Relational and First-Order Decision-Theoretic Planning: Foundations and Future Directions

Depth Oral Report

Scott Sanner
Department of Computer Science
University of Toronto
ssanner@cs.toronto.edu

April 24, 2004

Abstract

As research in different areas of planning, knowledge representation, decision theory, and reasoning under uncertainty has progressed over the years, it has become apparent that much of this work can be unified under the formalisms of relational and first-order decision theoretic planning. This report 1) outlines the foundations of this theory, 2) presents a framework for many of these different areas of planning, 3) considers many of the representational and algorithmic issues that allow these theories to be applied in practice, and 4) explores potential directions for future research.

Contents

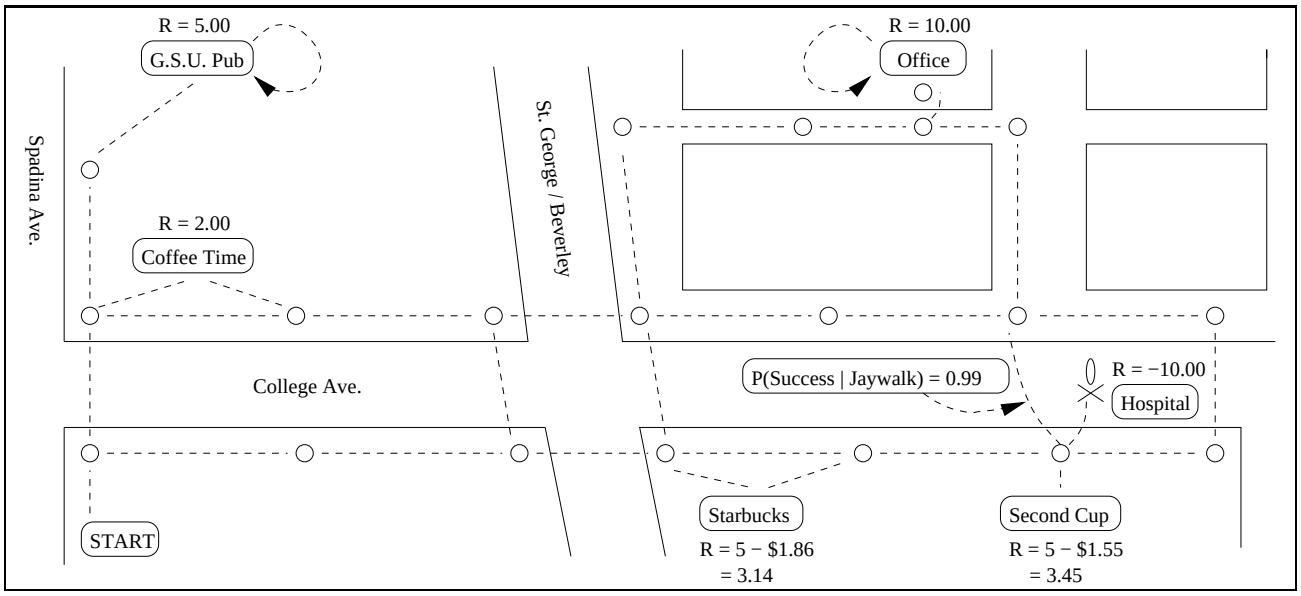

Figure 1: Above is a decision-theoretic planning problem of utmost importance to any grad student at the University of Toronto - planning the morning walk to the office. In this problem, the starting state is near the southeast corner of Spadina and College and all action transitions are deterministic, except for jaywalking from Second Cup. There are different reward values for various states and there are two absorbing states: The G.S.U. Pub - close by with a moderate reward, and the office - farther, but higher reward since this is the primary goal. The decision problem is the following: given that future rewards are discounted proportional to the number of steps n they occur in the future (e.g. γ^n where $0 \leq \gamma < 1$), find an action policy that maximizes the expected sum of discounted future reward beginning from the start state.

1 Introduction

1.1 Motivations

Decision-theoretic planning is ubiquitous. As agents operating in the world, we do it every day. For example, take the problem domain given in Figure 1. Which path should we take to the office? Maybe we're running late and we need to get there as quickly as possible. Maybe we prefer a path that takes us by a coffee shop. Maybe we'd like to optimally tradeoff *both* preferences.

In a more formal sense, decision-theoretic planning is the task of determining an optimal plan (or more generally, a policy) that optimizes some reward criterion given a state and action model of the environment. By definition, this task subsumes a large part of agent-oriented AI. And if generalized to handle partial observability, multiple agents, and sampled model dynamics (i.e., reinforcement learning), this framework subsumes almost any decision or control problem in AI.

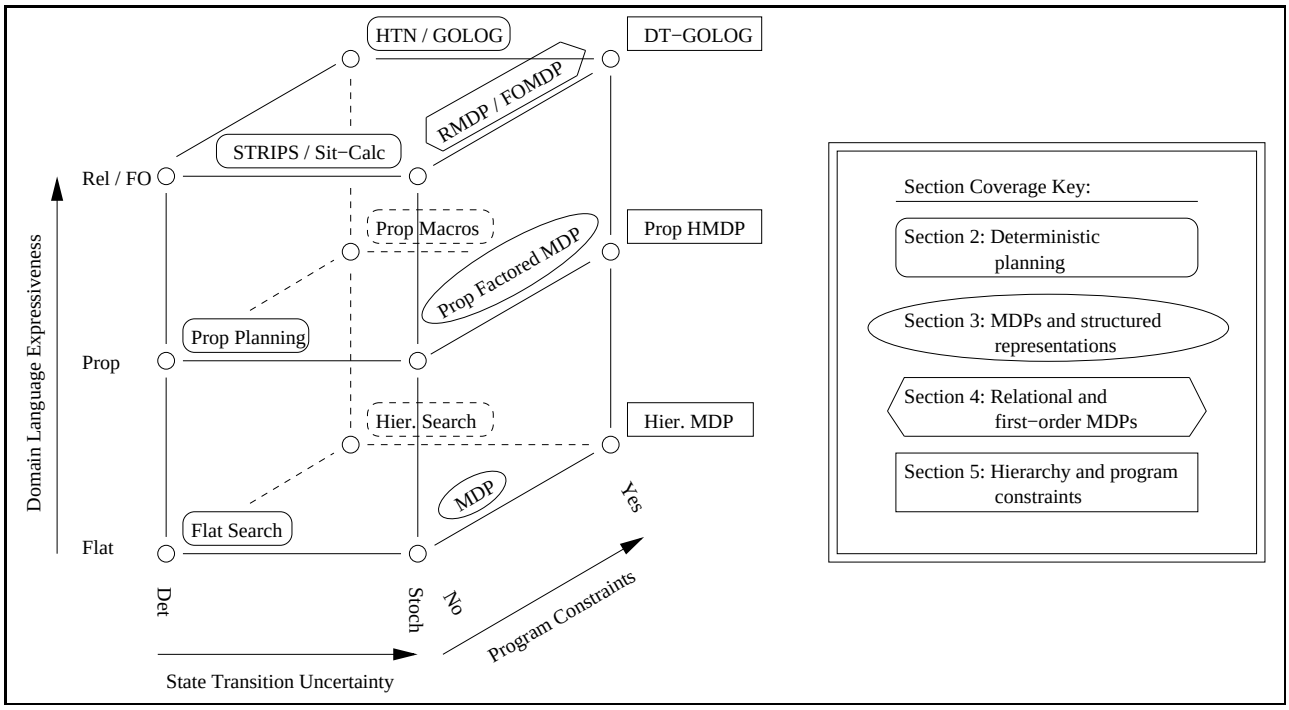

Figure 2: This diagram shows the major representational dimensions of fully-observable first-order decision-theoretic planning that will be covered in this paper. Each vertex has a corresponding label referring to the common term for this planning representation in the literature. The major conceptual divisions for these vertices and their corresponding sections in this paper are given in the key on the right. The goal of this paper is to cover the foundations building up to the most expressive formalism - planning with program constraints in relational/first-order stochastic domain representations. Once we have completed the foundations for this formalism, we will cover future directions for research from this departure point.

1.2 Outline

While we won't consider the most general frameworks for decision-theoretic planning in this paper, we will consider the general framework of relational and first-order decision theoretic planning that subsumes many planning problems for which the state is fully observable, the transition dynamics are known, and the state and action representation is discrete.

To build this unifying framework, we start from the most basic planning system - deterministic search in a flat, enumerated state space with atomic actions - and present the necessary formal machinery required to support enhancements along the following dimensions:

- *Domain language expressivity* $\in \{$*Unstructured, Propositional, Relational/First-Order*$\}$
- *State transition uncertainty* $\in \{$*Deterministic, Stochastic*$\}$
- *Program constraints* $\in \{$*No, Yes*$\}$

While these three dimensions admit 12 possible representations, we will not cover each representation in detail. Rather, we briefly cover the foundations for deterministic and stochastic planning and proceed to cover the advances that allow us to arrive at the general framework of planning with program constraints in relational/first-order stochastic domain representations. Figure 2 outlines this framework and the chapter divisions in this survey.

2 Deterministic planning

For a general overview of deterministic planning, Weld [44, 45] provide excellent references. Here, we only cover the basic representations for relational and first-order planning necessary for the understanding of subsequent sections.

2.1 Relational planning

In the paradigm of relational planning, state properties and actions are specified using relations over domain objects. When a relational planning model is grounded for a particular instantiation of domain objects, one attains a propositional model of the domain. STRIPS and PDDL are two of the most popular languages for relational planning representations so we cover these in the next section.

2.1.1 STRIPS and PDDL

STRIPS is one of the most commonly used planning languages and was introduced by Fikes and Nilsson [17]. Since the STRIPS syntax has been modified over the years, we cover a slightly enhanced but standardized STRIPS-like syntax known as PDDL (planning domain description language). The original PDDL syntax and semantics was created by McDermott et al. [30] and revised for version 2.1 by Long and Fox [18]. PDDL's most important departure from STRIPS is the addition of both universal and conditional effects that allow an action effect to modify *any* relations meeting some condition (e.g., a paint action that paints *all* objects *at* some location).

A PDDL domain consists of the following:

- *State Representation* States are described using relations over objects from a finite domain. Each ground atom (e.g., $On(b_1, b_2)$) represents a boolean proposition. To make this specification compact, a state is specified by stating only the $true$ ground atoms, any unspecified ground atoms are assumed $false$.
- *Action Representation* An action is specified by its preconditions and its effects (represented as add and delete lists in STRIPS or non-negated and negated effects in PDDL). Following is an example of the stack action expressed in PDDL (its intent should be clear from the syntax):

```
(:action stack
 :parameters (?a ?b)
 :precondition (and (forall (?c) (not (on-top-of ?c ?a)
                    (forall (?c) (not (on-top-of ?c ?b))))))
 :effect       (and (forall (?c)
                        (when (on-top-of ?a ?c)
                              (not (on-top-of ?a ?c))))
                        (on ?a ?b)))
```

- *Problem Representation* A planning problem is represented by a domain instantiation, an initial state and a goal formula. The initial state is represented by a set of true ground atoms. The goal representation in PDDL can be any closed first-order formula.

2.2 First-order planning

While the STRIPS and PDDL relational planning representations provide a template for any domain instantiation, many planning algorithms that use these representations cannot be applied when the domain instantiation is unknown. In order to plan for *all* domain instantiations, including those with infinitely many objects, it is often easier to convert the STRIPS and PDDL domains to a first-order formalism where there exist well-defined algorithms to handle these cases.

