ represents whether the robot is in the professor's office, RHC represents whether the robot is holding the coffee, and CR represents whether the coffee request is outstanding. The action $delCoff$ can be represented as

Condition	Outcome	Probability
$Office$, RHC	-CR, -RHC	0.7
	-RHC	0.2
	$\emptyset$	0.1
-$Office$, RHC	-RHC	0.7
	$\emptyset$	0.3
-RHC	$\emptyset$	1.0

This table is understood as follows: under the precondition that the robot is in the professor's office and is holding coffee, then after performing the $delCoff$ action, the probability of the case that coffee request is not outstanding and the robot is not holding coffee is 0.7, the probability of the case that the robot is not holding coffee and nothing else is changed is 0.2, and the probability of nothing-changed is 0.1; under the precondition that the robot is not in the professor's office but is holding coffee, then after performing the action, the robot will not hold coffee with probability 0.7 and nothing will be changed with probability 0.3; and, under the precondition that the robot is not holding coffee, then after performing the action, nothing will be changed with probability 1.

The reward models for decision-theoretic problems can be modeled similarly. The reward models described in PSTRIPS are usually decomposed into separate aspects. For example, the following table represents a reward model for the robot-delivering-coffee example:

Aspect	Condition	Reward
Coffee	-CR	10
	CR	0

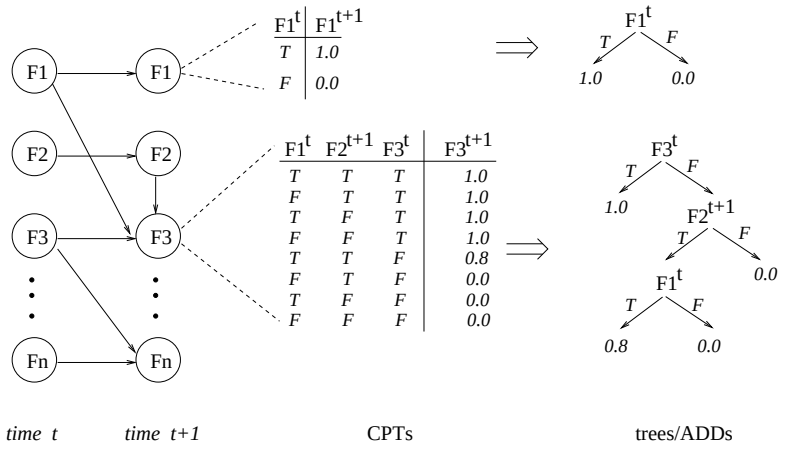
The above table represents that for aspect "Coffee", if currently coffee requesting is not outstanding, then the agent obtains 10 rewards; otherwise, it obtains 0 reward.

In an *additive* reward representation, the total reward of a state is the sum of the rewards of all aspects whose conditions are satisfied under the given state. Since the rewards often depends only on a few variable in many practical problems, PSTRIPS is generally able to represent the reward models for decision-making problems compactly.

3.1.2 Dynamic Bayesian Networks: Probabilistic Graphical Modeling

A *dynamic Bayesian network* (also known as dynamic Bayes net, DBN) [22] is a *Bayesian network* (also known as Bayes net, BN) [65] for dynamic domains, i.e., domains that change their states with time. Probability theory provides a tool of expressing randomness and uncertainty in this representation; and the graphical model of a DBN not only allows users to understand the dependency between variables easily, but also serves as a backbone for computing the marginal and conditional probabilities efficiently.

In the DBN representation for a Markovian dynamic system, the environment is also characterized by using a set of variables like PSTRIPS. One DBN is associated with each stochastic action in the system, representing the set of conditional distributions of any state at time stage $t+1$ given state in stage t. The graphical model for such a DBN (an example is shown in Figure 2.) is a directed acyclic graph, which has two 'slices': one is for time stage t, the other is for time stage $t+1$. Each slice includes nodes for all state variables. The dependency relationships of each node in stage $t+1$ on nodes of stage t and other nodes in stage $t+1$ are represented by arrows from causal nodes to the influenced node. Since this representation consists of two time stages, it is also known as *two-stage temporal Bayesian network* (2TBN). A conditional probability table (CPT) is associated with each variable node in stage $t+1$, recording the conditional probability of each value of the variable given the values of its parent nodes. The transition model for explicit states can be computed according to these CPTs.



(Assume that the variable set of the system is {F1,F2, ...,Fn}
and each variable has two values denoted as T and F)

Figure 2. A 2TBN example with two of n CPTs and trees shown

Hence, DBNs can be used to represent MDPs in a compact way.

3.1.3 Other Propositional Representations

Besides using CPTs to represent the dynamics, researchers proposed other methods, such as decision trees [75], algebraic decision diagrams (ADDs) [77] and logic rules [69], to represent transition and reward models compactly.

For example, assume that each state variable is boolean (say, its value is either T or F), instead of associating a CPT with each node in stage $t+1$ of an 2TBN for some action, an ADD for each node in stage $t+1$ is used to represent the effect of the action and the corresponding probabilities. An ADD representation of probability distribution for a node c in stage $t+1$ is an directed acyclic graph whose nodes consist of the parent variables of c in the 2TBN and leaves with numerical values representing probability values. Each edge in the graph is labeled with a value of the starting node of the edge. There is one node, say d_0, in the graph can be considered as root, since there are no directed arrows going into d_0. All numerical nodes are leaves, i.e., no edges have such nodes as starting points. Every path from the root to a leaf represents the conditional probability of $c = T$ (or $c = F$) in stage $t+1$ which equals to the value of the leaf, given the condition that along the path each node has the value on the labeled directed edge in which this node acts as the starting node. ADDs can be also used to represent the reward model in a similar way. In [12, 46], the modeling and solving MDPs based on ADDs representation were discussed in detail.

Decision trees [75] representing the conditional probability distributions in a given 2TBN for some action can be viewed as special ADDs: instead of being a directed acyclic graph, the decision tree associated with each node in time stage $t+1$ must be a tree structure. An example can be seen in Figure 2. Similarly, the decision tree structures also can be used to represent reward functions (reward trees), policies (policy trees) and value functions (value trees). For example, a policy tree is a tree with internal nodes labeled with state variables and edges labeled with variable values. The leaves of the policy tree are labeled with actions, denoting that the action to be performed at any state consistent with the labeling of the corresponding branch.

Another example, the Independent Choice Logic, proposed by Poole [69], is a propositional logic extended with possible choices to represent decision-making problems by using rules. This idea has lots in common with the stochastic situation calculus which we will see in next subsection.

3.2 The Situation Calculus: A First-order Representation

The representations we have seen above describe the state features by using variables which can be viewed as 0-ary predicates. For example, if there are several blocks in the system and we want to describe whether each block is on table or not, we may need to set binary variables for each of the blocks to represent the status. But, by using parameterized predicate, say $ontable(x)$, where the domain of x is the set of blocks, we can aggregate the representation further. Moreover, those representations are propositional, i.e., we may be unable to represent quantification properties when involving infinite domains.

On the other hand, in the area of knowledge representation, researchers have been developed systematic first-order languages, such as the situation calculus (SitCal) [59, 76], $\mathcal{A}$ calculus [34], features and fluents [78], and event calculus [53, 79], which are able to describe and reason about the dynamic systems very concisely and efficiently. Some of them are extended and used to model stochastic systems. For example, the SitCal has been extended for the efficient modeling of MDPs [76, 16, 15].

The SitCal was first proposed by J. McCarthy [59], and later improved and enriched

by other researchers such as Reiter et al. (e.g., [18, 41, 81, 29, 56, 33, 76]). It acts as a foundation for practical work in planning, control and simulation.

3.2.1 Basic Elements of the Language

The language of the SitCal [76] is a first-order language specifically designed for representing a dynamically changing world. It is a three-sorted language:

- an *action* is a first-order term representing actions an agent can take, the constant and function symbols for actions are completely application-dependent;
- a *situation* is a first-order term denoting possible world histories. S_0 denotes the *initial situation* before any action has been performed and $do(a, s)$ denotes the situation after performing action a in situation s;
- an *object* is a catch-all sort representing for everything else depending on the domain of application.

To complete the specification of the language, we require $fluents$ which are either predicates or functions whose values may vary from situation to situation. They are used to describe which properties hold in certain situations.

3.2.2 The Basic Action Theory

A *basic action theory* $\mathcal{D}$ is a set of axioms represented in the SitCal to model the actions and their effects in a given dynamic system together with *functional fluent consistency property* [33, 76]. It consists of following classes of axioms:

- Action precondition axioms $\mathcal{D}_{ap}$: for each action function $A(\vec{x})$, there is one axiom of the form $Poss(A(\vec{x}), s) \equiv \Pi_A(\vec{x}, s)$, where $\Pi_A(\vec{x}, s)$ is a formula *uniform in* s [3] representing the possible performance condition of the action in current situation.
- Successor state axioms $\mathcal{D}_{ss}$: the successor state axiom for an $(n+1)$-ary ($n \in \{0, 1, 2, \cdots\}$) relational fluent F is a sentence of $\mathcal{L}_{sc}$ of the form $F(\vec{x}, do(a, s)) \equiv \Phi_F(\vec{x}, a, s)$, where $\Phi_F(\vec{x}, a, s)$ is a formula uniform in s, and all atomic propositions in it are fluents or of form $a = a_i$ where a_i is some action. Similarly, there are also successor state axiom for functional fluents. These axioms completely characterize the value of fluents in the successor situation resulting from performing a primitive action; and, they satisfy the *Markov* property – the truth values of the fluents in next situation are dependent only on the action and the truth values of the fluents in current situation.
- Initial database $\mathcal{D}_{S_0}$: it is a set of first-order sentences in which S_0 is the only term of the situation sort (i.e., uniform in S_0). This class specifies the system properties in the initial situation and all of the non-situation related facts.

[3] Intuitively, a formula W uniform in s, where s is of sort situation, means that if there are any situation terms in W then they must all be s.

- Fundamental axioms $\mathcal{D}_f$ and unique name axioms $\mathcal{D}_{una}$ indicate several basic properties which the situations and description of the system will follow. Since these two classes of axioms are so mechanical, people normally do not write them out during description.

