# Leveraging and Learning Propositional Functions for Large State Spaces in Planning and Reinforcement Learning

D Ellis Hershkowitz

Under Professor Stefanie Tellex

Brown University, Computer Science Department
115 Waterman Street, 4th floor Providence, RI 02912

April 15, 2015

**Abstract**

Planning in large state spaces is a normally computationally infeasible task that crops up in many practical robotics problems. I present a means of leveraging propositional functions in object-oriented Markov Decision Processes (OO-MDPs) to prune state spaces which yields both significant computation speedups as well as generally better plans when applied to baseline planners. Modern reinforcement learning (RL) solutions also make critical use of propositional functions. Given the importance of propositional functions in the presented pruning work and modern RL algorithms and the fact that propositional functions are often difficult to specify by hand, I also present a means by which useful propositional functions can be inferred based on observations regarding transition dynamics. I then use these propositional functions to derive a novel version of deterministic object-oriented RMAX (DOORMAX) which requires no expert-provided propositional functions. Lastly, not only do I demonstrate high correspondence between these learned propositional functions with expert-provided propositional functions but I also demonstrate near classic DOORMAX-level performance of my novel DOORMAX algorithm.

# Contents

Thanks to Stefanie for her unbridled enthusiasm and wisdom, James for his unbridled patience and wisdom, Dave as well as the other authors on [1]. Also thanks to James for his nifty planning and reinforcement learning platform, BURLAP (`http://burlap.cs.brown.edu/`) and thanks to Michael Littman for agreeing to read this thesis.

# 1 Introduction

Planning in large stochastic state spaces is normally a prohibitively difficult task. Stochastic planning problems consist of applying a model of how actions stochastically transition one from state to state in order to discover a series of actions that maximizes some received reward. However, large stochastic state spaces are often mired by swaths of irrelevant subspaces into which planners needlessly sink gobs of computation. Often so much computation is wasted that the problem is rendered computationally intractable for all intents and purposes. This intractability would hardly be a problem if there weren't a great deal of practical problem formalizations that require planning and acting in large state spaces.

However, practical robotics tasks are riddled with stochastic large state spaces [18, 4, 1]. For instance, in pick and place tasks not only does the size of the state space increase exponentially with the number of objects to be manipulated but the robot must grapple with noise in its control; consequently, a task with a meager number of objects is characterized by an extremely large stochastic state space. Similarly, a robot engaged in a cooking task is capable of configuring the layout of its ingredients in numerous ways with some failure probability and so it too must grapple with a large stochastic state space [1]. There is, then, a great deal of motivation in manufacturing methods by which planning problems in large stochastic state spaces can be expediently solved.

To this end I present a methodology as developed by work I was involved in [1] to rein in the computational difficulties of planning in hefty stochastic state spaces in Section 3. This methodology exploits propositional functions in OO-MDP problem representations (see Section 2.6 for more on OO-MDPs) to prune states unlikely to be involved in optimal behavior. We (i.e. the authors of [1]) empirically demonstrate that this pruning greatly reduces planning time and computation all the while conferring more optimal behavior in complex domains.

However, planning is often impossible since a model of the effects of actions may well be obfuscated to the agent. For instance, the underlying physics that dictate a pick and place task are almost always unknown to the involved robot if for no reason other than the difficulty of supplying a correct and robust physics model.

Traditional reinforcement learning (RL) algorithms offer solutions to simultaneous reward maximization and model learning but these solutions fail to scale well to large state spaces. RL takes it upon itself to simultaneously learn problem mechanics while maximizing reward by observing the results of taking actions for various states. However, traditional RL algorithms like RMAX (see Section 2.5.2) are highly tabular: that is, they learn information specific to individual states. As a result they flounder in the face of exceedingly large state spaces wherein not every state can be visited under reasonable computation constraints. What would be superior for an RL framework is an algorithm which rapidly transfers knowledge across similar states.

More modern approaches like deterministic object-oriented RMAX (DOORMAX)[11] (see Section 4.2) and linear function approximation methods for value functions[14] do exactly this sort of state to state knowledge transfer by exploiting propositional functions. That is, these methods are roughly characterized by using propositional functions to quantify the similarity of states and then transferring model knowledge as appropriate.

Thus, propositional functions are of critical importance in planning and reinforcement learning techniques for large state spaces: they both allow us to accelerate stochastic planning as demonstrated by the work presented in Section 3 and they expedite modern RL techniques. If we were capable of fully specifying all useful propositional functions all would be good and well. However, the space of propositional functions is immense and so we can reasonably anticipate the difficulty of specifying all useful propositional functions for complex problems.

To address the impetus for useful propositional functions in planning and reinforcement learning problems I derive a means of inferring propositional functions in Section 4. These propositional functions are derived based on observations regarding transition dynamics. Additionally, as I demonstrate, they readily replace those propositional functions normally expert-supplied to DOORMAX. I confirm empirical alignment between learned propositional functions and those propositional functions normally expert specified. Lastly, I

also show that this novel version of DOORMAX performs nearly as well as classic DOORMAX in terms of reward maximization over time.