2.2.1 Situation calculus

The situation calculus is a first-order language for axiomatizing dynamic worlds [29]. It's basic language elements consist of actions, situations and fluents:

- *Actions*: Actions are first-order terms consisting of an action function symbol and arguments. For example, an action for placing b_2 on b_1 in the running blocks world example would be given by the term $stack(b_2, b_1)$.
- *Situations*: A situation is a first order term denoting a specific state. The initial situation is usually denoted as s_0 and subsequent situations resulting from action executions are obtained from the $do(a, s)$ function that represents the situation resulting from the execution of action a in state s. For example, the situation resulting from stacking b_2 on b_1 in the initial situation, and then stacking b_3 on b_2 is given by the term $do(stack(b_3, b_2), do(stack(b_2, b_1), s_0))$.
- *Fluents*: A fluent is a relation whose truth value varies from situation to situation. A fluent is simply a relation whose last argument is a situation term. For example, given a correct domain axiomatization and initial state s_0 where b_2 is not on b_1 and both blocks have nothing on them, then

$On(b_2, b_1, s_0)$ is false while $On(b_2, b_1, do(stack(b_2, b_1), s_0))$ is true. We do not consider functional fluents here for simplicity of exposition although all of the following ideas still apply in this case.

Without covering the various proposals for solutions to the Frame Problem (i.e., how to compactly characterizing what fluents change or do not change as the result of an action), we jump directly to Reiter's [38] default solution. In this solution, one must specify all positive and negative effects for a fluent. We use the following normal form for positive effect axioms:

$$\gamma_F^+(\vec{x}, a, s) \supset F(\vec{x}, do(a, s))$$

And we use the following normal form for negative effect axioms:

$$\gamma_F^-(\vec{x}, a, s) \supset \neg F(\vec{x}, do(a, s))$$

From this normal form, and explanation closure axioms stating that these are the only effects that hold, Reiter showed that we can build successor state axioms that compactly encode both the effect and frame axioms for a fluent. The format of the successor state axiom for a fluent F is as follows:[1]

$$\begin{aligned} F(\vec{x}, do(a, s)) &\equiv \Phi_F(\vec{x}, a, s) \\ &\equiv \gamma_F^+(\vec{x}, a, s) \vee F(\vec{x}, s) \wedge \neg\gamma_F^-(\vec{x}, a, s) \end{aligned} \quad (1)$$

The regression of a formula ψ through an action a is a formula ψ' that represents the conditions that must hold prior to performing a, if ψ holds after performing a. The successor state axioms lend themselves to a very natural definition of regression; specifically, if we want to regress a fluent $F(\vec{x}, do(a, s))$ through an action, we need only substitute that action for a in the successor state axiom and replace the fluent with its equivalent pre-action formula $\Phi_F(\vec{x}, a, s)$. In general, we can inductively define the regression for all first-order formulae as follows:

- $Regr(F(\vec{x}, do(a, s))) = \Phi_F(\vec{x}, a, s)$
- $Regr(\neg\psi) = \neg Regr(\psi)$
- $Regr(\psi_1 \wedge \psi_2) = Regr(\psi_1) \wedge Regr(\psi_2)$
- $Regr((\exists x)\psi) = (\exists x) Regr(\psi)$

[1] Although we do not provide it here, we note that this intuitive definition of a successor state axiom makes a translation from STRIPS/PDDL very straightforward.

2.3 First-order planning with program constraints

In general, whether one is forward-searching by progressing the initial state or backchaining by regressing a goal formula, there will always be a combinatorial explosion of possible action sequences. To mitigate this explosion, one can use program constraints to limit the sequences of actions to be explored. While a limited form of constraints on action execution were introduced in the context of hierarchical task network (HTN) planning [**?**], we instead focus on the use of general program constraints that subsume those used in HTNs.

2.3.1 GOLOG

A recent proposal for combining program constraints with search in a first-order situation calculus theory has been GOLOG [26]. In brief, GOLOG specifies the execution of a program δ as a $Do(\delta, s, s')$ relation macro that recursively expands until it terminates in primitive action applications. $Do(\delta, s, s')$ holds whenever s' is a terminating situation resulting from the execution of program δ starting in state s.

Do is defined inductively on the structure of its first argument as follows:

1. *Primitive actions (base case)*: $Do(a, s, s') \doteq Poss(a[s], s) \wedge s' = do(a, s)$

   Expands to a primitive action execution. This relation holds if a can be applied from s and s' is the successor state of doing action a in state s. It is false otherwise.

2. *Test actions*: $Do(\phi?, s, s') \doteq \phi[s] \wedge s = s'$

   Tests the truth value of expression ϕ in situation s. If $\phi[s]$ holds true, this relation holds when $s' = s$, and is false otherwise.

3. *Sequence*: $Do([\delta_1;\delta_2], s, s') \doteq \exists s''.(Do(\delta_1, s, s'') \wedge Do(\delta_2, s'', s'))$

   Applies to a sequence of two programs. This relation holds for any s' resulting from the sequential execution of both programs, and is false otherwise.

4. *Nondeterministic action choice*: $Do([\delta_1|\delta_2], s, s') \doteq (Do(\delta_1, s, s') \vee Do(\delta_2, s, s'))$

   Applies to the execution of either choice (but not both). This relation holds for any s' resulting from the successful execution of either program, and is false otherwise.

5. *Nondeterministic choice of arguments*: $Do((\pi x)\delta(x), s, s') \doteq (\exists x)\ Do(\delta(x), s, s')$

   Applies to an execution of any binding of x for $\delta(x)$. This relations holds for any s' resulting from the successful execution of any binding for $\delta(x)$, and is false otherwise.

6. *Nondeterministic iteration*: $Do(\delta^*, s, s') \doteq \{$Transitive closure of $0 \ldots \infty$ executions of $\delta\}$

   Technically, transitive closure requires a second order definition which we omit here. Fortunately, there is a constructive procedural method for computing transitive closure that will be exploited later during search. Intuitively, this relation holds for any situation s' reachable from $0 \ldots \infty$ applications of the action δ.

One can also construct definitions for if/then conditional statements, while loops, procedures, and programs based on these foundational definitions.

Given the previously defined GOLOG program semantics, one can perform the following reasoning:

- *Correctness*: $Axioms \models (\forall s).Do(\delta, S_0, s) \supset P(s)$

  Prove this to show that property P holds in any situation resulting from a termination of the program δ (starting from situation S_0).

- *Termination*: $Axioms \models (\exists s).Do(\delta, S_0, s)$

  Prove this to show that program δ has a successful terminating action sequence starting from S_0.

- *Goal-Oriented Planning*: $Axioms \models (\exists s).Do(\delta, S_0, s) \wedge G(s)$

  For planning, one can use this to show that there exists an action sequence that satisfies the program constraints δ and satisfies the goal G starting from an initial state.

We focus on this last type of reasoning for a planning problem axiomatized as a situation calculus domain theory. If we can prove that a valid action sequence constrained by δ satisfies a goal starting from an initial situation, then we can extract a successful plan by examining the situation binding for s. s should look something like $do(a_1, do(\ldots, do(a_n)))$, which is effectively a sequence of actions satisfying the constraints of δ and leading to a situation that achieves the goal. These properties can be verified by regressing them through the action sequence to see if they are satisfied in the initial state.

At first, it seems that theorem proving in the second-order semantics of GOLOG would be quite difficult. Fortunately, however, the second-order semantics is only used in cases of transitive closure, and there is a constructive, procedural method for computing a transitive closure without appealing to second-order theorem proving. Quite simply, we can perform a forward-search through all action sequences consistent with the program constraints, and regress the goal through these postulated actions to determine if they are implied by the initial state. Any method that searches *all* legal action sequences will find a solution *iff* there is a solution satisfying the second-order semantics of GOLOG programs [26].

3 MDPs and structured representations

So far, we have only covered the deterministic planning paradigm. To generalize from deterministic planning to decision-theoretic planning with stochastic action effects, we introduce the Markov decision process (MDP) model. We start by introducing the MDP representation and a number of traditional algorithms for finding optimal policies (to be defined shortly) for MDPs. Then we cover a number of enhancements to the MDP representation that often improve the time and space requirements of these traditional algorithms.

3.1 Markov decision processes

In the MDP [37] model, an agent is assumed to fully observe the current state and choose an action to execute from that state. Based on that state and action, nature then chooses a next state according to some fixed probability distribution. In an infinite-horizon MDP, this process repeats itself indefinitely. Assuming there is a reward associated with each state and action, the goal of the agent is to maximize the expected sum of discounted rewards received over an infinite horizon.[2] This criterion assumes that a reward received n steps in the future is discounted by γ^n where γ is a discount factor satisfying $0 \leq \gamma < 1$. The goal of the agent is to choose its actions in order to maximize the expected, discounted future reward in this model.

Given this high-level description of the MDP model, we now proceed to provide a more detailed mathematical definition of an MDP and describe a number of algorithms that can be used to find optimal action policies for this model.

3.1.1 MDP representation

Formally, a finite state and action MDP is a tuple: $\langle S, A, T, R \rangle$ where: S is a finite state space, A is a finite set of actions, T is a transition function: $T : S \times A \times S \rightarrow [0, 1]$ where $T(s, a, \cdot)$ is a probability distribution over S for any $s \in S$ and $a \in A$, and R is a bounded reward function $R : S \times A \rightarrow \mathbb{R}$.

As stated above, our goal is to find a policy that maximizes the infinite horizon, discounted reward criterion: $E_\pi[\sum_{t=0}^{\infty} \gamma^t \cdot r^t | s_0]$, where r^t is a reward obtained at time t, γ is a discount factor as defined above, π is the policy being executed, and s_0 is the initial starting state. Based on this reward criterion, we define the value function for a policy π as the following:

$$V_\pi(s) = E_\pi \left[\sum_{t=0}^{\infty} \gamma^t \cdot r^t \;\middle|\; s_0 = s \right] \tag{2}$$

[2] Although we do not discuss it here, there are other alternatives to discounting such as averaging the reward received over an infinite horizon.

Intuitively, the value function for a policy π is the expected sum of discounted rewards accumulated while executing that policy when starting from state s.

For the MDP model discussed here, the optimal policy can be shown to be stationary [37]. Consequently, we use a stationary policy representation of the form $\pi : S \rightarrow A$, with $\pi(s)$ denoting the action to be executed in state s. An optimal policy π^* is the policy that maximizes the value function for all states. We denote the optimal value function as $V^*(s)$ and note that it satisfies the following equality:

$$V^*(s) = \max_a \left\{ R(s,a) + \gamma \sum_{t \in S} T(s,a,t) \cdot V^*(t) \right\} \tag{3}$$

For convenience of notation in the following sections, we sometimes write the MDP in vector and matrix form. We represent the reward for each state and action a as $|A|$ column vectors denoted by R_a. We represent the value funcion for each state as a column vector V. And we represent the transition matrix indexed by states current state s and a next state t for each action a as $|A|$ transition matrices T_a. In this case, the optimal value function in equation 3 can be restated as the following:

$$V^* = \max_a (R_a + \gamma T_a V^* \tag{4}$$

3.1.2 Dynamic programming

We begin our discussion of dynamic programming by providing two equations that form the basis of the stochastic dynamic programming algorithms used to solve MDPs.