3.2.3 The Regression Operator

Regression is a central computational mechanism that forms the basis for many planning procedures [89] and for automated reasoning in the SitCal (e.g., [66, 33]).

The regression operator $\mathcal{R}$ is defined recursively in [76]. Roughly speaking, the regression of a formula W through an action a is a formula W' that holds prior to a being performed iff W holds after a. Successor state axioms support regression in a natural way. First, it is defined for atomic formulas as a base case by using the successor state axioms. For instance, suppose W is a relational fluent atom of form $F(\vec{t}, do(\alpha, \sigma))$ where α is of sort action and σ is of sort situation and F's successor state axiom in $\mathcal{D}_{ss}$ is $F(\vec{x}, do(a, s)) \equiv \Phi_F(\vec{x}, a, s)$[4], then

$$\mathcal{R}[F(\vec{t}, do(\alpha, \sigma))] = \mathcal{R}[\Phi_F(\vec{t}, \sigma)],$$

meaning that the regression of the fluent $F(\vec{t}, do(\alpha, \sigma))$ is equivalent to the regression of the equivalent formula obtained by using successor state axiom of F. If W is a formula uniform in S_0 then its regression equals W itself. Then, for non-atomic formulas, regression is defined inductively on the construction of the formulas.

3.2.4 Complex Actions and Golog

To deal with nondeterministic, conditional, or concurrent operations, complex actions, such as sequential actions, testing actions, conditional actions and procedures, were introduced, which results in a novel kind of high-level programming language – Golog [56].

A *Golog program* is a procedure of form: `proc` $P_1(\vec{v_1})\delta_1$ `endproc`;$\cdots$; `proc` $P_n(\vec{v_n})\delta_n$ `endproc`; δ_0, where δ_0 is the main program body, P_i is declaration of procedures with formal parameter $\vec{v_i}$ and procedure bodies δ_i for each $i(1 \leq i \leq n)$ are composed of complex actions. The semantics of a program is that when $P_1, \cdots, P_n$ are the smallest binary relations on situations that are closed under the evaluation of their procedure bodies $\delta_1, \cdots, \delta_n$, any transition obtained by evaluating the main program δ_0 is a Golog transition for the evaluation for the program. A Golog interpreter for interpreting Golog programs is implemented in Prolog.

Golog appears to offer significant advantages over current tools for applications in dynamic domains like the high-level programming of robots and software agents, process control, discrete event simulation, complex database transactions, etc. [8, 35]

3.2.5 Stochastic Actions, Probabilities and stGolog

Stochastic actions, actions with uncertain outcomes that an agent can perform, are introduced in an extended version of the SitCal, called stochastic SitCal [76]. A special predicate

$$choice(\alpha, a) \equiv a = A_1 \vee \ldots \vee a = A_n$$

[4]Without loss of generality, assume that all quantifiers (if any) of $\Phi_F(\vec{x}, a, s)$ have had their quantified variables renamed to be distinct from the free variables (if any) of $F(\vec{t}, do(\alpha, \sigma))$.

is introduced to represent any stochastic action and its nature's choices which are the possible outcomes determined by nature when executing the action, where α is a stochastic action, and $A_1, \ldots, A_n$ are the nature's choices of α. All of the nature's choices for stochastic actions are primitive deterministic actions. The action precondition axioms and successor state axioms are represented for every primitive action. Moreover, a new class of probability axioms $prob(a, \alpha, s)$ for each choice a of each stochastic action α in situation s was introduced to describe the probability of choosing a when α was executed in s. A special functional fluent $reward(s)$ was also used to express the reward functions representing the reward obtained by the agent in the situation s if we are dealing with agent decision-making problems.

Based on the above extensions, an *stGolog program* is constructed from stochastic actions together with the Golog program constructors. By using an stGolog interpreter which is modified from the Golog interpreter, the agent is able to compute the probability of satisfying given propositional formula after performing a stochastic stGolog program at the initial situation. The cost and reward of actions, exogenous events and uncertain initial situations are also formalized; and, along with descriptions of modeling sensor actions, the SitCal are able to compactly specify and reason about MDPs [16, 15], which we will see in Section 5.

4 Temporal Abstraction and Task Decomposition

The classical modeling and solution algorithms for MDPs require explicit representation of states and actions, and many of the classical solution algorithms such as value iteration and policy iteration need to explore the entire state space during each iteration. However, large DTP or learning problems usually involve a tremendous number of explicit states (and maybe actions as well) that is exponential in the number of the system's state features. For example, suppose an agent is asked to deliver mail and coffee for professors in a lab. This system may involve state features such as whether or not each professor wants coffee, whether or not there is mail for him or her, how far away each professor's office is from the coffee and mail room, and so on. If we want to enumerate all the explicit states, it will be exponentially large in the number of the above features. Thus, if we use classical approaches to deal with large MDPs, we may meet computational difficulty because of the size of the state space.

Therefore, hierarchical approaches to large MDPs have been proposed (e.g., [45, 60, 63]). For example, delivering coffee to certain offices or preparing coffee in certain lounge involves different time scales, and it is natural for one to plan or learn such knowledge at multiple levels of temporal abstraction. In this section, we go over several established approaches to temporal abstraction and task decomposition.

4.1 Options (Macro-Actions)

The first temporal abstraction approach is using *options* [85], also known as *macro-actions* [45].

4.1.1 Options

In [85], Sutton, Precup and Singh extend the usual notion of actions into a modified MDP framework that includes *options* – closed-loop policies for taking actions over a period of time. Given an MDP $< \mathcal{S}, \mathcal{A}, T, R >$, an option consists of three components: a non-deterministic policy $\pi : \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$, a termination condition $\beta : \mathcal{S} \rightarrow [0, 1]$ and an initiation set $\mathcal{I} \subseteq \mathcal{S}$. An option is available in state s_t at time t if and only if $s_t \in \mathcal{I}$. The initiation set and the termination condition of an option together restrict its range of application. If the option $< \mathcal{I}, \pi, \beta >$ is taken, then the actions are selected according to the policy π until the option terminates stochastically according to β. In particular, a *Markov option* executes as follows. First, at time stage t, the action a_t is selected according to probability distribution $\pi(s_t, \cdot)$. The environment then makes a transition to state s_{t+1}, where the option terminates with probability $\beta(s_{t+1})$, or else continues to take action a_{t+1} according to $\pi(s_{t+1}, \cdot)$ and possibly terminates in s_{t+2} according to $\beta(s_{t+2})$, and so on. The agent can select another option when the option terminates.

The primitive action $a \in \mathcal{A}$ can be considered as a special case of (*one-step*) options $< \mathcal{S}, \pi_a, \beta_a >$ where $\pi_a(s, a) = 1$ and $\beta_a(s) = 1$ for all $s \in \mathcal{S}$ meaning that a is selected everywhere and always lasts for exactly one step. Thus, any MDP model $< \mathcal{S}, \mathcal{A}, T, R >$ with a set $\mathcal{O}$ of options defined on this model can be viewed as a model with an option set $\mathcal{O}'$ which includes the options in $\mathcal{O}$ and the one-step options corresponding to primitive actions. Moreover, the Markov policy μ over options is defined as a function $\mu : \mathcal{S} \times \mathcal{O}' \rightarrow [0, 1]$.

Given options, it is required to establish a model of their consequences. Sutton et al. also prove that any MDP with a fixed set of options is a *semi-Markov decision process* (SMDP) [73], in which state transitions are stochastic, and action durations are stochastic but not time dependent. This relationship among MDPs, options and SMDPs provides a basis for a theory of planning and learning methods with options. The appropriate model for options is known from the existing theory of SMDPs. For each state in which an option may be started, this kind of model predicts the terminating state and the total reward along the way. The quantities are discounted in a particular way. The following is the state-prediction part of the model for all $s, s' \in \mathcal{S}$ and the option o starting in state s:

$$p(s, o, s') \quad = \quad \sum_{t=1}^{\infty} \gamma^t Pr(\tau = t, s_t = s' \mid s_0 = s, o), \tag{3}$$

where $Pr(\tau = t, s_t = s' \mid s_0 = s, o)$ represents the probability that the agent terminates in s' after t steps given o starting in state s. γ is the discount factor. So, $p(s, o, s')$ is a combination of the likelihood that s' is the state in which o terminates and the discount weight γ^t which decreases overtime. The reward part of the model is defined as

$$r(s, o) \quad = \quad E\{R_{t+1} + \cdots + \gamma^{k-1} R_{t+k} \mid \mathcal{E}(o, s, t)\}, \tag{4}$$

for any state $s \in \mathcal{S}$ and $o \in \mathcal{O}'$, where $\mathcal{E}(o, s, t)$ denotes the event of o being initiated in state s at time t, k is the possible steps that the agent executes when o terminates, and $R_{t'}$ is the reward obtained at any time t'.

Similar to the Bellman equation for computing the value function of a conventional MDP,

equations for $p(s,o,x)$ and $r(s,o)$ for any option o can be computed as

$$p(s,o,x) = \sum_{a\in\mathcal{A}} \pi(s,a) \sum_{s'} T(s,a,s')[(1-\beta(s'))p(s',o,x) + \beta(s')\delta_{s'x}],$$

where $\delta_{s'x} = 1$ if $s' = x$ and is 0 otherwise, and

$$r(s,o) = \sum_{a\in\mathcal{A}} \pi(s,a) \left[R(s,a) + \sum_{s'} p(s,o,s')(1-\beta(s'))r(s',o) \right].$$

Given the MDP model with an option set $\mathcal{O}$, and suppose the state-prediction and reward model for all options have been computed as above, the value function for an option policy μ can be written using the usual equation

$$V^{\mu}(s) = \sum_{o\in\mathcal{O}} \mu(s,o) \left[r(s,o) + \sum_{s'} p(s,o,s')V^{\mu}(s') \right]. \quad (5)$$

Moreover, Sutton et al. proved that $V^{\mu}(s) = V^{flat(\mu)}(s)$, where $flat(\mu)$ is the conventional policy over primitive actions (also called *flat policy*) determined by μ.