However, before moving to these original contributions I use Section 2 to cover relevant preliminaries in planning and reinforcement learning.

# 2 Preliminaries

The following sections consist of relevant background knowledge in AI. It is my hope that anyone with a basic knowledge of discrete math and probability will be capable of comprehending the entirety of the section and this will, in turn, allow the reader to fully comprehend Sections 3 and 4. In any case, any subsection of what follows is readily skipped by any reader who recognizes the titles of the section or else equivalently one who has an incisive disdain for learning.

## 2.1 Relevant AI Problems

Large swathes of artificial intelligence are concerned with optimal behavior in various problems. Below are natural language descriptions of relevant problems in artificial intelligence.

### 2.1.1 Grid World

Grid world consists of a two-dimensional world discretized by cells. The agent occupies a single cell and can move either north, east, south or west in order to move itself to the appropriate adjacent cell. There are also walls which occupy some cells which block the agent's movement action if the agent tries to move into them. The problem is usually considered solved when the agent arrives in a goal cell which is generally in the top right corner of the world. There are usually 4 "rooms" delineated by walls that the agent may occupy.

Figure 1 visualizes an instance of this world. Black squares correspond to walls, the blue square indicates the goal location of the agent and the grey circle corresponds to the agent. There are also implicit walls along the edge of the map.

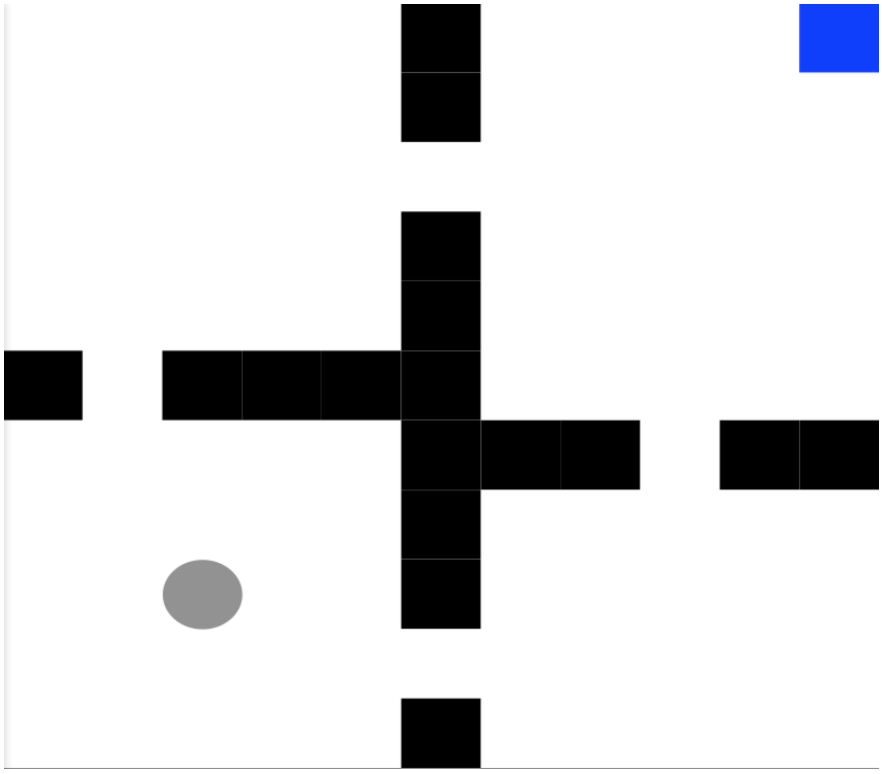


Figure 1: An example of a grid world problem

### 2.1.2 Taxi

Taxi problems[10] consists of an agent that, like the grid world agent, can move north, east, south and west in a two-dimensional grid world, wherein movement actions may be thwarted by walls. However, in taxi problems walls occupy space between cells rather than cells themselves and there are also passengers each of which occupy a single cell which the agent can pick up or drop off – when a passenger is picked up they move with the agent until dropped off. Also note that the agent can usually only carry a single passenger. The problem is considered solved when all passengers are delivered to their goal cells. [1]

Figure 2 visualizes a taxi problem. The grey circle corresponds to the agent. Squares of various colors indicate goal locations for theoretical passengers. The red circle corresponds to the passenger who wishes to be delivered to the red square. Walls are black lines and there are implicit lines around the edges of the map as before.

[1] More complex versions also include fuel and puddles.