We define $V_\pi^0 = R(s)$ and then define the i-stage-to-go value function for a policy π as the following:

$$V_i^\pi(s) = R(s, \pi(s)) + \gamma \sum_{t \in S} T(s, \pi(s), t) \cdot V_{i-1}^\pi(t) \tag{5}$$

Based on this definition, Bellman's *principle of optimality* [6] establishes the following relationship between the optimal value function at stage i and the optimal value function at the previous stage $i-1$:

$$V_i^*(s) = \max_{a \in A} \left\{ R(s, \pi(s)) + \gamma \sum_{t \in S} T(s, \pi(s), t) \cdot V_{i-1}^*(t) \right\} \tag{6}$$

Value iteration

We start with an algorithm known as value iteration that directly implements equation 6. Here, we start with $V^0(s) = R(s)$ and perform the dynamic programming backup given in equation 6 for each state $V^1(s)$ using the cached value of $V^0(s)$. We repeat this process for each i, backing up the value function for $V^i(s)$ from $V^{i-1}(s)$ until we have computed the intended i-stage-to-go value function. This algorithm is demonstrated graphically in part (b) of Figure 3.

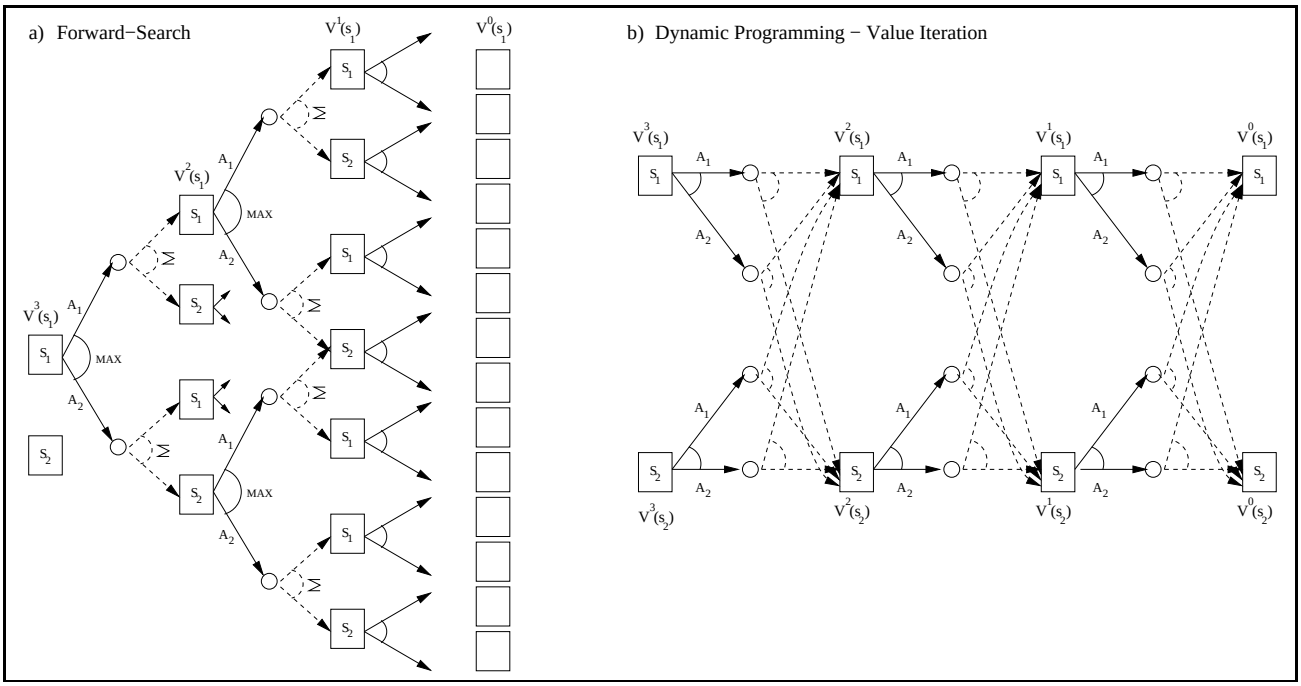

Figure 3: A diagram demonstrating a) forward evaluation of the MDP value function and b) dynamic programming regression evaluation of the MDP value function. Both methods return the same value for $V^4(s)$, but the forward evaluation requires exponential time in the search depth $O((|S| \cdot |A|)^d)$ and only calculates the value for one initial state whereas dynamic programming caches it's results on each backup thus requiring only polynomial time in the search depth $O(|S| \cdot |A| \cdot d)$ and solving for the value function at *every* state.

Often, value iteration is rewritten in two steps to separate out the action regression and maximization steps. In this case, we first perform a Bellman backup where we compute the i-stage-to-go Q function for every action and state:

$$Q^i(s,a) = R(s,a) + \gamma \cdot \sum_{t \in S} T(s,a,t) \cdot V^{i-1}(t) \tag{7}$$

Then we maximize over each action to determine the value of the regressed state:

$$V^i(s) = \max_{a \in A} \{Qi^*(s,a)\} \tag{8}$$

This is clearly equivalent to equation 6 but is in a form that we will refer to later since it separates the algorithm into its two conceptual components.

Puterman [37] shows that terminating once the following condition is met guarantees ϵ-optimality for

the policy derived on the ith iteration[3]:

$$\|V^i - V^{i-1}\|_\infty < \frac{\epsilon(1-\gamma)}{2\gamma} \tag{9}$$

ϵ-optimality ensures that $\|V^i(s) - V^*(s)\|_\infty < \epsilon$, thus the policy derived on the ith iteration loses no more than ϵ in value over the infinite horizon in comparison to the optimal value function.

We note that the value iteration approach requires time polynomial in the search depth d, i.e., $O(|S| \cdot |A| \cdot d)$, and solves for the value function at *every* state. It can be proved that value iteration has a linear rate of convergence.

Policy iteration

At each step of the value iteration backup, we are implicitly performing a policy update, determining the best action to take from every state in order to maximize reward. Another approach to dynamic programming is the following:

1. Pick an arbitrary initial decision policy π_0 and set $i = 0$.
2. Solve for V_{π_i} (see below).
3. Find a new policy $\pi_{i+1} = \arg\max_{\pi \in \Pi} \{R_\pi + \gamma T_\pi V_\pi\}$
4. If $\pi_{i+1} \neq \pi i$ then increment i and go to step 2 else return π_{i+1}.

We note that the policy evaluation of a *fixed* policy π reduces to a linear system since we no longer retain the max. Thus, we can solve for V_π by computing the right-hand side of the following equation:

$$V_\pi = R_\pi (I - \gamma T_\pi)^{-1} \tag{10}$$

We note that a unique solution for V^* always exists since the Markovian properties of T guarantee that $I - \gamma T$ is invertible. We also note that this step takes time $O(|S|^3)$ due to the matrix inversion.

Once policy iteration has terminated, the final policy returned is the optimal policy π^* and the value function corresponding to this policy is the optimal value function. It can be proved that policy iteration has a superlinear rate of convergence under certain conditions [37].

So far, we have implicitly assumed that the above algorithms perform synchronous updates, i.e., that we are updating the value function in value iteration for all states and that we are improving the policy in policy iteration for all states. We additionally note that there are a number of asynchronous variants of value and policy iteration that do not update the value or improve the policy at every state on all

[3]The policy derived on the ith iteration is simply the maximizing action a for each s computed during the dynamic programming backup.

iterations, yet still retain similar convergence properties. These algorithm variants are discussed by Puterman [37] and Bertsekas and Tsitsiklis [7].

Modified policy iteration

A comparison of the two previous algorithms reveals that they occupy two extremes in terms of policy updates: Value iteration performs a policy update on every intermediate value function whereas policy iteration performs an update only after solving directly for V_π.

Consequently, there is middle ground between these two approaches known as modified policy iteration. In this algorithm, we simply iterate between policy evaluation and policy improvement phases until our policy is ϵ-optimal using the same terminating criteria of value iteration. Algorithm convergence requires only that the policy evaluation decreases the residual Bellman error $\|V^*(s) - V(s)\|_\infty$, e.g., *any* number of backups for a fixed policy. Modified policy iteration has a superlinear convergence rate under certain conditions and as noted by Puterman [37], it often empirically requires less computation time than both value and policy iteration.

3.1.3 Forward-search

If we reexamine equation 6, we note that we can actually compute this recurrence in a forward-search manner by starting at an initial state and unfolding the recurrence to depth i and then computing the expectation and maximization as we return to the initial state. A graphical representation of the unfolding of this computation is shown in Figure 3, part (a). We note that determining the value $V^i(s)$ for a *single* state using this method requires time exponential in the search depth i (i.e. $O((|S| \cdot |A|)^i)$).

Since we are performing forward search to a fixed *a priori* search depth, we can determine the minimum depth i to search if we want an ϵ bound on the maximum error of our value function, given knowledge of our discount factor γ and our maximum reward R_{max}:

$$i \geq \log_\gamma \left(\frac{\epsilon(1-\gamma)}{R_{max}} \right) - 1 \tag{11}$$

3.1.4 Real-time dynamic programming

The real-time dynamic programming (RTDP) framework [5] is a hybrid approach that combines real-time forward search with dynamic programming. This approach uses limited depth, forward-search backups to update the value function of the set of states visited during on-line trials. The policy used for the trials is the optimal policy for the current value function. Since backed-up and cached values from one step are used by other steps, this approach mixes the forward-search and dynamic programming paradigms. It is provably convergent and has the advantage that it only derives the value function for the set of states reachable from an initial state distribution. This can often be more efficient than synchronous dynamic programming approaches when the set of reachable states is small compared to

the total number of states.

3.1.5 Linear programming

We briefly mention that the MDP can be solved by formulating it as a linear program (LP). The fact that such a solution exists follows from the notion that the optimal policy and value function must satisfy the following matrix inequality:

$$V^* \geq R_{\pi^*} + \gamma T_{\pi^*} V^* \tag{12}$$

The LP formulations are as follows:

Primal LP formulation

$$\begin{aligned}
\text{Variables:} \quad & \alpha(s) \\
\text{Minimize:} \quad & \sum_{s \in S} \alpha(s) V(s) \\
\text{Subject to:} \quad & V(s) - \sum_{s \in S} \gamma P(s,a,t) V(t) \geq R(s,a), \quad \forall a \in A, s \in S
\end{aligned}$$

Dual LP formulation

$$\begin{aligned}
\text{Variables:} \quad & \chi(s,a), \alpha(s) \\
\text{Minimize:} \quad & \sum_{s \in S} \sum_{a \in A_s} R(s,a) \chi(s,a) \\
\text{Subject to:} \quad & \sum_{a \in A_t} \chi(t,a) - \sum_{s \in S} \sum_{a \in A_s} \gamma P(t|s,a) \chi(s,a) = \alpha(t), \quad \forall t \in S \\
& \chi(s,a) \geq 0, \quad \forall a \in A, s \in S
\end{aligned}$$

Puterman [37] notes that solving the dual LP formulation is often more efficient than solving the primal LP formulation.