The empirical results in [85] showed that planning and learning problems can be solved more quickly by introducing and reusing proper options. Because choosing proper options, which might be already optimal in some states, can save the time of searching optimal choices of actions for these states, it may converge to a globally optimal policy faster. Moreover, Sutton et al. [85] also showed that the framework of options can be learned, used and interpreted mechanically by exhibiting simple procedures for learning and planning with options, for learning models of options, and for creating new options for sub-goals. In fact, the root of this work can be found in [86, 71, 72].

4.1.2 Macro-Actions

Macro-actions, described and used to solve MDPs hierarchically in [45], are in essence the same as options. However, this model of macro-actions is slightly different from the description of options given by Sutton et al. [85] as discussed above. In the work of Sutton et al., $p(s,o,s')$ (Eq. 3) and $r(s,o)$ (Eq. 4) are defined for all states in the state space; hence, solving the problems still relies on explicit dynamic programming over the whole state space (see Eq. 5), which does not reduce the size of state space. On the other hand, a macro-action is viewed as a local policy over a region of state space that terminates when the region is left. An *abstract MDP* is constructed consisting only of the states that lie on the boundaries of adjacent regions, and its action space only consists of macro-actions. This hierarchical MDP with macro-actions usually has a much smaller state space and action space than the original one. Therefore, it addresses the difficulties of MDPs with large state spaces and action spaces.

In their work, Hauskrecht et al. [45] base their model on a *region-based decomposition* of a given MDP $< \mathcal{S}, \mathcal{A}, T, R >$ as defined by Dean and Lin [23]: $\mathcal{S}$ is partitioned into regions

$\mathcal{S}_1, \cdots, \mathcal{S}_n$. The states lying on the boundaries of adjacent regions include the following types of states: for any region $\mathcal{S}_i$,

$$XPer(\mathcal{S}_i) = \{t \in \mathcal{S} - \mathcal{S}_i : T(s, a, t) > 0 \text{ for some } a, s \in \mathcal{S}_i\}$$

is called the *exit periphery* of $\mathcal{S}_i$, and

$$EPer(\mathcal{S}_i) = \{t \in \mathcal{S}_i : T(s, a, t) > 0 \text{ for some } a, s \in \mathcal{S} - \mathcal{S}_i\}$$

is called the *entrance periphery* of $\mathcal{S}_i$. In fact, $\bigcup_i XPer(\mathcal{S}_i) = \bigcup_i EPer(\mathcal{S}_i)$.

A macro-action for region $\mathcal{S}_i$ is a local policy $\pi_i : \mathcal{S}_i \to \mathcal{A}$. The termination condition of a macro-action is more specific than that of an option: the starting condition for π_i in time t is $s_t \in \mathcal{S}_i$ and the terminating condition in time t is $s_t \notin \mathcal{S}_i$. The discounted transition model for π_i is a mapping $T_i : \mathcal{S}_i \times XPer(\mathcal{S}_i) \to [0, 1]$ such that

$$T_i(s, \pi_i, s') = \sum_{t=1}^{\infty} \gamma^{t-1} \cdot Pr(\tau = t, s_t = s' | s_0 = s, \pi_i). \tag{6}$$

The discounted reward model for π_i is a mapping $R_i : \mathcal{S}_i \to \mathcal{R}$ such that

$$R_i(s, \pi_i) = E_\tau(\sum_{t=0}^{\tau} \gamma^t R(s_t, \pi_i(s_t)) | s_0 = s, \pi_i), \tag{7}$$

where the expectation is taken with respect to completion time τ of π_i. [5] Similarly, it is easy to deduce the Bellman equations for the discounted transition function and discounted reward function for any given macro-action π_i:

$$\begin{aligned} T_i(s, \pi_i, s') &= T(s, \pi_i(s), s') + \gamma \sum_{s'' \in \mathcal{S}_i} T(s, \pi_i(s), s'') T_i(s'', \pi_i, s'), \\ R_i(s, \pi_i) &= R(s, \pi_i(s)) + \gamma \sum_{s' \in \mathcal{S}_i} T(s, \pi_i(s), s') R_i(s', \pi_i). \end{aligned}$$

Thus, finishing the construction of the macro model for π_i is equivalent to solving the above equations.

Hauskrecht et al.[45] argued that the solution of an *augmented* MDP, the original MDP extended with a set of macro-actions, can be guaranteed to be optimal, but it can not ensure computational benefit. What they are really interested in is the *reduced* abstract MDP which is obtained by replacing primitive actions with macro-action sets and reconstructing the model on the marginal states. In detail, given an MDP $M = < \mathcal{S}, \mathcal{A}, T, R >$, a decomposition $\Pi = \{\mathcal{S}_1, \cdots, \mathcal{S}_n\}$, and a set of macros $\mathcal{A}_i = \{\pi_i^1, \cdots, \pi_i^{n_i}\}$ for each region $\mathcal{S}_i$, the abstract MDP $M' = < \mathcal{S}', \mathcal{A}', T', R' >$ induced by Π and the macro-actions is given by:

[5] Notice that the exponents of the discount factor γ differ slightly in Eq. 6 and Eq. 3, (and Eq. 7 and Eq. 4, respectively). Despite this difference, the underlying meaning of the MDP model keeps the same: later in the hierarchical MDP for macro-actions, the Bellman equation of value function over macro-actions in the abstract MDP will have the exactly format as that of primitive actions (Eq. 1), whereas the Bellman equation of value function for options (Eq. 5) is analogous and does not have the discount factor.

- $\mathcal{S}' = \cup\{EPer(\mathcal{S}_i) : i \leq n\} = \cup\{XPer(\mathcal{S}_i) : i \leq n\}$;
- $\mathcal{A}' = \cup_i \mathcal{A}_i$ with $\pi_i^k \in \mathcal{A}_i$ feasible only at state $s \in \mathcal{S}_i$;
- $T'(s, \pi_i^k, s') = T_i(s, \pi_i^k, s')$ for any $s \in EPer(\mathcal{S}_i)$ and $s' \in XPer(\mathcal{S}_i)$ and is 0 otherwise;
- and, $R'(s, \pi_i^k) = R_i(s, \pi_i^k)$ for any $s \in EPer(\mathcal{S}_i)$.

Any policy $\pi' : \mathcal{S}' \rightarrow \mathcal{A}'$ for this abstract model M' corresponds to a non-Markovian policy π for the original MDP M. Significant computational advantage can be obtained since the abstract MDP normally has a substantially smaller state space than the original one. However, due to the fact that M' allows only a limited range of macro-actions, the optimal solution π' for M' might correspond only to a sub-optimal policy π for M.

To improve the quality of the policy π, Hauskrecht et al. proposed a method to ensure the generated macro-actions offer a choice of behaviors that are of acceptable value in the sense that the difference between the optimal value of M' and the optimal value of M is no greater than a small positive number related to discount factor and the lower bound on the completion time of all macros. In fact, Parr in [62] discussed several algorithms to build up sets of local policies (i.e., *caches*) for the regions of a weakly coupled MDP. The local policies (i.e. macro-actions) in the caches can be used to find an optimal or approximately optimal policies with provable bounds for the given MDP. With one approach, called complete decoupling, and divides the problem into independent subproblems, and a cache of policies is built independently assuming that the upper and lower bounds for the values for the exit peripheral states of a region are given. In this approach, the author reviews the ϵ-grid algorithm described in [45], which devises a policy cache for any region as follows. First, an ϵ resolution grid over the space of possible values for the vector of the region's exit peripheral states is created. Then, an optimal policy with respect to each grid is produced. This approach, as Parr argues, can require a huge number of policies, $(\frac{V_{max}-V_{min}}{\epsilon})^d$, where V_{max} is the upper bound of the values of exit peripheral states, V_{min} is the lower bound of the values of exit peripheral states and d is the number of exit peripheral states of the region. Parr proposed two new algorithms for determining ϵ-optimal policy caches for any region. The first one uses a polynomial time test to decide when to add new policies to the cache, and the second one uses a computational geometry approach that can be exponential in the number of exit peripheral states d, but can be more efficient than the first one when d is small. Parr also presents a method of partial decoupling since complete decoupling is not always possible. In this approach, information is communicated among different regions. It assumes that imperfect caches have been constructed for all the regions of the given MDP with certain decompositions. A high-level decision problem is defined over the exit peripheral states of the regions. For a particular region G, this approach uses its given cache to bound the optimal values of the states in $EPer(G)$, and updates the cache for G based on the algorithm's current estimate of the values of the states in $XPer(G)$.

Although solving a single MDP problem can be sped up by using macro-actions with the above hierarchical approach, the process of generating macros and the abstract MDP model may sometimes outweigh the advantage. The essential reason for introducing the model of macro-actions is the reuse of macro-actions to solve multiple related MDPs. Multiple related MDPs are MDPs that have the same state and action space and different reward

and transition functions. The changes of these models are often local: the reward and the dynamics remain the same except for a few regions of the state space. If the macro-actions have been generated for a region such that they cover a set of different cases, they can be applied and reused in solving the revised MDPs. Hauskrecht et al. [45] proposed a hybrid MDP model for related MDPs so that one can reuse existing macro-actions. Suppose that for an original MDP M with a certain partition and macro-action set, one related MDP M_1 only changes in region $\mathcal{S}_i$. The corresponding hybrid MDP is constructed by replacing the dynamics on $EPer(S_i)$ in the abstract MDP of M with the dynamics on S_i itself in M_1. As a result, solving the hybrid MDP provides an acceptable sub-optimal (even globally optimal) solution for the revised MDP M_1, and the reuse of macro-actions offers us computational advantages.

4.2 Markov Task Decomposition

The "option" approach we discussed above can be considered as a technique from the family of *decomposition* approaches to solve large MDPs. Decomposing an MDP means that an MDP is either specified in terms of a set of "pseudo-independent" subprocesses (e.g., [82]) or automatically decomposed into such subprocesses (e.g., [9]). The solutions to these sub-MDPs are used to construct an approximate global solution. In the "option" approach, the state space of the original MDP is divided into regions to form sub-MDPs. However, there is another class of techniques which treat sub-MDPs as concurrent processes (e.g., [60]).