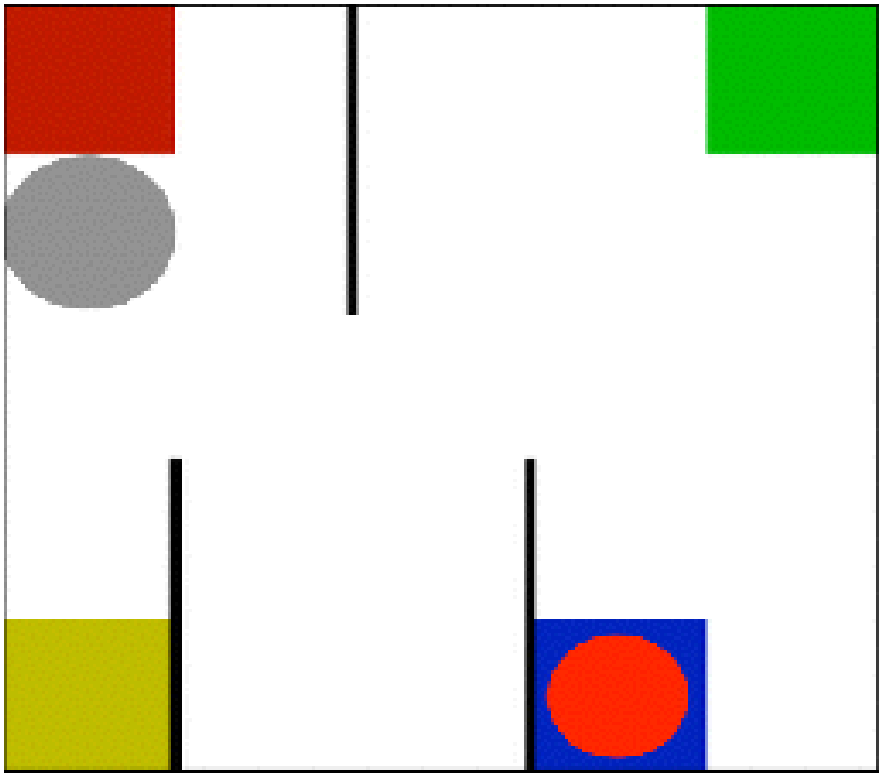
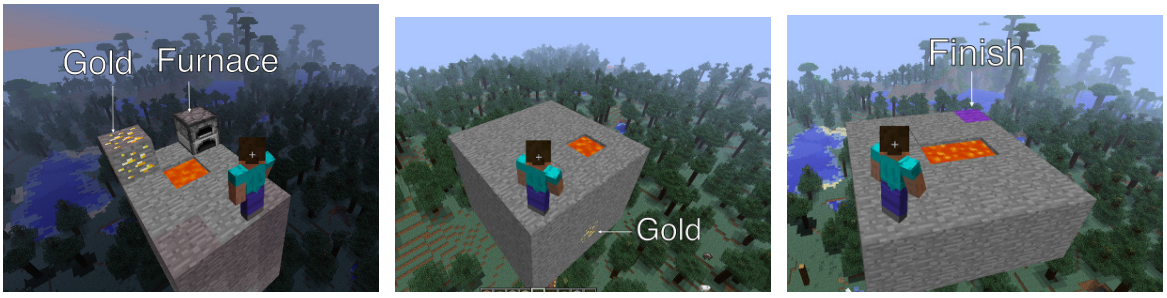


Figure 2: The classic taxi domain problem

### 2.1.3 Minecraft

Minecraft is an exceedingly complex sandbox style video game. An agent moves about in a three-dimensional voxelated world. Voxels may be occupied by blocks of various types or the agent itself. Additionally, the agent can both place and destroy blocks and is blocked by placed blocks. The game also supports an expansive crafting system for various items that alter how the agent interacts with the world – e.g. a shovel increases the rate at which the agent can dig dirt blocks – as well as a faux electricity simulation termed redstone.[24] Potential goals in the game are mining gold, smelting gold and navigating to a goal location all the while avoiding negative stimuli such as lava. Figure 3 visualizes minecraft problems with their goals.




(a) Mine the gold and smelt it in the furnace

(b) Dig down to the gold and mine it, avoiding lava.

(c) Navigate to the goal location, avoiding lava.

Figure 3: Three different problems from Minecraft.

Minecraft's complex mechanics render it not only capable of producing arbitrarily difficult problems with

burgeoning state spaces but its mechanics are fairly faithful to those which a robot must obey, making it an effective robotics-like problem simulator.

## 2.2 MDPs

If we are to solve the problems of Section 2.1 we can reasonably expect that we need a formal means of talking about the problems. Markov Decision Processes (MDPs) provide us with exactly such a formalism.

### 2.2.1 MDP Formalism

An MDP is a five-tuple, $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \gamma \rangle$:

$\mathcal{S}$ represents a state space where $s \in \mathcal{S}$ is a state representing a configuration of the world – this representation is robust enough that the planner need only ever consider its current state to plan optimally, hence the Markov in Markov Decision Process. Some subset of the states $\mathcal{S}_{term} \subseteq \mathcal{S}$ may also be terminal. When the agent arrives in a terminal state it ceases its executions of action.

$\mathcal{A}$ is the agent's set of actions. Each action represents how the agent can interact with its current state to potentially transition to a new state.