3.2 Structured MDP representations

While the more efficient MDP solution techniques from the previous section require time polynomial in $|S|$ and $|A|$ for each iteration, we note that $|S|$ can be very large. In fact, if we let S be represented by n binary state variables (i.e., $S = \{S_1 \times S_2 \times \cdots \times S_n\}$), then the total number of states is 2^n. This is the well-known curse of dimensionality and it unfortunately makes the enumerated-state MDP solution algorithms run in exponential time in the number of state variables n.

Consequently, efficient representations and algorithms are extremely important for the application of

MDPs in real-world applications. This is especially true for fields such as decision-theoretic planning that often use propositional representations encoding billions of states. The previously described algorithms would never terminate in a reasonable amount of time if we used an enumerated state representation for large numbers of state variables.

In the following sections, we describe structured representations and algorithms that mitigate these problems and make it possible to solve for exact and approximate solutions in extremely large MDPs.

3.2.1 Factored transition and reward dynamics

One of the huge MDP representation bottlenecks stems from representing the transition matrices. With a state formed from 10 binary variables, a single joint action probability would be of the form $P(S'_1, S'_2, \ldots, S'_{10}|S_1, S_2, \ldots S_{10}, A)$ (with the S' variables representing the next state variables). If this distribution were represented in tabular format, it would require $|A|$ matrices, each with 2^{20} entries. Even if it were possible to directly specify or learn a low variance estimate of this many parameters, it would become prohibitively difficult to store this table as more variables were added.

However, from an intuitive standpoint, most actions affect only very few variables. Since our effects are probabilistic, we need a compact, structured way of specifying the transition probability distribution. Consequently, a dynamic Bayes (DBN) [9] net is an ideal representation. We can similarly use a factored representation known as an influence diagram to model the state variables that influence the reward function. A sample DBN and influence diagram are pictured in part (a) of Figure 4.

For this DBN, we can write the full conditional joint transition distribution for an action a as: $P(S'_1, S'_2, \ldots, S'_5|S_1, S_2, \ldots, S_5, a)$. This distribution is given by the following factored definition:

$$\begin{aligned} P(S'_1, S'_2, S'_3, S'_4, S'_5|S_1, S_2, S_3, S_4, S_5, a) &= P(S'_1|S_1, a)P(S'_2|S_2, S_3, a)P(S'_3|S_2, S_3, S_4, a) \\ &\quad P(S'_4|S_4, a)P(S'_5|S_4, S_5, a) \end{aligned}$$

We note that the full conditional joint distribution for a single action would take 992 parameters to represent as a full CPT while the factored representation requires only 20 parameters to specify the same exact distribution under the given DBN conditional independence assumptions.

There are alternative representations to the DBN transition representation such as probabilistic generalizations of STRIPS [8] but Littman [27] proved that this representation can be converted to a dynamic Bayes net representation with only a polynomial blowup in size. This effectively demonstrates that both formalisms are representationally equivalent.

By using DBN and influence diagram structures to efficiently representing transition and reward dependencies, we can often save a considerable amount of space in these representations. In the worst case transition matrix representation, every state variable depends on every other variable, thus requiring a number of parameters exponential in the state variables. And in the best case, every state variable is

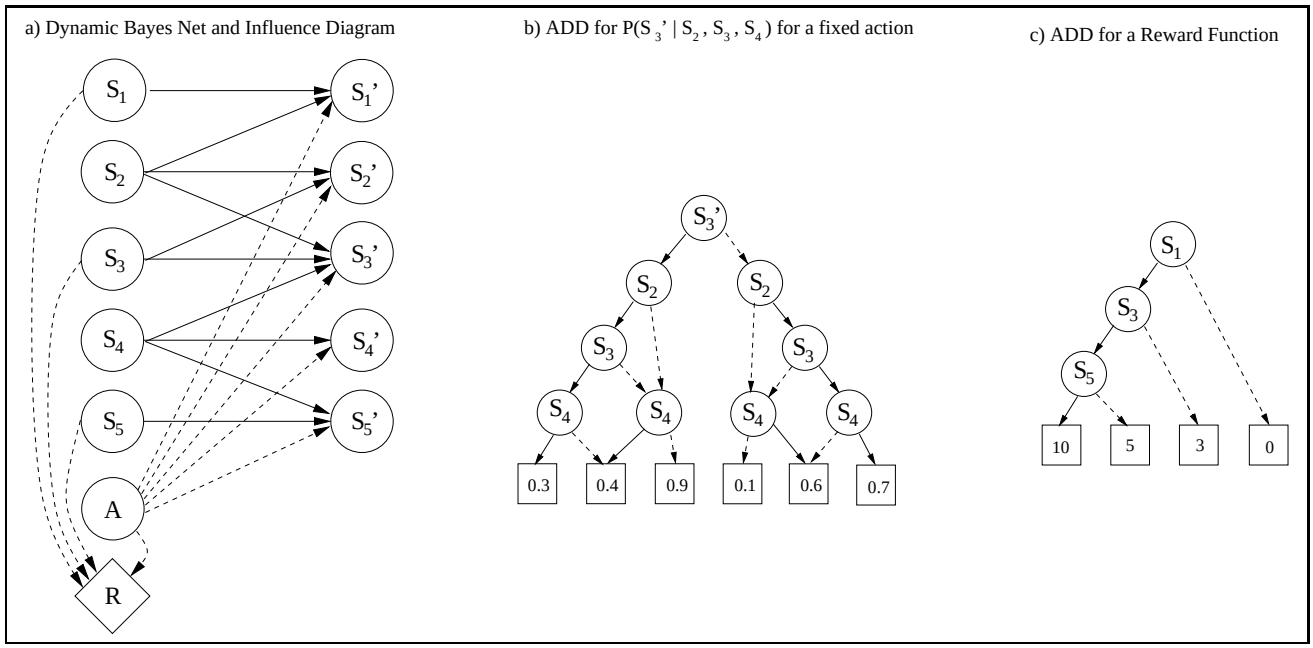

Figure 4: a) A dynamic Bayes network and influence diagram representing a transition function and a reward function. b) An efficient encoding of the transition function CPT for the DBN (assuming the action is known). Note that S_3' sums to one over all possible previous states. c) An efficient encoding of the reward function based on the states that influence it in the influence diagram.

completely independent of all other state variables, requiring just one parameter per state variable (in the binary case) and thus yielding an exponential reduction in parameters over the worst case. In the typical case, every state variable depends on only a *few* others in the transition matrix representation, thus falling somewhere between these two extremes. While a factored transition and reward representation can yield substantial savings for the MDP representation, we note that this factoring cannot often be preserved in the value function due to the correlation of action effects over sufficiently extended periods of time. Nevertheless, representing large MDPs is a first step toward solving them and subsequent techniques will take advantage of this factored structure for efficient computation and approximation.

3.2.2 Context-specific independence and approximation

Even if we can represent the joint transition probability as a Bayes net with a conditional probability table (CPT) for each successor state variable, we can often represent these tables more efficiently than by enumerating the entire table. Quite often, we find that certain values of variables in a CPT render the other values irrelevant. This is known as context-specific independence (CSI) [11].

For example, if we know that S_3' depends on S_2, S_3 and S_4 but that in the context of $S_2 = false$ that S_3' depends only additionally on S_4, then we can merge the entries for $S_3 = true$ and $S_3 = false$

in this context. In order to represent this CSI compactly, we can use a decision tree or more generally an ADD [14]. An example ADD for this probability distribution is given in Figure 4 part (b) and an example ADD for the reward is given in Figure 4 part (c).

In addition to the representational efficiency of merging identical values in a CPT, we note that computation with these CPT data structures, especially ADDs, is also very efficient. When we multiply or add two CPTs represented as ADDs, we are just performing the standard multiply and add operations for ADDs that avoid enumerating the entire state space. Consequently, we can use these data structures as all intermediate representations during value iteration, thus performing value iteration without explicitly requiring full enumeration of the state space.

One additional benefit of tree or ADD specification of CPTs is that it allows one to prune internal nodes in a decision tree and replace these nodes with the minimum and maximum value of the tree or ADD rooted at that node. One can then perform value iteration maintaining these upper and lower bounds. Since the Bellman backup is a known contraction operator, we know that the algorithm must still converge, this time to an approximately optimal solution. One can use various heuristics for pruning and reorganizing the internal decision variable orderings to minimize the amount of approximation required. One can also control the amount of pruning in response to the ADD size in order to bound the maximum representation size.

Various combinations of these ideas have been implemented in a number of propositional decision-theoretic planning systems, e.g. policy iteration using decision trees [10], value iteration using ADDs [23], and ADDs with approximation pruning techniques [42]. In practice, these algorithms are quite efficient, allowing one to easily multiply CPTs representing over a billion states. Nonetheless, while the best case value function and policy representations could be constants (independent of state), the worst case representations are exponential in the number of state variables, having a different value or action policy for every state. In the general case, Littman et al. [28] note that finding an optimal plan using tree-structured CPTs is EXP-COMPLETE although it is unclear whether such complexity results would extend to finding approximately optimal solutions when using tree or ADD-structured CPTs.

3.2.3 Additive structure and approximation

One can factor the value function into weighted additive components in an effort to capture additive structure in the value function and use techniques such as linear approximation to select weights that minimize the overall approximation error. One can also use more general function approximation such as nonlinear functions or neural nets [7] but it is generally difficult to provide useful convergence properties for such approximation architectures so we do not discuss them here. We formalize the case of linear value function approximation next.

If we have n states in our MDP, the exact value function can be specified as a vector in $\mathbb{R}^n$. This

vector can be approximated by a linear combination of basis functions (i.e. vectors). The linear subspace spanned by this linear combination might not include the actual value function, but one can perform a projection to minimize some error measure between the real value function and the linear combination of basis functions.

For the following formalization, we assume that A is an $n \times k$ matrix representing the concatenation of k basis function vectors (where k is usually small) and $\vec{w}$ is a k element column vector representing the linear combination weights for each of the basis vectors. Then $A\vec{w}$ represents a linear approximation of the value function.

As our projection method, we could use a least-squares linear projection (by minimizing the $\mathcal{L}_2$ (Euclidean norm) error measure). Unfortunately, as done in [24, 25] but as pointed out by Guestrin et al. [20, 21] (GKPV from here out), using these approximate representations leads to difficulties when proving convergence of the standard MDP algorithms. Consequently, as suggested in GKPV, we take a max-norm projection approach (by minimizing $\mathcal{L}_\infty$ (max-norm) error) that allows convergence to be shown much more easily since it directly minimizes the error used in the convergence proofs for MDPs. Furthermore, these techniques lead to very efficient projection algorithms.