[60] assumes that the given MDPs, sequential stochastic resource allocation problems, are *weakly coupled*, i.e., the problems are composed of multiple tasks whose utilities are additive independent, and the actions taken with respect to a task do not influence the status of any other task. This allocation problem is modeled in a special form of MDPs *Markov task set* (MTS) of n tasks, which is a tuple $< \mathcal{S}, \mathcal{A}, R, T, c, M_g, M_l >$. $\mathcal{S}$, $\mathcal{A}$, R and T are vectors of task 1 to n's state spaces, actions spaces, reward functions and transition functions respectively; c is the cost of a single unit of the resource; M_g is the global resource constraint on the amount of the total resource; and M_l is the local resource constraint on the amount of resource that can be used for a single step. In this framework, an optimal policy is a vector of local policies maximizing the sum of rewards associated with each task. This MDP is very large, involving hundreds of tasks.

For computational feasibility, [60]'s approach sacrifices optimality. Several greedy techniques are developed to deal with variants of this problem such as no resource constraints and instantaneous constraints. The approximation strategy, called *Markov task decomposition*, is divided into two phases. In the first, *off-line phase*, value functions and optimal solutions are calculated for the individual tasks using dynamic programming. In the second, *on-line phase*, these value functions are used to compute a gradient for a heuristic search to calculate the next action as a function of the current state of all processes. The empirical results show that this method is able to deal with very large MDPs and gain computational advantage as well.

4.3 Hierarchical Abstract Machines

Learning with hierarchical abstract machines (HAMs), proposed by Parr and Russell [63], is another temporal abstraction approach dealing with large state space MDPs. Similar in purpose to options, the use of HAMs reduces the search space and provides a framework in which knowledge can be transferred across problems and component solutions can be recombined to solve larger related problems. On the other hand, in contrast to options, which can be viewed as fixed local policies, HAMs use non-deterministic finite-state controllers to express prior knowledge and therefore constrain the possible policies that the agent can choose from.

As defined in [63], a HAM is composed of a set of machine states, a transition function, and a start function that determines the initial state of the machine. Machine states are of four types: an *action* state executes an action; a *call* state executes another machine as subroutine; a *choice* state nondeterministically selects a next machine state; a *stop* state halts execution and returns control to the previous call state. The transition function stochastically determines the resulting the next machine state according to current state machine state of type action or call and the resulting environment. A HAM is in fact a program which, when executed by an agent in an environment, constrains the actions that the agent can take in each state.

To solve an MDP M with a given HAM H is to look for an optimal *HAM-consistent* policy. A policy for M that is HAM-consistent with H is a scheme for making choices whenever an agent executing H in M enters a choice state. The optimal HAM-consistent policy can be obtained by looking for optimal policies for an *induced* MDP $H \circ M$ constructed from M and H. Briefly, $H \circ M$ consists of four tuples: a state space which is a cross-product of the states of H (machine component) with the states of M (environment component), a transition function which is a combination of H's and M's transition functions for each state in $H \circ M$ whose machine component is of type action, a set of newly introduced actions that change only the machine component of any state in $H \circ M$ whose machine component is of type choice, and the reward function taken from M for each primitive action in model M and the environment component of the current state. Parr and Russell proved that if π is an optimal policy for $H \circ M$, the primitive actions specified by π constitute the optimal policy for M that is HAM-consistent with H.

The experimental results on solving an illustrative problem with 3600 states by using HAM [63] showed dramatic improvements over the standard policy iteration algorithm applied to the original MDP without HAM. Parr and Russell also applied this approach to reinforcement learning, called HAMQ-learning, by modifying Q-learning function with HAM-induced MDP model, and this HAMQ-learning appears to learn much faster than traditional Q-learning.

Andre and Russell proposed a programmable HAM (PHAM) language [1] which is an expressive agent design language that extends Parr and Russell's [63] basic HAM language with the features of the programming language LISP. This language includes standard programmatic features such as parameterized subroutines, temporary interrupts, aborts, and memory variables. For example, parameterization allows multiple actions to be combined into a single parameterized action. The ability to utilize greater modularity eases the task

of coding learning agents which take advantage of prior knowledge.

Although the programming-language features are applied in several aspects in [1] by Andre and Russell, they are mainly used for action state abstraction. To gain further computational advantages, they [2] extended the context of PHAMs further by using the state abstraction techniques from the MAXQ method [27], which will be discussed in the next subsection. The key point of state abstraction is that certain sets of states are grouped together in equivalence classes, i.e. abstract states, if they have same values in some state features. Andre and Russell proposed several types of equivalence conditions for states so that state abstraction can be applied safely[6]. They present corresponding abstracted Bellman equations for the Q-functions for the state-abstracted model. Finally, they proposed a learning algorithm for the state-abstracted MDP using PHAMs and prove that the hierarchical optimality can be assured for the original MDP if the equivalence conditions for the states are satisfied while generating abstract states.

4.4 MAXQ Methods

Building upon work by Singh [84], Kaelbling [48], Dayan and Hinton [19] and Dean and Lin [23], Dietterich [26, 28, 27] proposed an approach named MAXQ to deal with large decision-making or reinforcement learning problems. Based on the assumption that the programmer can identify useful sub-goals and subtasks that achieve these sub-goals, MAXQ decomposes the original MDP into a hierarchy of smaller MDPs and at the same time decomposes the value function of the original MDP into an additive combination of the value functions of the smaller MDPs.

Compared with options (macro-actions), in which subtasks are defined in terms of fixed local policies, and with HAMs, in which subtasks are defined by using a non-deterministic finite-state controller, MAXQ is goal-oriented and each subtask is defined by using termination predicate and a local reward function. Given an MDP $M =< \mathcal{S}, \mathcal{A}, T, R >$ and supposing the policies are mappings from the state space to the action space, MAXQ decomposes M into a finite set of subtasks $\{M_0, M_1, \cdots, M_n\}$ with the convention that M_0 is the root subtask, and solving M_0 solves the entire MDP M. Each subtask M_i is a tuple $< T_i, A_i, R_i >$. T_i is a termination predicate partitioning $\mathcal{S}$ into a set of active states, say S_i, and a set of terminal states so that M_i will be executed while in one of the active states and will halt once in one of the terminal states. A_i is a set of actions that can be performed to achieve M_i. The actions in A_i can be primitive actions from $\mathcal{A}$, or other subtasks denoted by their indexes which are larger than i. Clearly, the sets A_i $(0 \leq i \leq n)$ define a directed acyclic graph (DAG) whose root is M_0 and whose leaves are those *primitive subtasks* composed from some primitive action in $\mathcal{A}$. Finally, $R_i(s)$ is a pseudo-reward function which specifies a (deterministic) pseudo-reward for each transition to a terminal state s. Intuitively, this pseudo-reward tells how desirable each of the terminal states is for the subtask. R_i will only be used during learning, i.e., for decision-theoretic problems this component is omitted.

A hierarchical policy π for $\{M_0, M_1, \cdots, M_n\}$ is a set containing one policy for each of the subtasks: $\pi = \{\pi_0, \pi_1, \ldots, \pi_n\}$. Each subtask policy π_i takes a state and returns a primitive

[6] An abstraction is called *safe* if optimal solutions in the abstract space are also optimal in the original space.

action to execute in the state or the index of a subroutine to invoke. Given such a policy, a *projected value function* of π on subtask M_i, denoted $V^\pi(i,s)$, is the expected cumulative reward of executing π_i (and the policies of all descendants of M_i) starting in state s until M_i terminates, i.e.,

$$V^\pi(i,s) = E\{\sum_{k=0}^{\infty} \gamma^k R_{t+k} \mid s_t = s, \pi\},$$

where R_{t+k} represents the reward obtained at time $t+k$.

Like options and HAMs, MAXQ also uses a customized version of value iteration to optimize approximate subtask selection. The MAXQ method decomposes $V^\pi(0,s)$ (the projected value function of the root task) in terms of the projected value functions $V^\pi(i,s)$ of all the subtasks in the MAXQ decomposition. It has been proved [27] that $V^\pi(0,s)$ can be computed by using the decomposed equations Eq. 8, Eq. 9 and Eq. 10:

$$C^\pi(i,s,a) = \sum_{s',N} P_i^\pi(s', N \mid s, a)\gamma^N Q^\pi(i, s, \pi(s')), \tag{8}$$

$$Q^\pi(i,s,a) = V^\pi(a,s) + C^\pi(i,s,a), \tag{9}$$

$$V^\pi(i,s) = \begin{cases} \sum_{s'} T(s,i,s')R(s,i,s') & \text{if } i \text{ is primitive action} \\ Q^\pi(i,s,\pi_i(s)) & \text{otherwise.} \end{cases} \tag{10}$$

where $P_i^\pi(s', N \mid s, a)$ is the transition function of a semi-MDP determined by M_i, representing the probability that the agent ends in state s' after N steps given that a is invoked in state s. $C^\pi(i,s,a)$ represents the expected discounted cumulative reward of completing subtask M_i after invoking the subroutine for M_a in state s. Hence, the projected value function $V^\pi(i,s)$ for any i is either the expected value of performing i in the state s if i is a primitive action, or otherwise it is same as the expected cumulative reward for subtask M_i of performing action $\pi_i(s)$ in state s, and then following hierarchical policy π until subtask M_i terminates, i.e., $Q^\pi(i,s,\pi_i(s))$.

Dietterich argued that given any MAXQ decomposition, similar to the methods of options [85] or hierarchical abstract machines (HAMs) [63], there is an algorithm to obtain the hierarchical optimal policy achieving the highest expected reward among all policies consistent with the decomposition. More important, an online learning algorithm, MAXQ-Q, has been presented and proved that it converges with probability 1 to a kind of locally-optimal policy known as *recursively optimal policy*. A policy $\pi' = \{\pi'_0, \cdots, \pi'_n\}$ is recursively optimal means that each π'_i is optimal on the local MDP which has state space S_i, action space A_i, transition function $P_i^\pi(s', N \mid s, a)$ for $s, s' \in S_i$ and $a \in A_i$, and reward function given by the sum over $R(s,a,s')$ and pseudo-reward $R_i(s')$. The reason to seek a recursively optimal policy, which is weaker than hierarchical optimal policy and global optimal policy, is that this context-free property makes it easier to share and reuse subtasks.