$\mathcal{T} : (\mathcal{S} \times A \times \mathcal{S}) \to [0, 1]$ denotes the transition dynamics as encoded by a real-valued function. $\mathcal{T}$ given the present state $s$, an action $a$, and a second state $s'$ returns the probability of $a$ transitioning from $s$ to $s'$. If the returned probability is always 1 or 0 for all triples in$(\mathcal{S} \times A \times \mathcal{S})$ the problem is deemed *deterministic*. Otherwise the problem is called *stochastic*.

$\mathcal{R} : \mathcal{S} \to \mathbb{R}$ denotes the reward function. Given some state, $s$, $\mathcal{R}$ returns the real-valued reward of arriving in $s$. An exceedingly common reward function is one which is uniformly negative; this uniform negative reward function encourages the agent to end its action taking as fast as possible by arriving in a terminal state and so is a convenient means of coaxing the agent towards a desired terminal state (or set of terminal states).

$\gamma \in [0, 1]$ is the discount factor. The discount factor defines how much the agent ought to prefer immediate reward over future reward; as $\gamma$ approaches 1 immediate reward becomes more important.

### 2.2.2 Grid World as an MDP

MDPs allow us to formalize those problems of Section 2.1. For example, a Grid World problem may now be understood as the following MDP:

$\mathcal{S}$: each state corresponds to the agent occupying a cell in the grid world. The state corresponding to the agent in the top right cell is terminal.

$\mathcal{A}$: north, east, south, west.

$\mathcal{T}$: actions are set to result in their desired behavior with probability 1. For example, the east action will transition to the state in which the agent occupies the cell one block to the right of its current cell (provided there are no walls).

$\mathcal{R}$: all states result in $-1$ reward.

$\gamma$: .95. Note that this (along with .99) is a fairly standard discount factor to use.

### 2.2.3 Solutions to MDPs

Solutions to an MDP are so called *policies*. A policy, $\pi : S \to A$, is a function which maps from states to actions. A policy has solved for behavior for all states and so may recover from unexpected outcomes and

therefore is well-suited to capturing intelligent behavior even when transition dynamics are stochastic. [2]

In solving for policies we are interested in maximizing the reward received from the states that result from our actions, $[s_0, s_1, s_2, \ldots]$. If we never arrive in a terminal state this will be an infinite number of states.

The most natural sort of reward to maximize is simply *additive reward*, the sum of the reward received for each state:

$$\mathcal{R}(s_0) + \mathcal{R}(s_1) + \mathcal{R}(s_2) + \ldots$$

However, consider the following two policies for a deterministic state space with no terminal states. $\pi_{bad}$ dictates cycling through a series of states, one cycle of which yields 1 reward. $\pi_{good}$ dictates cycling through a series of states, one cycle of which yields 1000 reward. $\pi_{good}$ seems to inform obviously more intelligent behavior given $\mathcal{R}$. However, over an infinite set of actions and both policies yield the same amount of additive reward, namely $\infty$. If we simply maximize additive reward, then, we cannot motivate choosing $\pi_{good}$ over $\pi_{bad}$. Discounted reward provides us with a means of avoiding this problem.

The *discounted reward* of a series of states is defined as the sum of the reward received over a series of states where each subsequent term is multiplied by the discount factor to one power higher than the last:

$$\mathcal{R}(s_0) + \gamma\mathcal{R}(s_1) + \gamma^2\mathcal{R}(s_2) + \ldots$$

Discounted reward is necessarily finite. This follows from the definition of discounted reward and the formula for the sum of an infinite geometric series:

$$\sum_{t=0}^{\infty} \gamma^t \mathcal{R}(s_t) \leq \sum_{t=0}^{\infty} \gamma^t \mathcal{R}_{max} = \frac{R_{max}}{1-\gamma}$$

where $R_{max}$ is the maximum reward returned by $R$ on a state. Given its finitude over an infinite series of states, discounted reward avoids the aforementioned problem of additive reward.

Thus, we define an **optimal policy** as one which maximizes discounted reward over an infinite series of actions. More formally an optimal policy, $\pi^*$, is:

$$\pi^* = \underset{\pi}{argmax} \left( E\left[ \sum_{t=0}^{\infty} \gamma^t \mathcal{R}(s_t) | \pi \right] \right)$$

[28]

Figure 4 shows an optimal policy for a grid world problem with a uniform negative reward function. States are labeled with arrows for their optimal action (where each grid cell is taken as identical to the state in which the agent occupies that cell). Note that multiple actions may be optimal as indicated by multiple arrows in a single state. Also note how the optimal policy coaxes the agent towards the terminal state so that it ceases to receive negative reward as soon as possible.

[2] A policy should not be confused with a plan. A plan is a finite series of actions. A plan is generally incapable of performing optimally in stochastic domains since when actions result in unexpected states, the plan is unable to respond in an intelligent, online fashion. Note that any algorithm which solves for a plan can trivially be turned into a policy by solving for the plan starting at every state and then mapping each state to the first action of said plan.