Following is a method to determine the minimum $\mathcal{L}_\infty$ error representation for an approximate value function under a fixed policy π:

$$\begin{aligned} \vec{w}^{\,*} &= arg\min_{\vec{w}} \ \|A\vec{w} - (R_\pi + \gamma P_\pi A\vec{w})\|_\infty \\ &= arg\min_{\vec{w}} \ \|(A - \gamma P_\pi A)\vec{w} - R_\pi\|_\infty \end{aligned} \tag{13}$$

From this, we can derive a simple algorithm for approximate policy iteration by repeating the following steps until the policy converges:

$$\vec{w}^{(t)} = arg\min_{\vec{w}} \ \|(A - \gamma P_{\pi^{(t)}} A)\vec{w} - R_{\pi^{(t)}}\|_\infty \tag{14}$$

$$\pi^{(t+1)} = arg\max_{\pi \in \Pi} \ (R_\pi + \gamma P_\pi A\vec{w}^{(t)}) \tag{15}$$

If we let $C = A - \gamma P_{\pi^{(t)}} A$ and $\vec{b} = R_{\pi^{(t)}}$, then we can use the following linear program to find the

optimal $\vec{w}$ for the max-norm projection:

$$\begin{aligned}
\text{Variables:} \quad & w_1, \ldots, w_n, \phi \\
\text{Minimize:} \quad & \phi \\
\text{Subject to:} \quad & \phi \geq \sum_{j=1}^{k} c_{ij} w_j - b_i, \quad \forall i \in \{1 \ldots n\} \\
& \phi \geq b_i - \sum_{j=1}^{k} c_{ij} w_j, \quad \forall i \in \{1 \ldots n\}
\end{aligned}$$

This linear program selects a set of weights that minimize the absolute projection error over all states. The problem as noted by GKPV is that there is a constraint required for every state and this is unfortunately exponential in the number of state variables. Fortunately though, GKPV show that one can use a variable elimination algorithm to generate a number of constraints bounded by the tree width of the transition DBN. Additionally, one could use a constraint generation approach to avoid specifying all of these constraints [40].

Using these techniques, one can either take a policy iteration approach [20] as described above or a direct linear programming approach [39] to solving the MDP in one step. On problems with a large amount of additive structure in the value function, such approaches can approximately solve problems far larger than those solvable by context-specific independence alone. Unfortunately, one cannot obtain *a priori* error bounds on the final value function and *a posteriori* error bounds can only be directly obtained when using approximate policy iteration. Furthermore, we note that while such techniques can be used in cases where the value function is believed to have additive separability, additive separability in the reward does not imply that such additive structure will be reflected in the optimal value function.

One additional difficulty with linear value function approximation is that of generating a good set of basis functions. Certainly, a poor set of basis functions that is far from the exact value function under any norm can have a severely adverse impact on decision quality. Consequently, one can take a number of approaches to generating basis functions such as finding subtasks with additive reward [35] or performing branch-and-bound search to find Bellman-error minimizing basis functions [36]. Unfortunately, at this point in time, generating a good basis function set is still more of an art than a science, and there are no currently known methods that allow one to attain *a priori* guarantees on the decision quality for a given set of basis functions.

4 Relational and first-order MDPs

In the previous section, we discussed decision-theoretic planning models that were based on a propositionally factored structure. However, we can generalize from the relational domain models such as

STRIPS or PDDL discussed in the deterministic planning section to probabilistic variants of these representations based on the MDP model. From these relational MDP specifications, we can either fix the domain objects and convert the representation to a propositional MDP or we can generalize the domain to a first-order MDP that concisely represents all possible domain instantiations. In this section, we survey some of the approaches for solving relational and first-order MDPs.

4.1 Relational MDPs

4.1.1 PSTRIPS / PPDDL

Just as we used the STRIPS and PDDL languages for specifying relational MDPs (that could be generalized to the first-order case), we can also specify probabilistic versions of these representations. The generalization that PSTRIPS [8] and PPDDL [47] make to their predecessors is the ability to specify probabilistic effects, e.g., with respect to the blocks world, we can specify that with probability 0.7 a stack operation will successfully place a block on another block, and with probability 0.3, it will be dropped.

To make this more concrete, an example of a PPDDL action specification follows:

```
(:action stack
 :parameters (?a ?b)
 :precondition (and (forall (?c) (not (on-top-of ?c ?a)
                    (forall (?c) (not (on-top-of ?c ?b))))))
 :effect (probabilistic 0.7 (and (forall (?c)
                                      (when (on-top-of ?a ?c)
                                            (not (on-top-of ?a ?c))))
                                 (on ?a ?b))))
```

4.1.2 Policy induction methods

An approach to solving relational MDPs motivated by policy iteration is a policy induction approach used by Givan et al [46, 16]. The basic idea is to start with a domain in a PSTRIPS-like language, an initial policy and a heuristic cost function that offers a heuristic estimate of distance to the goal. Then the following two operations are repeated until the change in a decision-list policy π is small:

1. $D \Leftarrow \mathit{Draw\text{-}Training\text{-}Set}(\pi, \cdots)$
2. $\pi \Leftarrow \mathit{Learn\text{-}Decision\text{-}List}(D)$

The *Draw-Training-Set* function samples a number of initial states and then samples a number of trajectories starting from these states using the current policy π. The reward is accumulated and

used to improve the Q-function estimate of each state and action pair in the trajectory. After this is performed for a set number of trials, the set of state and action pairs generated and the newly estimated Q-functions are returned.

The *Learn-Decision-List* function takes this sampled information and Q-function estimates and learns a decision-list policy from these improved Q-function estimates. The decisions in this list use a restricted object-oriented relational language, thus providing a policy language bias that seems to apply well to many domains. For example, in a blocks world domain with relations $holding(a)$, $on(a, b)$, and $gon(a, b)$ specifying the goal state, a small decision list policy could be given as follows:

$$gon(a, b) \wedge \neg on(a, b) \wedge holding(a) \longrightarrow putdown(a, b)$$
$$gon(a, b) \wedge \neg on(a, b) \longrightarrow pickup(a)$$

These decision lists are learned via a heuristic beam search to avoid enumeration of all possible relational patterns.

Overall, this algorithm can be summarized as a set of local search techniques (similar in spirit to the previously mentioned RTDP) for value determination under a given policy, and policy improvement via an inductive policy inference step. An approach additionally used in this line of research is to start with small problems and build up to larger ones over time. This approach bootstraps from policies learned for small problems with the hope that these policies will generalize to larger domains (which is often empirically the case).

This approach has provided many promising empirical results. It learns reasonable policies for many domains and often performs better than state-of-the-art planners such as FF-Plan on the average solution-length criteria. Furthermore, it is one of the few methods that has been reliably used in stochastic relational domains. These successes largely relate to the fact that the learned policies, which are represented in a subset of first-order logic, often generalize well to new domains. Additionally, the assumption that policies for small domains generalize to larger domains seems to be one that holds for many practical problems. Yet, the main drawback of this work is that it provides no theoretic guarantees on the quality of a policy or the quality of its generalization to a larger domain. And given that a domain may have an abrupt switch in policy for a given domain size (i.e., use a plane once you have at least 1,000 boxes to deliver), it is unlikely that such a guarantee could be given without severe restrictions.

4.1.3 Generalizing from approximate policies

The work of Guestrin et al. [19] is based on a similar idea that policies often generalize over multiple domain instantiations. This work uses the relational nature of the domain specification to determine small object-centered value functions that can serve as basis functions that are summed to form a global

value function. For example, in the FreeCraft domain [19], one has control over multiple instances of *footmen* that battle against *enemies*. For this domain, basis functions might be generated for each footman and the state of any enemies that footman is fighting. Then the overall value function would be a sum of the individual value functions for each footman.

Since the local basis value functions are solved for in their entirety, these basis functions must be chosen such that 1) they are small enough to be efficiently represented and computed and 2) their sum provides an approximation to the full value function that yields a policy of reasonable quality. In this approach, the basis functions can either be specified by hand or learned inductively from small problem solutions using decision tree algorithms.

Once the basis functions are chosen for the global value function representation, this approach then proceeds to learn the individual value functions based on a standard linear program program formulation. However, because this MDP solution should generalize to many worlds, some of these worlds are sampled from a distribution that falls off exponentially with the world size. Then a joint LP is formulated that contains a linear program for all sampled worlds (weighted by their probability) that sum over the basis functions for objects used in that world.

Once a solution is obtained, the basis value functions can clearly be used for *any* instantiation of a new world. Specifically, when a world is instantiated, a value function is constructed as a sum of basis functions for the instantiated objects.[4] Then, one can simply plan by performing one-step look-ahead and choosing the optimal action.

This work provides a PAC-bounded generalization guarantee under the assumption that worlds are stochastically sampled and the probability of sampling a world falls off exponentially with its size (i.e., the number of domain objects instantiated). In addition to providing PAC-generalization bounds for the global value function, this work also provides promising empirical results for two very large domains with many similar objects and an additive reward function. However, this approach suffers from the drawbacks that plague approximation methods: There is no known method that always generates *good* basis functions and there is no way to obtain *a priori* guarantees on the decision quality afforded by a given set of basis functions.

4.2 First-order MDPs

First-order MDPs (FOMDPs) are a generalization of the situation calculus under an MDP framework. These ideas were first introduced by Boutilier et al. [12]. While we do not show it here, we note that one can convert a PSTRIPS or PPDDL domain to a first-order representation.

[4]There are no weights for basis functions here. In this approach, the basis value functions themselves are learned.

4.2.1 Notational preliminaries

Before we delve into the representation of FOMDPs, we introduce a case representation that will be used to represent probability, reward, and value function partitions over first-order state space. A case partition

$$t \quad = \quad case[\phi_1, t_1; \cdots ; \phi_n, t_n]$$

is an abbreviation for the formula

$$\bigvee_{i \leq n} \phi_i \wedge t = t_i$$

where the ϕ_i are first-order state formula and the t_i are terms. This notation partitions first-order state space such that the value of a case statement corresponds to the value of the formula partitions that are true for that state. While, it is not the case that this partitioning is always exhaustive and mutually exclusive, we often explicitly maintain mutual exclusivity.

We will need to perform mathematical operations on these case statements so we formalize these operations as follows:

$$case[\phi_i, t_i : i \leq n] \otimes case[\psi_j, v_j : j \leq m] = case[\phi_i \wedge \psi_j, t_i \cdot v_j : i \leq n, j \leq m]$$
$$case[\phi_i, t_i : i \leq n] \oplus case[\psi_j, v_j : j \leq m] = case[\phi_i \wedge \psi_j, t_i + v_j : i \leq n, j \leq m]$$
$$case[\phi_i, t_i : i \leq n] \ominus case[\psi_j, v_j : j \leq m] = case[\phi_i \wedge \psi_j, t_i - v_j : i \leq n, j \leq m]$$
$$case[\phi_i, t_i : i \leq n] \cup case[\psi_j, v_j : j \leq m] = case[\phi_1, t_1; \cdots ; \phi_n, t_n; \psi_1, v_1; \cdots ; \psi_m, v_m]$$

Intuitively, the $\otimes$ operator states that the sum of two cases statements is simply the cartesian product of the formulae in each respective case statement labelled with the product of their respective terms. The same idea holds true for $\oplus$ and $\ominus$ while $\cup$ is a staightforward union of all case partitions of two statements.

4.2.2 FOMDP representation

Stochastic action representation

Building on the foundation of the situation-calculus for first-order deterministic planning, we generalize this representation to handle stochastic actions in an MDP framework. To specify stochastic actions, we use actions that decompose into deterministic actions under nature's control [3], e.g. we may choose a stochastic action such as $Load(box, truck)$ in some state but then according to some prespecified probability distribution (possibly conditioned on properties of the state), nature will choose the action

$LoadSuccess(box, truck)$ or $LoadFailure(box, truck)$.