A state abstraction method is introduced into MAXQ method by illustrating with some examples: suppose each state in the model can be represented as a vector of values of state variables, and for each subtask the state variable set is partitioned into two parts, the

full MDP is represented in a compact way by using state variables associated with a few values, and the MAXQ representation gave a much more compact representation of the value function which is decomposed into a sum of terms such that no single term depends on entire state space of the MDP. Therefore, by using such a state abstraction technique, the MAXQ method not only represents a large MDP compactly, but also gains computational advantage in computing the decomposed value function.

4.5 Discussion and Summary

In this section, we reviewed several well-known temporal abstraction and task decomposition approaches for dealing with large state space MDPs. These methods have their own advantages and disadvantages. The main advantage of using options (macro-actions) is that the models and properties of options or macro-actions can easily be established on local regions, which generally have smaller state spaces than the one of the original MDP, and can later be reused. The disadvantage of using options is that these local policies are fixed. Thus, to obtain acceptable approximate optimal policies for the original MDP, we have to look for suitable sets of macros for each region, which can be difficult. HAMs and MAXQ, on the other hand, are not fixed. MAXQ defines subtasks in terms of termination predicates, which require the programmer to guess the relative desirability of the different states in which the subtasks might terminate. This can also be difficult. The advantage of MAXQ is that it hierarchically decomposes the large state space MDP into multi-layered subtasks, and the policies learned in lower levels of the subtasks can be shared for multiple parent tasks. Moreover, MAXQ uses a state abstraction technique to eliminate irrelevant aspects in the subtask state spaces and decomposes the value function for the given problem into a set of sub-functions for the subproblems, which therefore can obtain computational advantage. HAMs restrict the large number of possible policies to be considered for computing optimal or nearly-optimal policies by using non-deterministic controllers, therefore achieving computational advantages. But, to compute an optimal policy that is HAM-consistent, we need to construct an induced MDP which has an even larger state space. PHAM, programmable HAM language, also adopts state abstraction techniques similar to those in the MAXQ method to gain further computational advantages.

5 State Space Abstraction for MDPs by Using Logical Representations

Unlike the classical MDP framework, which requires an explicit representation of states and actions, several recently proposed MDP representations, such as factored MDPs, first-order MDPs, relational MDPs and MDPs represented by the Integrated Bayesian Agent Language (IBAL), describe state spaces (and action spaces) based on certain properties or features of the system rather than to enumerate them explicitly. The state features are characterized by a number of variables or fluents, and the corresponding explicit state space for the conventional MDP can be viewed as the cross products of the domains of each feature or instantiated fluents, which therefore grows exponentially in the number of

features or instantiated fluents. The benefit is obvious: such an intensional representation is much more convenient and more compact. By exploiting the logical structure of the MDPs and describing the effects of actions in a logical fashion, with DBNs, decision trees, ADDs, PSTRIPS or basic action theories, we may also gain computational advantages.

5.1 Factored MDPs

A survey by Boutilier et al. [10] gives a detailed description and discussion of *factored MDPs* (FMDPs). An FMDP framework represents the state space of a decision-theoretic problem by describing the features of the system. It can be considered as a type of propositional representation framework. Each feature has a set of values, i.e., a domain. Based on factored representation of state space, *factored representations* of actions, rewards and other components are also adopted, which often provide representational economy. There are several ways to represent the factored actions affecting state features which we have reviewed in Section 3: one uses 2TBNs associated with decision-tree or decision-graph representations of CPTs (e.g., [14, 46]); another uses PSTRIPS (e.g. [42, 43, 55]). Similarly, simplified formulations of the reward function for an FMDP often can be represented in a factored way depending on the fact that the reward associated with a state often relies only on the values of certain features of the state. Several ways of formulating the factored reward functions were proposed by using decision trees or graphs [12], STRIPS-like tables [11], or logical rules [69, 70]. Another way of representing a reward function is to assume the reward function is equivalent to the sum of individual contributions, which is based on the assumption that the components make independent contributions to the total reward. Finally, policies and value functions for FMDP frameworks can be represented compactly by using decision trees [12] or STRIPS-like representation [80].

Algorithms for solving or approximately solving FMDPs (e.g., [46, 13, 51, 39, 64, 40, 17]) have been widely studied and used to solve fairly substantial problems. For instance, the SPUDD algorithm [46] using DBNs with ADDs to represent MDPs has been used to solve MDPs with hundreds of millions of state optimally, producing logical descriptions of value functions that involve only hundreds of distinct values. In [13], Boutilier, Dearden and Goldszmidt modeled MDPs using DBNs with decision-tree representations of rewards, and developed modified versions of dynamic programming algorithms that directly manipulate decision-tree representations of policies and value functions, which generally obviated the need for state-by-state computation. In [40], FMDPs were applied to multiagent coordination and planning problems, which scaled very complex problems and were solved efficiently.

5.2 First-Order MDPs

First-order MDPs (FOMDPs) are the underlying models of decision-theoretic Golog (DTGolog) [16] described using the SitCal extended with decision-theoretic aspects (the stochastic SitCal). And in [15], a symbolic dynamic programming approach for solving FOMDPs was also proposed.

5.2.1 Decision-theoretic Golog

Decision-theoretic Golog (DTGolog) proposed by Boutilier et al. [16] is a high-level programming framework which allows the seamless integration of agent programming with decision-theoretic planning (DTP). This programming language uses the stochastic SitCal to model any DTP problem: the modeled system includes axioms of nature's choices of each stochastic action in the system, the basic action theory for all nature's choices, the *prob* axioms representing probabilities for nature's choices to corresponding stochastic actions, and *reward* axioms asserting rewards as a function of nature's choices, properties of current situation, or both.

The underlying model of the above described system in the SitCal is a first-order MDP. The fluents of the system defined for given problems represent state features of the system, and any ground state in the MDP model can be viewed as combinations of the values of the instantiated fluents in the system. The action space of the classical MDP is the collection of all instantiated stochastic actions. The transition model is captured by choice axioms, the basic action theory, and the probability axioms; and, the reward model is represented by reward axioms. The reason that the underlying MDP is first-order is that the states and actions are aggregated by using parameterized fluents and parameterized action functions. And representing and reasoning with the axioms of the system is first-order. Using first-order logic makes it possible to represent the existence of unspecified objects by using quantification.

Finite horizon MDPs are considered in DTGolog; and its optimality criterion achieves non-discounted maximal expected reward. DTGolog programs are written using the same complex action operators as Golog programs. The semantics of a DTGolog is defined with a predicate $BestDo(prog, s, h, pol, val, prob)$ that returns the expected value val of an optimal policy pol with the probability $prob$ of the success of executing pol, given the DTGolog program $prog$ starting in the situation s and terminating before reaching h steps. An interpreter for $BestDo$ is defined recursively on the structure of the given program $prog$ and the horizon h, and is implemented in Prolog.

This work provided a way of specifying DTP problems which have tremendously large ground state and action spaces concisely and imposed constraints on the space of allowable policies by writing a program. It provided a natural framework for combining DTP and agent programming with an intuitive semantics.

5.2.2 Symbolic Dynamic Programming for First-Order MDPs

Boutilier, Reiter and Price [15] proposed a dynamic programming approach for solving FOMDPs by using a technique called *first-order decision-theoretic regression* (FODTR). Similar to DTGolog's system framework, FOMDPs are represented in the SitCal extended with stochastic actions. It produces a logical representation of the optimal policy and expected value function by constructing a set of first-order formulas that (minimally) partition state space according to distinctions made by the value function and policy.

The decision-theoretic regression (DTR) operator working with propositional representation of MDPs was fully discussed in [13]. As we discussed above, the advantage of first-order representation is obvious: it can easily specify and reason with domain properties which

become problematic in propositional setting. Therefore, they [15] would like to extend DTR to be able to deal with FOMDPs. In detail, suppose we have an FOMDP described in the SitCal. For any probability function or reward function, its value can be represented as a minimal partition of state space by using finitely many first-order partition conditions in a special case-format: $t = case[\phi_1, t_1; \cdots; \phi_n, t_n]$, which is an abbreviation for $\bigvee_{i=1}^{n} \phi_i \wedge t = t_i$. Moreover, assume the value function can also be represented in the form of $V(s) = case[\beta_1(s), v_1; \cdots; \beta_n(s), v_n]$, where partition conditions $\beta_i(s)$ partition the state space in situation s, then the FODTR for a given value function V through stochastic action $A(\vec{x})$ is a logical description of the Q-function $Q^V(A(\vec{x}), s)$: it equals to the sum of the reward $R(s)$ in the situation s and the discounted cumulative value after performing $A(\vec{x})$, i.e.

$$Q^V(A(\vec{x}), s) \quad = \quad R(s) + \gamma \sum_j prob(n_j(\vec{x}), A(\vec{x}), s) \cdot V(do(n_j(x), s)), \tag{11}$$

where s is of sort situation and $prob(n_j(\vec{x}), A_(\vec{x}), s)$ is the probability with which nature chooses $n_j(x)$ when executing $A_i(x)$ in s. To complete the construction of FODTR of the Q-value function, the following manipulation is performed. By asserting the the case-format represented formulas for $R(s)$, $prob(n_j(\vec{x}), A_(\vec{x}), s)$ and $V(do(n_j(x), s))$, applying regression operator and performing sums and products of case statements, the formula will also result in case statements uniform in s, i.e. the situations appearing in the partition conditions of the resulting formulas are all s. By using FODTR, the state and action enumeration in dynamic programming is avoided, and the Q-value function $Q(a, s)$ for each situation s and each instantiation a of $A(\vec{x})$ is captured with very small number of formulas.

Suppose we have computed the n-stage-to-go Q-function $Q(a, s)$; then, the n-stage-to-go value function can be symbolically represented as

$$V(s) \quad = \quad (\exists a).(\forall b).Q(b, s) \leq Q(a, s), \tag{12}$$

meaning that $V(s)$ is the maximal $Q(a, s)$ with respect to action a. This formula can also be represented as a case statement if the Q-function is of case format. It is proved that Eq. 12 together with Eq. 11 is a correct logical formalization for value function $V(s)$ relative to the specification of classical value iteration algorithm introduced in Section 2. Therefore, this is a method that can compute the exact value functions for FOMDPs rather than a approximation approach.