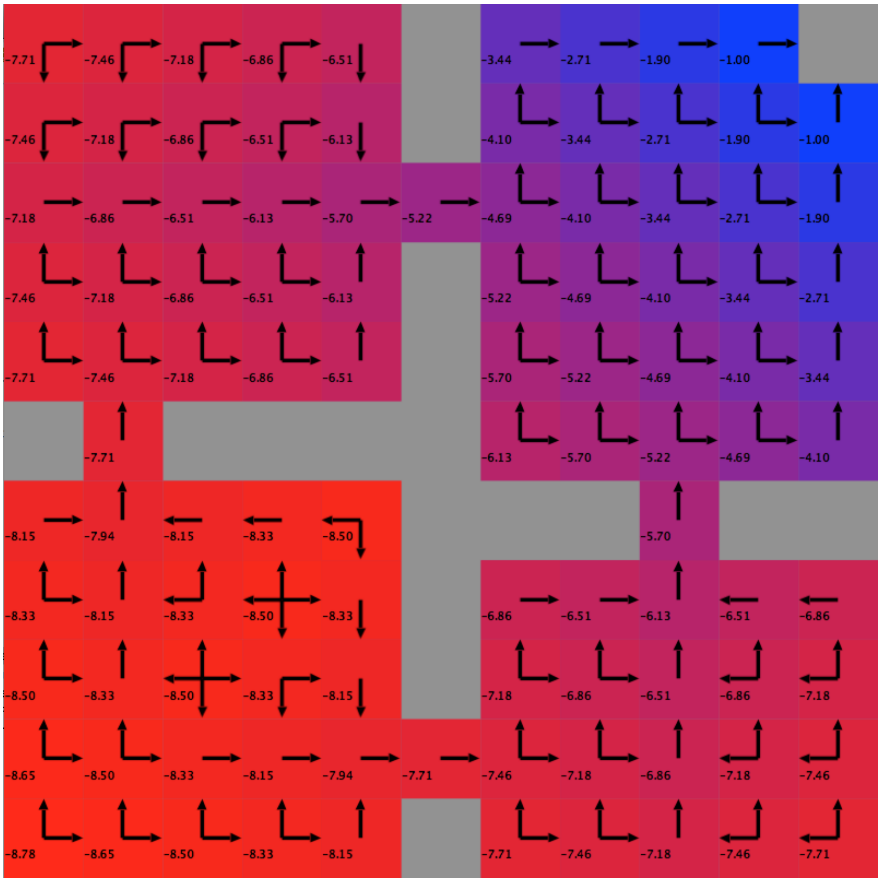



Figure 4: Sample Policy for Grid World Problem

What do the numbers and colors indicate? Read on to Section 2.3.1 and find out!

## 2.3 Important Notions in Planning and Reinforcement Learning

There remain a number of notions that crop up in much of planning and RL to cover before we can make sense of the core of planning and RL.

### 2.3.1 Value Functions

A value function, $V : \mathcal{S} \to \mathbb{R}$ is a critical notion in planning and reinforcement learning algorithms.

We can define the *value of a state* $s$ under a particular policy, $\pi$, as:

$$V^{\pi}(s) = E\left[\sum_{t=0}^{\infty} \gamma^t \mathcal{R}(s_t) | \pi, s_0 = s\right]$$

This can be understood as the (discounted) reward which we expect to receive starting in this state and then following $\pi$ ad infinitum.

The *optimal value of state* $s$ is defined as the value of that state under the optimal policy, $\pi^*$:

$$V(s) = V^{\pi^*(s)} = E\left[\sum_{t=0}^{\infty} \gamma^t \mathcal{R}(s_t) | \pi^*, s_0 = s\right]$$

Figure 5 provides an example grid world problem with a uniform $-1$ reward function where each state is annotated with its corresponding optimal value.