Consequently, for a stochastic version of the situation calculus, we require a probability distribution over the decomposition of a stoachastic action into nature's actions. We can specify this by partitioning first-order state space into cases:

$$P(LoadSuccess(box, truck) \mid Load(box, truck), s) = [Raining(s) : 0.7 \; ; \; \neg Raining(s) : 0.9]$$
$$P(LoadFailure(box, truck) \mid Load(box, truck), s) = [Raining(s) : 0.3 \; ; \; \neg Raining(s) : 0.1]$$

We will refer to this probabilistic case statement subsequently using the $pCase(n(\vec{x})|A(\vec{x}), s)$ where $n(\vec{x})$ is the probability of nature's action choice for stochastic action $A(\vec{x})$ in state s. Since the successor state axiomatization of a domain is specified for nature's actions only, they are unaffected by the generalization of the situation calculus to stochastic domains. Additionally, we note that since nature's distribution over action choices depends only on the current state s, this representation is Markovian.

Reward and state representation

When solving a FOMDP, our goal will be to derive a first-order value function. Because we will determine this value function using regression techniques, we don't need an initial state - this algorithm will provide the optimal value function and policy for *every* state in terms of a first-order case partitioning. We can represent our value and reward functions as case statements, e.g., a reward function could be specified as the following:

$$\begin{aligned} case \quad [\quad & (\exists b).BIn(b, Paris, s) \wedge TypeA(b) : 10 \; ; \\ & \neg((\exists b).BIn(b, Paris, s) \wedge TypeA(b)) \wedge (\exists b).BIn(b, Paris, s) : 5 \; ; \\ & \neg(\exists b).BIn(b, Paris, s) : 0 \;] \end{aligned}$$

This reward states that one receives a reward of 10 for having a box of type A in Paris, one receives a reward of 5 for having any box in Paris not of type A, and one receives a reward of 0 for any other state. We note that this first-order partitioning is mutually exclusive and exhaustive, thus we must take care to prevent partitions from overlapping.[5] Also, we note that since our first iteration of value iteration uses the reward function as $V^0(s)$, this representation is also the general format for our value function. We refer to the reward and value case statements as $vCase(s)$ and $rCase(s)$ to show the explicit dependence of the reward and value functions on the state s.

[5] This accounts for the extremely long formula resulting in reward 5 that essentially states, if we don't satisfy the situation where we received a reward of 10, we can still get a reward of 5 for have a box in Paris not of type A.

4.2.3 Dynamic programming for FOMDPs

Our next step is to generalize dynamic programming methods to efficiently solve for FOMDP value functions. We use a value iteration approach and separate the dynamic programming backup into two steps: regression and maximization. Then the overall algorithm is the same as for value iteration: We continue performing the DP backup until the maximum Bellman error between iterations (easily computed by $vCase^i(s) \ominus vCase^{i-1}(s)$) ensures ϵ-optimality. See the section on value iteration in MDPs for details on the termination criteria.

First-order decision theoretic regression

On each regression step during value iteration, we want to determine the value of an action. In doing so, we generalize equation 7 which provided a definition for i-state-to-go value function. We replace a in the original equation with the stochastic action $A(\vec{x})$ parameterized by *free* variables to obtain the following definition:

$$Q^i(A(\vec{x}), s) = R(s) + \gamma \cdot \left\{ \sum_{t \in S} P(t|A(\vec{x}), s) \cdot V^{i-1}(t) \right\} \tag{16}$$

Through a sequence of steps, we 1) replace the probabilities with their decomposition into each of nature's j action choices $n_j(\vec{x})$, 2) we replace the reward and value functions with their case representations defined previously, 3) and we regress $V^{i-1}(s)$ through nature's action choice since it is in the successor state and we need an expression in terms of the current state $V^i(s)$. We arrive at the following expression for determining the one-step backup Q-function for a FOMDP:

$$Q^i(A(\vec{x}), s) = rCase(s) \oplus \gamma \cdot \{\oplus_j \ pCase(n_j(\vec{x})|s) \otimes Regr(vCase^{i-1}(do(n_j(\vec{x}), s)))\} \tag{17}$$

Symbolic dynamic programming

After we've regressed each action, we need to maximize over each of the action Q-functions in order to derive the maximum value that could be achieved from each state:

$$V^i(s) = \max_{a \in A} \left\{ Q^i(s, a) \right\} \tag{18}$$

To generalize this to the first order case, we note that the Q-functions can be represented as case partitions $qCase^i_{A(\vec{x})}(s)$. However, these Q-functions are defined in terms of the free variables in the action bindings. Instead, we need a Q-function that states, if an action exists that causes this state partition to hold, we can achieve the given value. Consequently, we need to existentially quantify the action parameters of each Q-function and combine them all into a single value function that chooses

a maximizing action for all partitions of state space. By abusing notation slightly, we arrive at the following definition for an i-stage-to-go value function:

$$vCase^i(s) \equiv \exists A, \vec{x}\; qCase^i_{A(\vec{x})}(s) = t_1 \wedge \forall A, \vec{x}\; qCase^i_{A(\vec{x})}(s) = t_2 \supset t_1 \geq t_2 \tag{19}$$

In practice, we can determine partitioning that satisfies this definition by ordering the partitions of the union of the existentially quantified Q-functions for each stochastic action by decreasing value, and ensuring that the conditions for the first n partitions are negated in the partition formula for the $n+1^{\text{st}}$ partition. This ensures that the highest value is assigned to every state and results in an optimal, exclusively partitioned first-order value function.

General remarks

In concluding this section on first-order MDPs, we note that this framework offers many attractive properties from an MDP perspective. First, it allows one to concisely represent PSTRIPS and PPDDL domains in a first-order domain. Second, its solution algorithms do not require explicit state and action enumeration and therefore can solve for very concise representations of value functions when they exist. This latter property enables symbolic dynamic programming to solve for value functions in domains with potentially infinitely many objects. On the other hand, this expressivity comes with the drawback that theorem proving methods are required to fully simplify the Q and value function representations. Such simplification is infeasible in the general case, and it is an open question as to whether acceptable performance can be attained through simplification methods that are strictly less powerful than first-order theorem proving.

5 Hierarchy and program constraints

Our previous approaches to decision-theoretic planning have made two assumptions:

1. We are planning at one level searching the entire state space (even if factored) and using all possible actions.

2. We are allowing any action to be executed at any point during planning or execution.

In making these assumptions, we are often presenting ourselves with a planning problem where much of the state space could be abstracted with little or no impact on the quality of the value function. Furthermore, by considering every action at every step, we are often doing repetitive search that could easily be compiled into a macro-action allowing us to search in an abstracted, higher-level state space. Addressing both of these issues is an important issue for scalability in MDPs and we discuss enhancements along both of these lines in the following sections.

5.1 Hierarchy and program constraints in MDPs

5.1.1 Subtasks and weakly coupled MDPs

If a problem domain consists of many independent subprocesses that only interact via their dependence on globally shared resources, one can often factor these MDPs into tasks represented as independent subMDPs whose actions are globally constrained. This model is referred to as a Markov task set (MTS) [32].

One can solve MTS problems by solving the equivalent joint MDP formed by taking the cross-product of all state and action spaces for each of the task subMDPs, but obviously the MDP representation will grow exponentially with the number of tasks and thus becomes intractable for all but the smallest problems. A more tractable approach is known as Markov Task Decomposition (MTD).

MTD is an approximately optimal approach to solving the joint MDP that divides the solution into local and global optimization steps. MTD first determines the optimal value function for each subtask. Following this local optimization, a global optimization phase then chooses a joint action at each time step that enforces the global resource constraint while trading off local action choices for each task in order to maximize the expected reward. Since an optimal sequential solution in this case would be equivalent to solving the full joint MDP, one is usually restricted to both heuristic and myopic solution techniques. Nevertheless, a good choice of heuristic allocator has allowed for near-optimal and extremely efficient solutions to these problems when compared to standard dynamic programming over the full joint MDP.

While the MTD approach provides an attractive method for approximately solving problems that can be formulated as MTSs, we note that the MTS model is severely restrictive in that it only allows interaction between tasks via shared resources. Additionally, we note that the use of heuristic and myopic methods in the MTD approach prevent us from obtaining optimality guarantees for the resulting policy.

5.1.2 Semi-MDPs and options

Sutton et al. [43] introduced a temporally extended action definition known as an option based on the semi-MDP model. In short, rather than restricting actions to primitives that complete in exactly one time step, we now allow for a more general model. Options consist of three components: a policy $\pi : S \times A \to [0, 1]$, a termination condition $\beta : S^+ \to [0, 1]$ for the subset of reachable states S^+ in this option, and an entry or initiation state set $I \subseteq S$. Consequently, an option corresponds to a policy that can be initiated from any state in I and which terminates in any reachable state s for which $\beta(s) = 1$. An option will obviously require as many time steps as it needs to reach a terminal state starting from an initial state while executing a given policy. And since MDP transitions are stochastic, this number of time steps may vary from execution to execution.

The one difficulty with handling multi-time-step actions stems from the need to apply a discount on

the reward for every step taken. If we simply treat a macro-action as a primitive action and discount it by γ as usual, we are ignoring the fact that it may have taken 1,2 or in general n time-steps. Certainly we don't want to calculate the discount for a 20 time-step action the same way we handle a discount for a single step action. Thus, we need to introduce a framework that can handle the discount for macro-actions in a correct manner.

The Semi-MDP framework allows one to do this, but rather than discuss the option model in the context of Semi-MDPs, we defer to the next section where we show that we can compile a Semi-MDP with options (a.k.a. macro actions) into an MDP with non-Markovian policies, i.e., the policy may depend upon both the current state *and* the entry state for the option being currently executed.

5.1.3 Macro-actions and MDP abstraction

The idea of subdividing (or hierarchically subdividing) an MDP into weakly coupled components has been covered by Hauskrecht et al [22] and Parr [33]. These papers cover a number of automated methods for constructing these macros that range from using heuristic information concerning the value of macro exit states to piece together policies for local MDPs to methods for improving the global policy based on various macro policy error-analysis and improvement techniques. We note that these techniques can yield ϵ-optimal global policies.

As before, we can think of a macro-action m as a policy in a subMDP: We have a set of entrance states, a set of reachable states, and a subset of the reachable states which are termination states where we exit the macro. Then our macro is just a policy π_m to follow while in this region.

From the perspective of the MDP that invoked the macro m, we can actually think of this macro as a primitive action if we modify the transition and reward dynamics to implicitly take the discount factor into account. Letting s be any start state for a macro m, s' be any of m's termination states, and taking an expectation with respect to the time of termination τ, we can define the discounted transition function for this macro as follows:

$$\begin{aligned} T(s, \pi_m, s') &= E_\tau[\gamma^{\tau-1} \cdot P(s^\tau = s' | s^0 = s, \pi_m)] \\ &= \sum_{t=1}^{\infty} \gamma^{t-1} \cdot P(\tau = t, s^t = s' | s^0 = s, \pi_m) \end{aligned}$$

And we can define the discounted reward function for this macro as the following:

$$R_m(s, \pi_m) = E_\tau \left(\sum_{t=0}^{\tau} \gamma^t R(s^t, \pi_m(s^t)) | s^0 = s, \pi_m \right)$$

These are both linear systems that can be solved directly or through iterative means. If $s \in S_i$, this computation requires time $O(|S_i|^3)$.