5.3 Relational MDPs

The *relational MDP* (RMDP) model is another state aggregation model [38] based on the probabilistic relational model framework [52]. RMDPs use state variables and links, which are similar to fluents in the FOMDPs, to capture the state features. For action representation, RMDPs define action schemata, which are similar to parameterized action terms in FOMDPs, to represent different types of actions. RMDPs also have lots of different aspects from FOMDPs. First, RMDPs are *class-based*, i.e., they partition the instant objects in the world into classes, restrict each parameter of the state variables, links and action schemata to suitable classes. Second, FOMDPs represent the transition model through the probability

of nature's choices of action schemata and the basic action theory, hence the number of axioms is linear to the number of fluent and action schemata; while, the transition functions of RMDPs associate with classes and define distribution over the next state of an object in the classes, given the current status of the object, the action taken and the states and actions of all the objects linked to the object. The transition model of an RMDP is the same for all instances in the same class. Finally, FOMDPs use partition conditions which partition the state space into finitely many classes and agents obtain different rewards according these conditions; on the other hand, any RMDP given by Guestrin et al. defines the reward model based on the assumption that the reward obtained at each state after performing certain action schema is the sum over the rewards obtained by all instances that perform the instantiated action at the instantiated starting state. It is argued [38] that for any RMDP there is a corresponding ground *factored* MDP, whose transition model is specified as a DBN. So, from this point of view, the RMDP framework is propositional when all the given classes are finite.

In [38] based on the assumption that the value function can be well-approximated as a sum of local value sub-functions associated with the individual objects in the model, the optimal solution of the approximated value function of the given RMDPs is computed by using a linear programming approach with some reasonable number of *sampled worlds*. [7]

Other studies on solving RMDPs were proposed as well. For example, Weld et al.[58] intended to solve RMDPs by using inductive techniques based on regression trees [54] learning from the solution of propositional MDPs obtained by extending given RMDPs with a small number of objects.

5.4 Integrated Bayesian Agent Language

The *Integrated Bayesian Agent Language* (IBAL) developed by Pfeffer [67] is a declarative programming language for probabilistic modeling, parameter estimation and decision making. It can model MDPs in a concise way like DTGolog. But, they are quite different in the sense that DTGolog is a logic programming framework and its underlying models are FOMDPs, while IBAL is a functional programming framework with its underlying roots in Bayesian networks.

The top level language component in IBAL is a *block*, which includes a sequence of *declarations*. There are several kinds of declarations such as *definitions*, *rewards*, and *decision* declarations, etc [8]. *Definitions* state how values of things are stochastically generated. A

[7] A *world* specifies the set of objects for each class. The reason that sampling worlds are needed is that the size of the linear program formulation of the value function, the size of the objective and number of constraints, grows with the numbers of possible worlds which generally grows exponentially with the number of possible objects, or may even infinite. Moreover, according to the fact that different objects in the same class behave similarly: they share the transition model and reward function, it is practical to generalize the results of the local value sub-functions of sampled worlds to the ones that are not sampled.

[8] There are other types of declarations including *observation*, *prior* and *pragmatics*. *Observations* state that some property holds of generated values for variables, and models like Bayesian networks and relational probability models can be expressed by using observation declarations. *Priors* describe the prior probability distributions over learnable parameters; an IBAL program with prior declarations specifies a joint model, which defines a joint probability distribution over the parameters, and the observation conditions of the model in a standard Bayesian way. *Pragmatics* contain control knowledge on how to perform inference in a

definition has the form of $x = e$, which means that expression e defines a stochastic experiment generating the value of x. For example, `fair()= dist`$[0.5 :' h, 0.5 :' t]$ represents that $fair$ is a function that return either $'h$ with probability 0.5, or $'t$ with probability 0.5. [9] *Rewards* describe the rewards an agent receives, which are used in modeling decision-theoretic problems. It is either of form `reward case` e `of` $[\pi_1 : r_1, \cdots, \pi_k : r_k]$ or `reward` $\gamma\ x$. The first form means that the reward depends on the value of e, and it is the real number r_i associated with the first pattern π_i that the value satisfies. The second states that the reward is γ times the value of x. To represent MDPs, IBAL introduces *decision declarations* as well, which describe the range (e.g., action instances) of decision variables (e.g., action schemata) given available informations (e.g., current state). For example, a typical MDP is schematically represented as follows:

```
MDP(s)={                 /* takes current state as argument */
   transition(s,a)=...  /* returns next state with certain prob. distribution */
   reward(s,a) ={ reward case(s,a) of [('s1,'a1): r1,...]
   choose a from a1,a2,...  given s
   next_state = transition(s,a)
   current_reward = reward(s,a)
   future_reward = MDP(next_state)
   reward 1 current_reward
   reward 0.9 future_reward }.
```

Each block including decision declarations and reward declarations constitutes on distinct decision problems, and state variables are the only free variables. An expected utility function is a mapping from states to real numbers, which equals to the sum over reward declarations given a corresponding state instance. Since the reward declarations in the block are defined in a recursive format and include the discount factor for future reward, the sum over all the reward declaration ensures the correctness of expected utility, a longterm cumulative value. Hence, solving the decision problem is same as maximizing the expected utility.

IBAL is indebted to several relate work integrating probabilistic and first-order representation languages [50, 68, 61]. For instance, a stochastic functional language proposed by Koller et al. [50] can be considered as the most direct precursor to IBAL. It is a Lisp-like language extended with a $flip$ construct to describe random events. IBAL's basic definition and expression language is similar to the one in [50], but it is richer: IBAL uses higher-order functions and integrates decisions and learning into the framework.

5.5 Discussion and Summary

Representing MDPs as logical structures is feasible and reasonable. Comparing among these representations, they clearly have different aspects, some are advantageous, and others are disadvantageous. First, the FMDP model is the earliest and best studied model among

block. Since they are useful for describing other probabilistic models, but rarely used in describing decision problems modeled in MDPs, we omit the details of these declarations here.

[9] Besides the form `dist`$[p_1 : e_1, \cdots, p_n : e_n]$, expression e in the definition $x = e$ can have other forms such as block b, constant $'o$, conditional `if-then-else`, etc, which make IBAL more expressive. The details are in [67].

all the logical representations of MDPs. The representation of state spaces for FMDPs is very simple. By using state features (variables), and by using DBNs to represent the transition model and the reward model, FMDPs achieve a much more compact and more intuitive representation than traditional MDPs, because they take advantage of conditional independency between state and action variables. Moreover, the algorithms developed upon such representations are able to gain a great computational benefit, since they make use of the structured representations. On the other hand, FOMDPs, RMDPs and IBAL are more expressive. For instance, the representations of FOMDPs not only are very simple and compact, but are also powerful enough to represent even infinite state spaces by using first-order expressions. FOMDPs are represented based on logical models of the decision problems, which are more dynamic and adaptable to other worlds in the sense that the domains of the fluents in the system are easily changeable without destroying the underlying relational representations of the transition model and reward model. RMDPs are also able to express parameterized state features, and the description of transition models for each action is based on the conditional dependencies among such parameterized state features for classes of objects. Therefore, the models represented as FOMDPs or RMDPs are more compact and more reusable than FMDP models. Compared to FOMDPs, RMDPs and IBAL express transition functions more directly, since in FOMDPs programmers not only need to figure out the exact nature's choices (deterministic outcomes) of the stochastic actions, but also need to describe the exact logical effect of the deterministic outcomes on fluents through basic action theory. The transition probability from one state to another after performing certain action described in the conventional MDPs is therefore not so obvious. However, the syntax of the representation of FOMDPs is simpler, and the basic action theory makes it easier for users to understand the dynamics of the system comparing to RMDPs and IBAL.

6 Introducing Temporal Abstraction into Logically Represented MDPs

As we have seen, various approaches to tackle MDPs with large state spaces have been proposed. The approaches of using logical representations are only at the beginning (almost all the work was done in the past few years), especially for first-order representations of MDPs. In the case of temporal abstraction approaches, there are still open problems. For instance, most temporal abstraction methods need to sacrifice optimality in exchange for computational advantages. Although this sacrifice might be inevitable, we may be able to minimize it by developing new techniques such as learning suitable sets of reusable options or discovering standard conditions to ensure a reasonable subtask decomposition for given MDPs. For example, like the work in [62], one might think of developing new algorithms to look for acceptable caches which are able to ensure near-optimality when they are used in higher level abstract MDPs.

One more tentative direction that we are most interested in and focus on currently is this: It seems to us that introducing temporal abstraction approaches into logic-represented MDPs would further reduce their computational complexity. For instance, in an FOMDP represented in the SitCal, we may consider an stGolog program as a general local policy for

some local FOMDP, which can be viewed as a compact representation of a class of explicit macro-actions for instances that share the same characteristics as the transition and reward models. Therefore, it can be easily reused not only while entering the exact local region for one instance, but also in the similar regions for other similar instances. Here is an intuitive example. Suppose we know the optimal policy $travel(A, B)$ traveling from person A's house to person B's house. For any variable x of sort person living with A and variable y of sort person living with B, this implies that x and A (y and B, respectively) live in the same place and therefore the transition model and reward model of traveling from x's home to y' home is the same as those of traveling from A's home to B' home. We can then naturally apply policy $travel(x, y)$ when we want to go to y's home from x's home. Doing so agrees to natural thinking ways of human beings.

In fact, some primitive work (e.g., [27, 1, 37, 31]) has been proposed in combining logical representations with temporal abstraction and task decomposition methods. In the next subsection we take a look at the prior work, and then we briefly discuss some general idea of our future work.

6.1 Prior Work

We have seen that the MAXQ approach [27] and PHAMs method [1] have used factored state spaces in their representations of MDPs to gain ease of representation and speed up computation. This could be considered as the earliest attempting of joining temporal abstraction with logical representations.

Guestrin and Gordon [37] presented a principled planning algorithm for collaborative multiagent dynamical systems. Like large weakly coupled MDPs, each agent only needs to model and plan for a small part of the system. These subsystems are connected through a subsystem hierarchy. They modeled this system by using FMDPs and computed the local reward functions by local linear programs. When two portions of the hierarchy share the same structure, the algorithm was able to reuse local optimal plans and messages to gain a computational advantage.