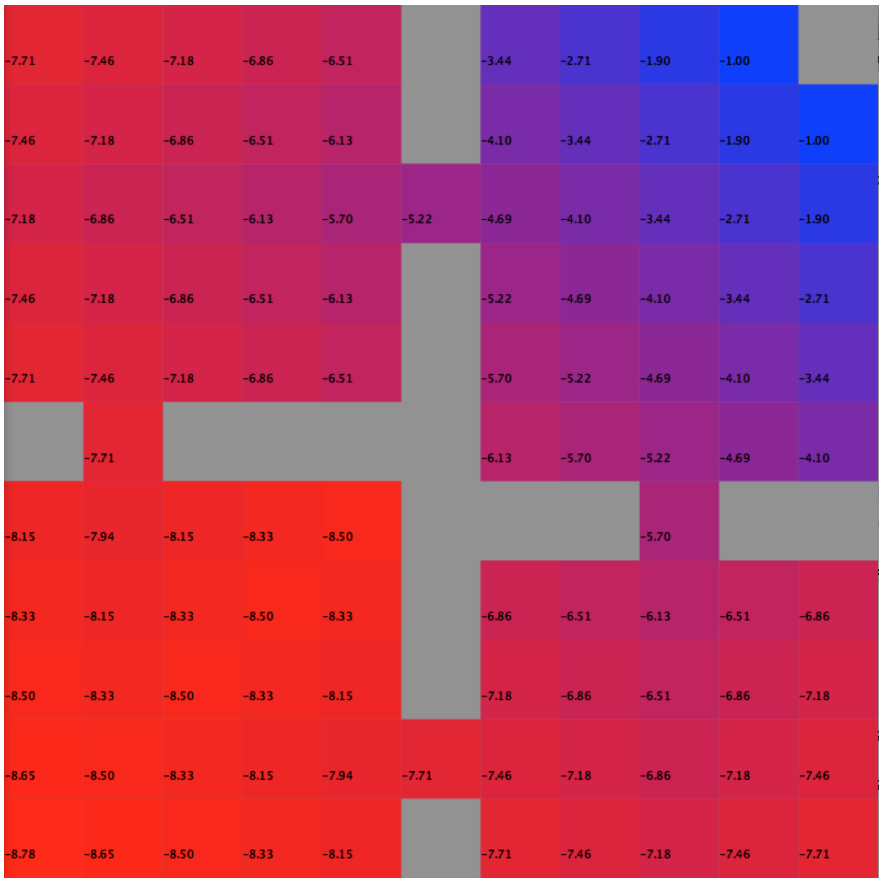



Figure 5: A grid world problem with states annotated with their optimal value given a uniform negative $\mathcal{R}$

The *optimal value function* is that which given a state returns the optimal value of that state – the term "value function" is generally used to refer to functions which estimate the optimal value function. Value functions, then, are estimations of the quality of states. Many planners, including value iteration (see Section 2.4.1) and RTDP (see Section 2.4.2) work by storing estimations of the optimal value function. The value function, however, should not be confused with the Q-function.

### 2.3.2 Q-Functions

A Q-Function, $Q : (\mathcal{S} \times \mathcal{A}) \rightarrow \mathbb{R}$ is also highly important to planning and reinforcement learning algorithms.

We can define the *Q-value* of an action $a$ in a state $s$ under a particular policy, $\pi$ as:

$$Q^{\pi}(s, a) = \mathcal{R}(s) + \sum_{s' \in S} T(s, a, s')R(s') + V^{\pi}(s')$$

This can be understood as the reward which we expect to receive after taking action $a$ in $s$ and then following $\pi$ ad infinitum. It can be thought of the value of $s$ under $\pi$ provided that we take $a$ from $s$.

The *optimal Q-value* of action $a$ in state $s$ g is defined as the Q-value of that action in that state under the

optimal policy, $\pi^*$:

$$Q(s,a) = Q^{\pi^*}(s,a) = \mathcal{R}(s) + \sum_{s' \in S} T(s,a,s')R(s') + V^{\pi^*}(s')$$

The *optimal Q-function* is the function which given a state $s$ and action $a$ returns the optimal Q-value of action $a$ in state $s$. A "Q-function" generally refers to a function which estimates the optimal Q-function.

Thus, while value functions estimate the quality of states, Q-functions estimate the quality of a state provided a particular action is taken in that state. RTDP detailed below, works by storing estimations of the true Q-function.

### 2.3.3 Exploration vs Exploitation

A dichotomic thread that runs through a great deal of planning and reinforcement learning is the trade-off between *exploration and exploitation*. On the one hand, we do not want to encourage dogmatic behavior wherein our agent does nothing but exploit those avenues of positive reward that it has so far found; there may well be other even more rewarding regions of the state space that our agent simply has not yet explored. On the other hand, we want our agent to maximize reward to the extent that what it's learned so far allows. Striking a balance between this maximization of known reward and discovery of new reward is dubbed the problem of exploration and exploitation and crops up throughout algorithms associated with MDPs.

## 2.4 Planning and Planning Algorithms

Problems in which the goal is to solve for an optimal policy and the algorithm has full access to elements of the MDP are dubbed *planning* problems. This is to be contrasted with reinforcement learning problems wherein the transition dynamics, $T$, are not in scope of the algorithm as discussed later.

### 2.4.1 Value Iteration

Value iteration (VI) is the canonical planning algorithm for solving for the optimal policy. Value iteration works by first solving for the true value of *every* state. It then sets the action of each state to that action which maximizes the expected value of the state the agent arrives in.

How do we solve for the true value of state $s$? We can first remove the first term from our sum in our definition of $V$ which is simply the reward of $s$.

$$V(s) = E\left[\mathcal{R}(s) + \sum_{t=1}^{\infty} \gamma^t \mathcal{R}(s_t) | \pi^*, s_0 = s\right] = \mathcal{R}(s) + E\left[\sum_{t=1}^{\infty} \gamma^t \mathcal{R}(s_t) | \pi^*, s_0 = s\right]$$

Since we assume we are using the optimal policy, $\pi^*$, the remaining expectation is simply the expected value that results from the optimal action – i.e. that action for which the expected value of the resulting state is maximal.

$$V(s) = \mathcal{R}(s) + \gamma \underset{a}{max} \sum_{\forall s' \in \mathcal{S}} T(s,a,s')V(s') \tag{1}$$

Equation 1 is the so called Bellman equation.