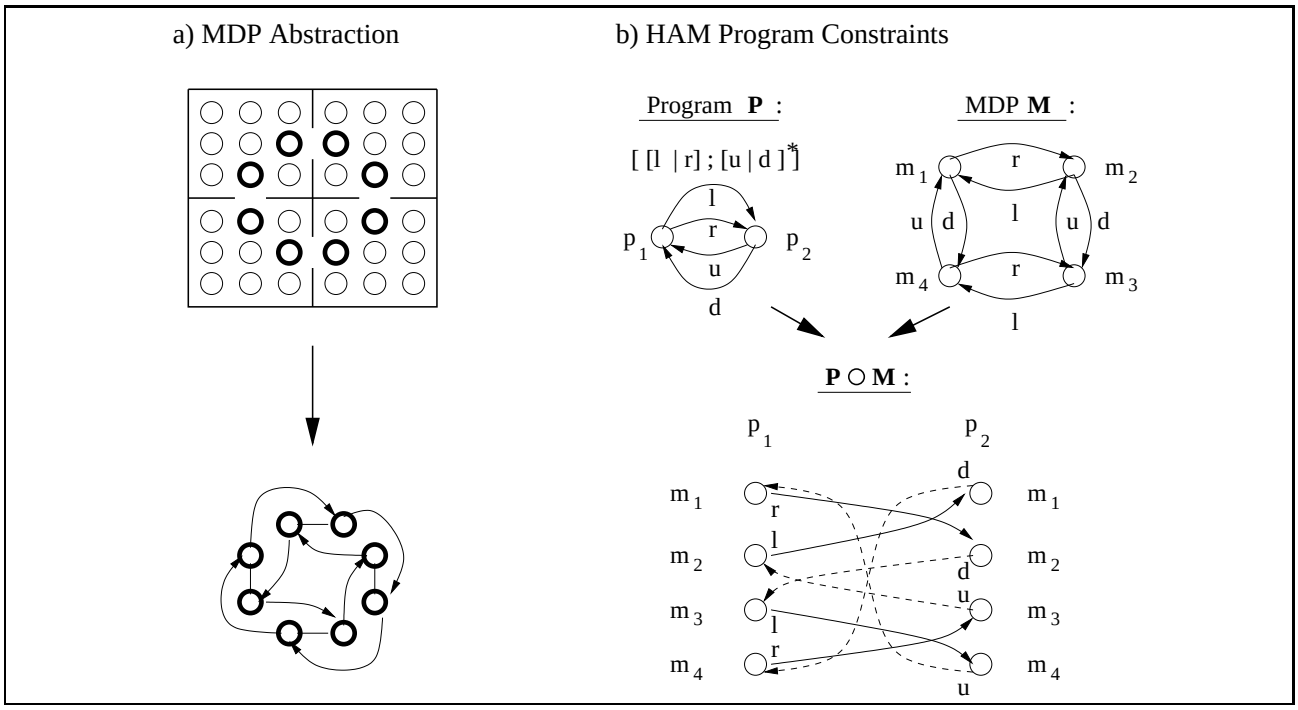

Figure 5: a) An example of partitioning a large MDP state space into independent partitions with entry and exit states. The action arcs in the abstracted MDP conform to local policies for their respective regions of state space. b) A small example of the HAM model by applying program constraints P to an MDP M. The resulting composition $P \circ M$ has more states (i.e. $|P \times M|$), but the transition model is a constrained subset of the original (cooresponding to the program constraints) that should be more efficient to solve.

We note that when forward-searching or regressing through this macro transition and reward model, we will be implicitly taking into account the expected sum of discounted reward for *all* possible execution trajectories: The expected reward accumulated while executing the macro is handled via the *discounted* reward function. And the reward received after the macro is handled via the *discounted* transition model.

Additionally, we note that when executing this macro, we will actually be following the policy π_m until it terminates. This means that our high-level policy representation is no longer Markovian – it is dependent on the entry state of the macro being executed as well as the current state being evaluated.

In using macro actions to solve problems, we can *augment* the original action space with these macros. Macros may speed the convergence of a solution since forward-search or regression through a single macro propagates value that is equivalent to forward-searching or regressing through many actions over a large time scale [43]. However, if the initial value function V^0 is an upper bound on V^*, then value iteration in the augmented action space requires at least the same number of iterations for convergence as the original MDP [22].

However, if we are willing to tradeoff policy quality for computational savings[6] we can disregard some of the primitive actions and abstract to a higher-level MDP, one that only mentions the entrance and exit states of the macros (and primitive actions) that are retained. An example of this type of MDP abstraction is shown in Figure 5a. In doing this, we see that abstraction can yield much more compact MDPs (and therefore fewer states and actions to iterate over) with the drawback that we are limited to the subset of policies that can be constructed from our macro set. Nonetheless, if we choose these macros well, this can make solving very large MDPs much more efficient than solving the original MDP. While we previously discussed a few macro construction techniques that allow us to attain an ϵ-optimal policy, there are many open questions such as how to use MDP structure to infer macros, or how to compactly construct macros by exploiting their structure.

5.1.4 Hierarchies of abstract machines

Parr [34] defines a model known as Hierachies of Abstract Machines (HAMs) that introduces a method for solving an MDP under program constraints. This differs from the previously discussed options and macro-actions in that we are directly restricting the global policy as opposed to restricting the set of policies to compositions of macro actions. Yet in a general sense, these models are actually very similar in spirit - both abstract an MDP by restricting the set of allowable policies. In this case, a program can be thought of as a mechanism for restricting the set of allowable actions given a program state. Consequently, one can think of a HAM as a GOLOG-like constraint on MDP action selection such that at non-deterministic choice points, we want to determine the decision-theoretically optimal action. We note that these HAMS can be hierarchical in the same way that one GOLOG program may call another. The PHAM formalism [1] provides a similarly motivated LISP-like language with parameterized procedures for defining program constraints.

When solving a HAM, one can compose an augmented state space $P \circ M$ consisting of the cross-product of the program state and the MDP state. An example of this type of model composition is shown in Figure 5b. Clearly, if one can derive the optimal policy at each of these augmented states then one has an optimal solution to the MDP under the given program constraints. While it would seem that the cross-product state space would be larger and therefore more difficult to solve, solving this constrained MDP is often easier than the unconstrained MDP since there are *typically* fewer choice points.

These ideas will prove extremely useful when we generalize them to solve FOMDPs with GOLOG program constraints (i.e. DT-GOLOG). However, as is the case for program macros, this body of work is largely disconnected from the other work involving MDP structure. One exception to this observation is the work of Andre and Russell [2] that combines state abstraction techniques with the PHAM model.

[6] Sometimes we can discard actions without sacrificing policy quality, but this is not always the case.

5.1.5 MAXQ hierarchical reinforcement learning

While hierarchical reinforcement learning (RL) is not something that we are addressing specifically in this survey, Dietterich's [15] MAXQ hierarchical RL is based on an MDP macro-action model that combines elements of both MDP abstraction and HAMs thus warranting a brief mention.

In the MAXQ model, one is given a recursive task decomposition that can be thought of as a set of program constraints with potentially recursive subtasks. The goal is to learn the optimal action policy for the choice points in this task hierarchy via the Q-learning algorithm [4][7] At the base of this hierarchy are primitive actions with primitive Q-functions $Q(a, s)$ whose value is defined as usual. All other levels of the task hierarchy are defined with subtask Q-functions $Q(p, s, a)$ that are augmented with the task p since the policy while executing a subtask is both state and subtask dependent (i.e., non-Markovian). $Q(p, s, a)$ is the expected discounted future reward for executing p (including subtasks) until termination. Dietterich shows that finding the optimal Q-functions according to these definitions allows one to determine a recursively optimal policy.[8]

We briefly note that each subtask in the task network conforms to a macro-action and thus we see that the MDP abstraction and HAM models can be seamlessly merged.

5.2 Hierarchy and program constraints in FOMDPs

5.2.1 DT-GOLOG

Building on the ideas presented in the last section, we finish our survey of current work with perhaps the most expressive vertex of the representational cube provided in Figure 2: first-order MDPs with program constraints.

In this framework, our goal is to generalize GOLOG to a decision-theoretic framework where we can explicitly determine a policy at non-deterministic choice points that maximizes the expected sum of discounted reward. This framework is known as DT-GOLOG [13] and it allows a programmer to partially specify a high-level program while leaving the program intepreter to determine the optimal actions for the choices that remain.

To formalize DT-GOLOG, we simply need to generalize the $Do(\delta, s, s')$ macro to a decision-theoretic version $BestDo(\delta, s) \rightarrow \mathbb{R}$.[9] Whereas $Do(\delta, s, s')$ evaluated to true or false based on whether the program could be executed, we now assume that every action/macro can be executed from every state to allow $BestDo$ to be treated as a function. This is easy to do by folding the preconditions into the positive and negative effect axioms, thus leading to a NOOP when the action cannot be executed. The $BestDo$

[7] However, we can easily generalize any model-based MDP solution algorithm to handle this model as well.

[8] Optimal with respect to the state and subtasks available at each recursion level, assuming the subtasks themselves are recursively optimal.

[9] For the following discussion, we slightly abuse notation in order to use an intuitive notation that draws a connection with previously discussed work.

function returns the expected sum of discounted reward that can be achieved by executing a GOLOG program δ starting from s.

Executing a macro action returns a reward and it also has an associated (perhaps multi-time-step) transition distribution. We can compute both the expected transition and reward functions as we did previously by generalizing these methods to a first-order state space. Additionally, the two sources of non-deterministic choice can now be formalized in a decision theoretic manner using Q-functions:[10]

- *DT nondeterministic program choice*: $Q(s, BestDo([\delta_1|\delta_2])) = \max\{Q(s, BestDo(\delta_1)) \ ; \ Q(s, BestDo(\delta_2))\}$

  The action executed corresponds to the one offering the maximal expected reward from situation s. The policy for this macro-action is simply a state partitioning mapping to the program to be executed. We note that this is what the maximization step is computing in symbolic dynamic programming for FOMDPs.

- *DT nondeterministic choice of arguments*: $Q(s, BestDo((\pi x)\delta(x)) = \max\{\exists x \ Q(s, BestDo(\delta(x)))\}$

  The action executed here is the one that maximizes the $\delta(\cdot)$ binding. The policy for this macro-action is again a state partitioning mapping to the action bindings that yield optimal execution. We note that this non-deterministic choice is what we are computing for all states when performing action regression in symbolic dynamic programming for FOMDPs.

We would have to appropriately define the Q-function distributions for all of the DT-GOLOG program constructs, but this perhaps gives one a flavor for the DT-GOLOG extension.

We note that in practice, the original version of DT-GOLOG [13] was used in an off-line forward-search MDP solution algorithm. In an empirical comparison to unconstrained search, the DT-GOLOG approach allowed search to a considerably deeper horizon. Later work [41] took a more RTDP-style approach to on-line search for the optimal action and execution. However, we note that these search-based approaches are not as efficient as dynamic programming approaches if a large percentage of the state-space is reachable. Consequently, a better approach to solving FOMDPs under GOLOG program constraints may be to do dynamic programming in this framework; This will be discussed in the future work.

6 Future directions

Here we outline some future directions for research in the areas of FOMDPS and DT-GOLOG.

[10] This is a manner somewhat similar to Dietterich's QMAX notation. We will diverge from the actual notation used in the original DT-GOLOG paper [13] for simplicity of exposition.