Other work given by Ferrein, Fritz and Lakemeyer [31] extended DTGolog with options defined in [85]. In their work, the underlying MDP of DTGolog is considered in factored representation, and an option is defined as a tuple $< I, \pi, \beta >$, where I is a finite set of possible (explicit) initial states $\{s_1, s_2, \cdots, s_n\}$, and β is the set of possible (explicit) terminating states. It is easy to see that the local MDP for the region I and β is classical, and it was argued that the classical DP algorithm with value iteration can be applied to obtain the local optimal policy π which later is represented in the form of a while-loop:

```
senseState(O);
   while (⋁ φ_i)
    if φ_1 then A_1 else if ... else if φ_n then A_n;
    senseState(O);
   endwhile
```

where each ϕ_i is a condition that can uniquely determine s_i and the sense action `senseState(`O`)` detects the values of ϕ_i's, so that they can be tested later. The local transition function given π can be also obtained by the value iteration algorithm. In [31], the options are given by the

programmer. Then, this option $< I, \pi, \beta >$ is treated like a primitive action and is reused. When several options are generated and reused in the next round of decision making in the same DTP setting, the overall computational effort is reduced greatly.

6.2 Our Future Directions

In the approaches mentioned above, the MDPs are treated in a factored way, and there is no attempt to use the power of first-order representations. In [31], although the presentation of the system uses the situation calculus and DTGolog, the underlying MDPs of the local regions are still explicit, and need to be solved explicitly; moreover, even though the system is able to reuse the options when an agent entered the exact region, it is unable to reuse them in related MDPs.

We intend to investigate a broader type of macro-actions, which can be any small stGolog program. Since such macro-actions are parameterized and first-order, they could be reused in related MDPs to gain further computational benefit for us. In [36], a small finite sequence of parameterized stochastic actions is considered as a macro-action, and an extended basic action theory is automatically generalized for the given macro-actions. When the macro-actions are reused in stGolog, they are treated as a whole and the extended axioms are applied, therefore sped up the computation slightly. This was our first attempt of using and reusing macro-actions. It has the shortage that the style of the macro-actions is very limited, the knowledge base including the extended basic action theory might be oversized, and the reasoning is dependent on the exact nature's choices of the macro-actions. What we are really interested in, however, are only the discounted probabilities and discounted rewards after executing the macro-actions. To overcome these problems, inspired by the work in [45], our future work attempts to deal with general types of macro-actions (any stGolog programs) and analyze the local FOMDPs that might be affected by the macro-actions. We will also develop new techniques to obtain the discounted transition function and discounted reward function similar to Eq. 6 and Eq. 7 for each macro-action on its local FOMDP region by using a finite state machine determined by the program. Finally, we plan to reuse them in large related MDPs represented in first-order languages.

There may be further work we can think of in this direction. For example, above macro-actions we mentioned are assumed to be given by the programmer. Like [45], we would like to ask the question about what kinds of macro-actions can be considered as acceptable ones, whether it is possible to generate or learn acceptable macro-actions automatically and how, etc.

References

[1] David Andre and Stuart J. Russell. Programmable reinforcement learning agents. In Todd K. Leen, Thomas G. Dietterich, and Volker Tresp, editors, *Advances in Neural Information Processing Systems 13*, pages 1019–1025. MIT Press, 2001.

[2] David Andre and Stuart J. Russell. State abstraction for programmable reinforcement learning agents. In *Proceedings of the Eighteenth National Conference on Artificial*

Intelligence and Fourteenth Conference on Innovative Applications of Artificial Intelligence (AAAI/IAAI-02), pages 119–125, Edmonton, Alberta, Canada, July 2002.

[3] Andrew G. Barto, Steven J. Bradtke, and Satinder P. Singh. Learning to act using real-time dynamic programming. *Journal of Artificial Intelligence*, 72(1-2):81–138, 1995.

[4] Richard Bellman. *Dynamic Programming*. Princeton Univ. Press, Princeton, NJ, 1957.

[5] Dimitri P. Bertsekas. *Dynamic Programming: Deterministic and Stochastic Models*. Prentice-Hall, Englewood Cliffs, NJ, 1987.

[6] Dimitri P. Bertsekas and John N. Tsitsiklis. *Neuro-Dynamic Programming*. Athena Scientific, Belmont, MA, 1996.

[7] Blai Bonet, Gabor Loerincs, and Hector Geffner. A robust and fast action selection mechanism for planning. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI-97)*, pages 714–719, Providence, Rhode Island, 1997. AAAI Press / MIT Press.

[8] Anthony J. Bonner and Michael Kifer. Transaction logic programming. Technical report, Dept. of Computer Science, University of Toronto, 1993.

[9] Craig Boutilier, Ronen Brafman, and Christopher Geib. Prioritized goal decomposition of Markov decision processes: Towards a synthesis of classical and decision theoretic planning. In Martha Pollack, editor, *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pages 1156–1163, San Francisco, 1997. Morgan Kaufmann.

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

[11] Craig Boutilier and Richard Dearden. Using abstractions for decision-theoretic planning with time constraints. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, volume 2, pages 1016–1022, Seattle, Washington, USA, August 1994. AAAI Press/MIT Press.

[12] Craig Boutilier, Richard Dearden, and Moises Goldszmidt. Exploiting structure in policy construction. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (IJCAI-95)*, pages 1104–1111, Montreal, Canada, 1995.

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

[14] Craig Boutilier and Martin L. Puterman. Process-oriented planning and average-reward optimality. In *IJCAI*, pages 1096–1103, 1995.

[15] Craig Boutilier, Ray Reiter, and Bob Price. Symbolic dynamic programming for solving first-order Markov decision processes. In G. Lakemeyer, editor, *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence*, pages 690–970, Seattle,Washington, 2001.

[16] Craig Boutilier, Raymond Reiter, Mikhail Soutchanski, and Sebastian Thrun. Decision-theoretic, high-level agent programming in the situation calculus. In *Proceedings of Seventeenth National Conference on Artificial Intelligence*, pages 355–362, 2000.

[17] Ronald Parr Carlos Guestrin, Daphne Koller and Shobha Venkataraman. Efficient solution algorithms for factored MDPs. *Artificial Intelligence Research (JAIR)*, 19:399–468, 2003.

[18] Ernest Davis. *Representations of Commonsense Knowledge*. Morgan Kaufmann Publishers, San Francisco, CA, 1990.

[19] Peter Dayan and Geoffrey E. Hinton. Feudal reinforcement learning. In C. L. Giles, S. J. Hanson, and J. D. Cowan, editors, *Advances in Neural Information Processing Systems 5, Proceedings of the IEEE Conference in Denver (to appear)*, San Mateo, CA, 1993. Morgan Kaufmann.

[20] Thomas Dean and Robert Givan. Model minimization in Markov decision processes. In *AAAI/IAAI*, pages 106–111, 1997.

[21] Thomas Dean, Leslie Pack Kaelbling, Jak Kirman, and Ann Nicholson. Planning with deadline in stochastic domains. In *Proceedings of Eleventh National Conference on Artificial Intelligence*, pages 574–579, 1993.

[22] Thomas Dean and Keiji Kanazawa. A model for reasoning about persistence and causation. *Computational Intelligence*, 5:142–150, 1989.

[23] Thomas Dean and Shieu-Hong Lin. Decomposition techniques for planning in stochastic domains. *IJCAI-95*, pages 1121–1127, 1995.

[24] Richard Dearden and Craig Boutilier. Integrating planning and execution in stochastic domains. In *Proceedings of the AAAI Spring Symposium on Decision Theoretic Planning*, pages 55–61, Stanford, CA, 1994.

[25] Richard Dearden and Craig Boutilier. Abstraction and approximate decision-theoretic planning. *Artificial Intelligence*, 89(1–2):219–283, 1997.

[26] Thomas G. Dietterich. The MAXQ method for hierarchical reinforcement learning. In *Proc. 15th International Conf. on Machine Learning*, pages 118–126. Morgan Kaufmann, San Francisco, CA, 1998.

[27] Thomas G. Dietterich. Hierarchical reinforcement learning with the MAXQ value function decomposition. *Journal of Artificial Intelligence Research*, 13:227–303, 2000.

[28] Thomas G. Dietterich. State abstraction in MAXQ hierarchical reinforcement learning. In *Advances in Neural Information Processing Systems 12*, pages 994–1000, 2000.

[29] Charles Elkan. Reasoning about action in first-order logic. In *Proceedings of the Ninth Biannual Conf. of the Canadian Society for Computational Studies of Intelligence (CSCSI'92)*, pages 221–227, San Francisco, CA, 1992. Morgan Kaufmann Publishers.

[30] Zhengzhu Feng and Eric A. Hansen. Symbolic heuristic search for factored Markov decision processes. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence (AAAI-02)*, Edmonton, Alberta, Canada, July 2002.

[31] Alexander Ferrein, Christian Fritz, and Gerhard Lakemeyer. Extending DTGolog with options. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence*, pages 1394–1395, Acapulco, Mexico, August 2003.

[32] Richard Fikes and Nils J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3-4):189–208, 1971.

[33] Ray Reiter Fiora Pirri. Some contributions to the metatheory of the situation calculus. *Journal of the ACM*, 46(3):325–364, 1999.

[34] Michael Gelfond and Vladimir Lifschitz. Representing action and change by logic programs. *Journal of Logic Programming*, 17:301–322, 1993.

[35] Giuseppe De Giacomo, Yves Lespérance, and Hector J. Levesque. Reasoning about concurrent execution, prioritized interrupts, and exogenous actions in the situation calculus. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pages 1221–1226, 1997.

[36] Yilan Gu. Macro-actions in the situation calculus. In *Proceedings of IJCAI'03 Workshop on Nonmonotonic Reasoning, Action, and Change (NRAC'03)*, Acapulco, Mexico, August 2003.

[37] Carlos Guestrin and Geoffrey J. Gordon. Distributed planning in hierarchical factored MDPs. In *Proceedings of the Eighteenth Annual Conference in Uncertainty in Artificial Intelligence (UAI-02)*, pages 197–206, 2002.