We can now formulate a system of $|\mathcal{S}|$ Bellman equations, one for the value of each state to solve for our $|\mathcal{S}|$ unknown state values. The equations are nonlinear so we estimate solutions by iteratively performing so called Bellman updates; the value of state $s$ at $i+1$ is updated as follows:

$$V_{i+1}(s) = \mathcal{R}(s) + \gamma \max_a \sum_{\forall s' \in \mathcal{S}} T(s, a, s') V_i(s')$$

A single iteration of value iteration consists of performing this update for every state in $\mathcal{S}$. Iterations are performed until the maximum change in the value of a state is below a pre-defined threshold, $\epsilon$.

Value iteration in full can be seen in algorithm 1.

**Algorithm 1** Value Iteration

INPUT: an MDP $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \gamma \rangle$, $\epsilon$ the convergence threshold for value change
OUTPUT: a policy, $\pi$
$\delta \leftarrow 0$

//Solve for value function, $V$
**repeat**
  **for all** $s \in S$ **do**
    $V'(s) \leftarrow \mathcal{R}(s) + \gamma \max_a \sum_{s'} T(s, a, s') V(s')$
    $\delta = max(\delta, |V(s) - V'(s)|)$
  **end for**
  $V \leftarrow V'$
**until** $\delta < \epsilon$
//Solve for optimal policy given $V$
**for all** $s \in S$ **do**
  $\pi(s) \leftarrow \underset{a \in A}{argmax} \sum_{s' \in \mathcal{S}} T(s, a, s') V(s')$
**end for**
**return** $\pi$

VI's advantages and disadvantages are both symptoms of its tabular nature. VI is referred to as a *tabular approach* since it requires examining every state in the state space to iteratively perform bellman updates. The tabular nature of VI pays off because the policy returned is guaranteed to be optimal (as $\epsilon$ approaches 0). However, examining every state is often cumbersome, particularly in very large state spaces so fully running VI is extremely computationally burdensome.

### 2.4.2 RTDP

Real Time Dynamic Programming (RTDP) [3] offers an online solution to planning problems where not all states need be examined. RTDP begins by pessimistically setting its estimation of the value of states to some input parameter $v$. It then learns in a series of rollouts (sometimes called trials): a single rollout consists of simulating the agent taking actions starting in the initial state until a terminal state is reached. Each action is greedily chosen given the agent's estimation of the value of states (ties are broken randomly). After each action the value of the state in which the action is taken is updated to be an estimation of the Q-value of the taken action. Rollouts continue until the value function converges for the initial state.[3] The returned policy is simply that by which actions are taken in rollouts: namely that which is greedy with respect to Q-value estimates[4] [5]. Full pseudo-code is detailed in figure 2.[5]

The great advantage of RTDP is that it need not examine every state and still produces optimal policies as the number of trials approaches infinity. Thus, RTDP achieves optimal policies in the limit with minimal

[3]Other termination conditions are certainly feasible though this is what is suggested by [5].

[4]Often time constraints necessitate placing a hard cap either on the number of actions in a single rollout or the number of rollouts. This was done in our work that made use of RTDP presented in Section 3.

[5]Note that the pseudo code's loops over $s \in \mathcal{S}$ are indicated for merely pedagogic and in actual implementation not every state actually need be examined. For instance, the initialization of $V(s) \leftarrow v$ is had by simply using $v$ when we need to estimate $V(s)$ for some $s$ which we haven't yet observed

**Algorithm 2** RTDP

INPUT: an MDP $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \gamma \rangle$, $\epsilon$ the convergence threshold for value change, $v$ initial pessimistic value of states, an initial state $s_0$
OUTPUT: a policy, $\pi$

//Initialize value function pessimistically
**for all** $s \in S$ **do**
  $V(s) \leftarrow v$
**end for**
//Run rollouts until value of initial state is below $\epsilon$
**while** $\delta > \epsilon$ **do**
  currentState $\leftarrow s_0$
  **while** currentState is not terminal **do**
    **for all** $a \in \mathcal{A}$ **do**
      $Q(\text{currentState}, a) \leftarrow \mathcal{R}(\text{currentState}) + \sum_{s' \in \mathcal{S}} T(\text{currentState}, a, s') * V(s')$
    **end for**
    $\text{nextAction} = \underset{a \in A}{\text{argmax}}\ Q(\text{currentState}, a)$
    $\delta \leftarrow |V(s) - Q(s, \text{nextAction})|$
    $V(s) \leftarrow Q(s, \text{nextAction})$
    $\text{currentState} \leftarrow T(s, \text{nextAction}, *)$
  **end while**
**end while**
//Solve for optimal policy given $V$
**for all** $s \in S$ **do**
  $\pi(s) \leftarrow \underset{a \in A}{\text{argmax}}\ Q(\text{currentState}, a)$
**end for**
**return** $\pi$

computation since it need only examine some states. The great difficulty of RTDP is setting the $\epsilon$ parameter: if it is much lower than the value of the worst state then we will nearly never explore and if it is much better than the value of the best state then we are doomed to explore ad nauseum.