6.1 First-order ADDs and approximation

Just as we used ADDs for efficient representation of context-specific structure in propositional MDPs, it is interesting to note that we can generalize to first-order ADDs and use similar techniques in the first-order case. These techniques seem to hold promise since FOMDP value functions often have a great deal of redundant first-order structure. If this redundant structure can be minimized by using a decision-diagram structure, then we can efficiently represent the policy while saving redundant computation. This is the subject of current research.

6.2 First-order basis functions and approximation

When it is impossible or intractable to derive an exact value function via the above methods, we can generalize the propositional idea of a basis function to the first order case. In this case, our value function would be represented as a sum over potentially weighted $vCase(s)$ partitions. This would allow us to exploit additive structure in FOMDP value functions. The major challenge with this paradigm is selecting a good set of first-order basis functions. In addition to what relations to choose, we now have the question of how to quantify and specify variables used in these first-order basis functions – this adds an additional layer of complexity that was not present for propositional basis function selection. However, the structure of the FOMDP (e.g., the transition dynamics and the successor state axioms) may give us some hints concerning useful basis functions.

6.3 DBN factored action decomposition

Our current decomposition of nature's choice actions from user actions is specified in a non-factored manner. For example, if we can execute parallel actions in the blocks world, e.g., $stack(a, b)$ and $stack(c, d)$ where a, b, c, d are all pairwise unequal, then we might expect that the result of performing the first stack action will not affect the second. Under the current FOMDP model, we would need to treat this parallel action as a single stochastic action decomposing into a joint nature's choice action over the cross product of individual outcomes. But if the effects are independent, then we could compute the result in a factored manner and avoid this joint computation. Thus, one improvement would be to look at a factored action effect model based on a DBN. This would allow for a considerable increase in computational efficiency if we allow parallel non-interfering actions or if actions have multiple independent effects.

6.4 FODTR with domain constraints

While solving a FOMDP can be difficult because we are implicitly solving for all domains, we can restrict ourselves to a specific domain and use model-checking techniques to perform simplification.

Typically, we use first-order theorem proving or analogous methods to perform simplification of the first-order formula generated by value iteration. However, if we have known domain constraints, we can use a model-checking approach for determining inconsistency, subsumption, and tautology in the simplification process. While this would restrict the policy optimality to the given domain constraints, it would make the simplification task much more feasible to implement in practice.

6.5 FODTR with GOLOG program constraints

We note that GOLOG program constraints can be applied to dynamic programming/regression solution methods. Such an extension involves an application of the HAM model using GOLOG as a specification of program constraints and a FOMDP for the state and transition model. While the presence of GOLOG constructs such as *while* loops make this task non-trivial, a dynamic programming solution to FOMDPs under GOLOG program constraints would likely be much more efficient than the forward-search or RTDP approaches discussed previously.

6.6 FOMDP decomposition and abstraction

As opposed to solving a FOMDP under program constraints, we can also partition state space and learn policies for each partition and entry-state as described previously. This approach would result in an abstracted FOMDP that could be solved more efficiently than the original, albeit optimal only with respect to the restricted policy space. The deterministic version of such a model has been explored by McIlraith and Fadel [31].

6.7 First-order count aggregators

Many domains use an explicit representation of count and even make transitions or rewards dependent upon this count. Consequently, it is interesting to consider augmenting our first-order domain language to handle explicit counts. One could do this via $\exists_n$ macro extensions or via $\#_n$ count aggregators.

References

[1] David Andre and Stuart Russell. Programmable reinforcement learning agents. In *In Advances in Neural Information Processing Systems*, volume 13, 2001.

[2] David Andre and Stuart Russell. State abstraction for programmable reinforcement learning agents. In *In Proc. AAAI-02*, Edmonton, Alberta, 2002. AAAI Press.

[3] Fahiem Bacchus, Joseph Y. Halpern, and Hector J. Levesque. Reasoning about noisy sensors in the situation calculus. In *IJCAI 95*, pages 1933–1940, Montreal, 1995.

[4] Andrew Barto and Richard Sutton. *Reinforcement Learning.* MIT Press, 1998.

[5] Andrew G. Barto, Steven J. Bradtke, and Satinder P. Singh. Learning to act using real-time dynamic programming. Technical Report UM-CS-1993-002, U. Mass. Amherst, , 1993.

[6] Richard E. Bellman. *Dynamic Programming.* Princeton University Press, Princeton, NJ, 1957.

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

[8] Craig Boutilier, Thomas Dean, and Steve Hanks. Planning under uncertainty: Structural assumptions and computational leverage. In *Proceedings of the Third European Workshop on Planning*, Assisi, Italy, 1995.

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

[10] Craig Boutilier, Richard Dearden, and Moisés Goldszmidt. Stochastic dynamic programming with factored representations. *Artificial Intelligence*, 121:49–107, 2000.

[11] Craig Boutilier, Nir Friedman, Moisés Goldszmidt, and Daphne Koller. Context-specific independence in Bayesian networks. In *UAI 96*, pages 115–123, Portland, OR, 1996.

[12] Craig Boutilier, Ray Reiter, and Bob Price. Symbolic dynamic programming for first-order MDPs. In *IJCAI 01*, pages 690–697, Seattle, 2001.

[13] Craig Boutilier, Ray Reiter, Mikhail Soutchanski, and Sebastian Thrun. Decision-theoretic, high-level agent programming in the situation calculus. In *AAAI 00*, pages 355–362, Austin, TX, 2000.

[14] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[15] Thomas G. Dietterich. The MAXQ method for hierarchical reinforcement learning. In *ML 98*, pages 118–126, Madison, WI, 1998.

[16] Kutluhan Erol, James Hendler, and Dana S. Nau. HTN planning: Complexity and expressivity. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, volume 2, pages 1123–1128, Seattle, Washington, USA, 1994. AAAI Press/MIT Press.

[17] Alan Fern, SungWook Yoon, and Robert Givan. Approximate policy iteration with a policy language bias. In *Advances in Neural Information Processing Systems 16 (NIPS-2003)*, Vancouver, 2003.

[18] Richard E. Fikes and Nils J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *AI Journal*, 2:189–208, 1971.

[19] M. Fox and D. Long. Pddl2.1: An extension to pddl for expressing temporal planning domains., 2001.

[20] Carlos Guestrin, Daphne Koller, Chris Gearhart, and Neal Kanodia. Generalizing plans to new environments in relational MDPs. In *IJCAI 03*, Acapulco, 2003.

[21] Carlos Guestrin, Daphne Koller, and Ronald Parr. Max-norm projections for factored MDPs. In *IJCAI 01*, pages 673–680, Seattle, 2001.

[22] Carlos Guestrin, Daphne Koller, Ronald Parr, and Shobha Venktaraman. Efficient solution methods for factored MDPs. *JAIR*, 2002.

[23] Milos Hauskrecht, Nicolas Meuleau, Leslie Pack Kaelbling, Thomas Dean, and Craig Boutilier. Hierarchical solution of Markov decision processes using macro-actions. In *UAI 98*, pages 220–229, Madison, WI, 1998.

[24] Jesse Hoey, Robert St-Aubin, Alan Hu, and Craig Boutilier. SPUDD: Stochastic planning using decision diagrams. In *UAI 99*, pages 279–288, Stockholm, 1999.

[25] Daphne Koller and Ronald Parr. Computing factored value functions for policies in structured MDPs. In *IJCAI 99*, pages 1332–1339, Stockholm, 1999.

[26] Daphne Koller and Ronald Parr. Policy iteration for factored MDPs. In *UAI 2000*, pages 1332–1339, Stockholm, 1999.

[27] Hector J. Levesque, Ray Reiter, Yves Lespérance, Fangzhen Lin, and Richard Scherl. GOLOG: a logic programming language for dynamic domains. *Journal of Logic Programming*, 31(1-3):59–83, 1997.

[28] Michael L. Littman. Probabilistic propositional planning: Representations and complexity. In *AAAI 97*, pages 748–754, Providence, RI, 1997.

[29] Michael L. Littman, Judy Goldsmith, and Martin Mundhenk. The computational complexity of probabilistic planning. *Journal of Artificial Intelligence Research*, 9:1–36, 1998.

[30] J. McCarthy. Situations, actions and causal laws. Technical report, Stanford University, 1963. Reprinted in Semantic Information Processing (M. Minsky ed.), MIT Press, Cambridge, Mass., 1968, pages 410-417.

[31] D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins. PDDL—The planning domain definition language, 1998.

[32] Sheila McIlraith and Ron Fadel. Planning with complex actions. In *Proceedings of the Ninth International Workshop on Non-Monotonic Reasoning (NMR2002)*, pages 356–364, 2002.

[33] Nicolas Meuleau, Milos Hauskrecht, Kee-Eung Kim, Leonid Peshkin, Leslie Pack Kaelbling, Thomas Dean, and Craig Boutilier. Solving very large weakly coupled Markov decision processes. In *AAAI 98*, pages 165–172, Madison, WI, 1998.

[34] Ron Parr. Flexible decomposition algorithms for weakly coupled markov decision problems. In *UAI 98*, 1998.

[35] Ronald Parr and Stuart Russell. Reinforcement learning with hierarchies of machines. In M. Kearns M. Jordan and S. Solla, editors, *Advances in Neural Information Processing Systems 10*, pages 1043–1049. MIT Press, Cambridge, 1998.

[36] Pascal Poupart, Craig Boutilier, Relu Patrascu, and Dale Schuurmans. Piecewise linear value function approximation for factored MDPs. In *AAAI 02*, pages 292–299, Edmonton, 2002.

[37] Pascal Poupart, Relu Patrascu, Dale Schuurmans, Craig Boutilier, and Carlos Guestrin. Greedy linear value-approximation for factored Markov decision processes. In *AAAI 02*, pages 285–291, Edmonton, 2002.

[38] Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming.* Wiley, New York, 1994.

[39] Raymond Reiter. The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. In V. Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation (Papers in Honor of John McCarthy)*, pages 359–380. Academic Press, San Diego, 1991.

[40] D. Schuurmans and R. Patrascu. Direct value-approximation for factored mdps. In *In Advances in Neural Information Processing 14 (NIPS*2001)*, 2001.

[41] Dale Schuurmans and Relu Patrascu. Direct value approximation for factored MDPs. In *Advances in Neural Information Processing Systems 14 (NIPS-2001)*, Vancouver, 2001. to appear.

[42] Mikhail Soutchanski. An on-line decision-theoretic golog interpreter. In *IJCAI-2001*, Seattle, Washington, 2001.

[43] Robert St-Aubin, Jesse Hoey, and Craig Boutilier. APRICODD: Approximate policy construction using decision diagrams. In *Advances in Neural Information Processing Systems 13 (NIPS-2000)*, pages 1089–1095, Denver, 2000.

[44] Richard S. Sutton, Doina Precup, and Satinder P. Singh. Between MDPs and Semi-MDPs: Learning, planning, and representing knowledge at multiple temporal scales. *Artificial Intelligence*, 112:181–211, 1999.

[45] D. S. Weld. An introduction to least commitment planning. *AI Magazine*, Winter 1994:27–61, 1994.

[46] Daniel S. Weld. Recent advances in AI planning. *AI Magazine*, 20(2):93–123, 1999.

[47] SungWook Yoon, Alan Fern, and Robert Givan. Inductive policy selection for first-order markov decision processes. In *UAI 02*, Edmonton, 2002.

[48] Hakan Younes and Michael Littman. PPDDL: The probabilistic planning domain definition language, 2004.