[38] Carlos Guestrin, Daphne Koller, Chris Gearhart, and Neal Kanodia. Generalizing plans to new environments in relational MDPs. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence(IJCAI-03)*, pages 1003–1010, Acapulco, Mexico, August 2003.

[39] Carlos Guestrin, Daphne Koller, and Ronald Parr. Max-norm projections for factored MDPs. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence(IJCAI-01)*, pages 673–682, 2001.

[40] Carlos Guestrin, Shobha Venkataraman, and Daphne Koller. Context specific multiagent coordination and planning with factored MDPs. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence and Fourteenth Conference on Innovative Applications of Artificial Intelligence (AAAI/IAAI-02)*, pages 253 – 259, Edmonton, Alberta, Canada, July 2002.

[41] Andrew R. Haas. The frame problem in artificial intelligence. In F.M. Brown, editor, *Proceedings of the 1987 workshop*, pages 343–348, San Francisco, CA, Los Altos, California, 1987. Morgan Kaufmann Publishers.

[42] Steve Hanks. *Projecting Plans for Uncertain Worlds*. PhD thesis, Yale University, January 1990.

[43] Steve Hanks. Modeling a dynamic and uncertain world I: Symbolic and probabilistic reasoning about change. *Artificial Intelligence*, 66(1):1–55, 1994.

[44] Eric A. Hansen and Shlomo Zilberstein. Heuristic search in cyclic AND/OR graphs. In *AAAI/IAAI*, pages 412–418, 1998.

[45] Milos Hauskrecht, Nicolas Meuleau, Leslie Pack Kaelbling, Thomas Dean, and Craig Boutilier. Hierarchical solution of Markov decision processes using macro-actions. In *Proceedings of the Fourteenth Conference on Uncertainty In Artificial Intelligence*, pages 220–229, San Francisco, 1998. Morgan Kaufmann.

[46] Jesse Hoey, Robert St-Aubin, Alan Hu, and Craig Boutilier. SPUDD: Stochastic planning using decision diagrams. In *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence*, pages 279–288, Stockholm, 1999.

[47] Ronald A. Howard. *Dynamic Programming and Markov Processes*. MIT Press, 1960.

[48] Leslie P. Kaelbling. Hierarchical learning in stochastic domains: Preliminary results. In *International Conference on Machine Learning*, pages 167–173, 1993.

[49] Sven Koenig and Reid G. Simmons. Real-time search in non-deterministic domains. In *IJCAI*, pages 1660–1669, 1995.

[50] Daphne Koller, David A. McAllester, and Avi Pfeffer. Effective bayesian inference for stochastic programs. In *AAAI/IAAI*, pages 740–747, 1997.

[51] Daphne Koller and Ronald Parr. Policy iteration for factored MDPs. In *Proceedings of the Sixteenth Annual Conference in Uncertainty in Artificial Intelligence (UAI-00)*, pages 326–334, 2000.

[52] Daphne Koller and Avi Pfeffer. Probabilistic frame-based systems. In *AAAI/IAAI*, pages 580–587, 1998.

[53] Robert Kowalski and Marek J. Sergot. A logic-based calculus of events. *New Generation Computing*, 4:67–95, 1986.

[54] Stefan Kramer. Structural regression trees. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, pages 812–819, Cambridge/Menlo Park, 1996. AAAI Press/MIT Press.

[55] Nicholas Kushmerick, Steve Hanks, and Daniel S. Weld. An algorithm for probabilistic planning. *Artificial Intelligence*, 76(1-2):239–286, 1995.

[56] Hector J. Levesque, Raymond Reiter, Yves Lespérance, Fangzhen Lin, and Richard B. Scherl. Golog: A logic programming language for dynamic domains. *J. of Logic Programming, Special Issue on Actions*, 31(1-3):59–83, 1997.

[57] Long-Ji Lin. Reinforcement learning for robots using neural networks. *PhD thesis*, 1993.

[58] Mausam and Daniel S. Weld. Solving relational MDPs with first-order machine learning. In *Programme of ICAPS'03, Workshop on Planning under Uncertainty and Incomplete Information*, Trento, Italy, 2003.

[59] John McCarthy. Situations, actions and causal laws. Technical report, Stanford University, 1963.

[60] Nicolas Meuleau, Milos Hauskrecht, Kee-Eung Kim, Leonid Peshkin, Leslie Kaelbling, Thomas Dean, and Craig Boutilier. Solving very large weakly coupled Markov decision processes. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)*, pages 165–172, Madison, WI, 1998.

[61] Stephen Muggleton. Stochastic logic programs. *Logic Programming*, 2001. Accepted subject to revision.

[62] Ronald Parr. Flexible decomposition algorithms for weakly coupled Markov decision problems. In *Proceedings of the Fourteenth Annual Conference in Uncertainty in Artificial Intelligence (UAI-98)*, pages 422–430. Morgan Kaufmann, 1998.

[63] Ronald Parr and Stuart Russell. Reinforcement learning with hierarchies of machines. In Michael I. Jordan, Michael J. Kearns, and Sara A. Solla, editors, *Advances in Neural Information Processing Systems*, volume 10. The MIT Press, 1997.

[64] Relu Patrascu, Pascal Poupart, Dale Schuurmans, Craig Boutilier, and Carlos Guestrin. Greedy linear value approximation for factored Markov decision processes. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence and Fourteenth Conference on Innovative Applications of Artificial Intelligence (AAAI/IAAI-02)*, pages 285–291, Edmonton, Alberta, Canada, July 2002.

[65] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, 1988.

[66] Edwin P. D. Pednault. ADL and the state-transition model of action. *J. Logic and Computation*, 4(5):467–512, 1994.

[67] Avi Pfeffer. IBAL: A probabilistic rational programming language. In *IJCAI*, pages 733–740, 2001.

[68] Avi Pfeffer, Daphne Koller, Brian Milch, and Ken T. Takusagawa. SPOOK: A system for probabilistic object-oriented knowledge representation. In *Proceedings of the Fifteenth Annual Conference in Uncertainty in Artificial Intelligence (UAI-99)*, pages 541–550, 1999.

[69] David Poole. Exploiting the rule structure for decision making within the independent choice logic. In *Proceedings of the Eleventh Annual Conference in Uncertainty in Artificial Intelligence (UAI-95)*, pages 454–463, 1995.

[70] David Poole. The independent choice logic for modeling multiple agents under uncertainty. *Artificial Intelligence*, 94(1-2):7–56, 1997.

[71] Doina Precup, Richard S. Sutton, and Satinder P. Singh. Planning with closed-loop macro actions. In *Working notes of the 1997 AAAI Fall Symposium on Model-directed Autonomous Systems*, 1997.

[72] Doina Precup, Richard S. Sutton, and Satinder P. Singh. Theoretical results on reinforcement learning with temporally abstract options. In *European Conference on Machine Learning*, pages 382–393, 1998.

[73] Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Eiley & Sons, Inc., New York, NY, 1994.

[74] Martin L. Puterman and Moon Chirl Shin. Modified policy iteration algorithms for discounted Markov decision processes. *Management Science*, 24:1127–1137, 1978.

[75] J. Ross Quinlan. *C45: Programs for Machine Learning*. Morgan Kaufmann, San Mateo, 1993.

[76] Raymond Reiter. *Knowledge in Action: Logical Foundations for Describing and Implementing Dynamical Systems*. The MIT Press, Cambridge, 2001.

[77] R.Iris Bahar, Erica A. Frohm, Charles M. Gaona, Gary D. Hachtel, Enrico Macii, Abelardo Pardo, and Fabio Somenzi. Algebraic decision diagrams and their applications. In *IEEE /ACM International Conference on CAD*, pages 188–191, Santa Clara, California, 1993. IEEE Computer Society Press.

[78] Erik Sandewall. *Features and Fluents: A Systematic Approach to the representation of Knowledge about Dynamical Systems*. Oxford University Press, 1994.

[79] Erik Sandewall. *Solving the Frame Problem: A Mathematical Investigation of the Common Sense Law of Inertia*. MIT Press, 1997.

[80] Marcel J. Schoppers. Universal plans for reactive robots in unpredictable environments. In John McDermott, editor, *Proceedings of the Tenth International Joint Conference on Artificial Intelligence (IJCAI-87)*, pages 1039–1046, Milan, Italy, 1987. Morgan Kaufmann publishers Inc., San Mateo, CA, USA.

[81] Lenhart K. Schubert. Monotonic solution to the frame problem in the situation calculus: A efficient method for worlds with fully specified actions. In H.E. Kyberg, R.P. Louis, and G.N. Carlson, editors, *Knowledge Representation and Defeasible Reasoning*, pages 23–67, Boston, Mass., 1990. Kluwer Academic Press.

[82] Satinder Singh and David Cohn. How to dynamically merge Markov decision processes. In Michael I. Jordan, Michael J. Kearns, and Sara A. Solla, editors, *Advances in Neural Information Processing Systems*, volume 10. The MIT Press, 1998.

[83] Satinder Pal Singh. Scaling reinforcement learning algorithms by learning variable temporal resolution models. In *MLC-92*, 1992.

[84] Satinder Pal Singh. Transfer of learning by composing solutions of elemental sequential tasks. *Machine Learning*, 8:323–339, 1992.

[85] Richard Sutton, Doina Precup, and Satinder Singh. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Journal of Artificial Intelligence*, 112:181–211, 1999.

[86] Richard S. Sutton. TD models: Modeling the world at a mixture of time scales. In *International Conference on Machine Learning*, pages 531–539, 1995.

[87] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[88] Jonathan Tash and Stuart Russell. Control strategies for a stochastic planner. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, volume 2, pages 1079–1085, Seattle, Washington, USA, 1994. AAAI Press/MIT Press.

[89] Richard Waldinger. Achieving several goals simultaneously. In E.Elcock and D.Michie, editors, *Machine Intelligence 8*, pages 94–136, Edinburgh,Scotland, 1977. Ellis Horwood.

[90] Christopher J.C.H. Watkins and Peter Dayan. Q-learning. In *Machine Learning*, pages 8:279–292, 1992.