## 2.5 Reinforcement Learning and Reinforcement Learning Algorithms

The problem of solving for an optimal policy without direct access to the transition dynamics, $\mathcal{T}$, is termed *reinforcement learning* (RL). In RL problems the agent has a current state and may take actions from its current state and observe the outcome but the agent may not simply query the value of $\mathcal{T}$ for various $(S \times A \times S)$ triples. The goal of the RL agent is the same as that of the planning agent: it wishes to maximize its discounted reward over an infinite action horizon. However, note that the reward received by those actions taken by the RL agent as it explores the state space count towards the discounted reward it receives in its infinite action horizon. Additionally, in RL problems if a terminal state is found, the agent returns to its initial state and continues taking actions.

### 2.5.1 Model-based Vs Model-Free Solutions

Solutions to reinforcement learning problems develop policies in one of two ways: either they do not estimate $\mathcal{T}$ (so called model-free approaches) or they do (so called model-based approaches). The canonical model-free approach is Q-Learning [28]: Q-Learning simply maintains estimates of true Q-values without any sort of model of $\mathcal{T}$. The canonical model-based approach is RMAX which is discussed below.

### 2.5.2 RMAX

RMAX[6] is a straightforward model based reinforcement learning algorithm: it observes the results of its actions and uses these results to estimate $\mathcal{T}$ and then solve for a policy. $\mathcal{T}$ is estimated by maintaining estimates for arriving in state $s'$ from state $s$ by taking action $a$ for all $(S \times A \times S)$ triples.[6] It estimates the transition probability as simply the number of times $a$ resulted in $s'$ from $s$ divided by the total number of times the agent took $a$ in $s$. This is conveniently the maximum-likelihood estimator for $T(s, a, s')$.

However, there is a caveat: if $a$ has not been taken at least some pre-specified number of times, $m$, then the agent hallucinates that $a$ will result in a transition to an illusory state with exceedingly positive reward, $RMAX$ (not to be confused with the name of the algorithm). Note that I also use $RMAX$ to refer to the state with $RMAX$ reward. This optimism about unobserved states is how RMAX encourages exploration.

Since RMAX provides us with a specification for $\mathcal{T}$, we now have a fully specified MDP so the problem becomes a planning problem. We may now run most any planning algorithm to yield a policy – VI is generally used.[7] RMAX need only run VI each time its estimate of $\mathcal{T}$ changes since this is the only time the policy as determined by VI changes. Additionally, note that the actions chosen as RMAX interacts with the environment are those dictated by the policy returned by VI given RMAX's estimation of $\mathcal{T}$. Lastly, RMAX does not have well defined termination conditions so these are left to the implementer. Full pseudocode is specified in figure algorithm 3.

[6] Non-observed triples are stored implicitly.

[7] There is a bit of ambiguity in what RMAX is meant to refer to: sometimes it is used just to refer to the process by which $\mathcal{T}$ is estimated and sometimes it is used to refer to the full process by which the policy is generated, including VI. I understand it in the latter sense.

**Algorithm 3** RMAX

INPUT: a partial MDP $\langle \mathcal{S}, \mathcal{A}, \mathcal{R}, \gamma \rangle$, the minimum number of observations to consider a transition dynamic known, $m$, an initial state $s_0$, $\epsilon$ for VI
OUTPUT: a policy, $\pi$

//Initialize data structures
**for all** $(s, a, s') \in (S \times A \times S)$ **do**
  $C(s, a, s') = 0$
  $\mathcal{T}(s, a, s') = RMAX$
**end for**
currentState $\leftarrow s_0$
$\pi \leftarrow$ ValueIteration$(\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \gamma \rangle, \epsilon)$

//Learn over actions
**while** terminating conditions not met **do**
  nextAction $\leftarrow \pi$(currentState)
  nextState $\leftarrow$ nextAction(currentState)
  $C$(currentState, nextAction, nextState)$+ = 1$
  **if** $C$(currentState, nextAction, nextState) $> m$ **then**
    **for all** $s' \in \mathcal{S}$ **do**
      **if** $C$(currentState, nextAction, $s'$) $> m$ **then**
        $\mathcal{T}$(currentState, nextAction, $s'$) $\leftarrow \frac{C(\text{currentState,nextAction},s')}{\sum_{s \in \mathcal{S}} C(\text{currentState,nextAction},s)}$
      **end if**
    **end for**
    $\pi \leftarrow$ ValueIteration$(\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \gamma \rangle, \epsilon)$
  **end if**
  currentState $\leftarrow$ nextState
**end while**
**return** $\pi$

RMAX's principle advantage is its model-based nature. Because it completes the five-tuple of the MDP, it allows for most any planning algorithm to be plugged into it. However, it tends to be exceedingly slow since in the worst case it must run VI over all observed states to convergence each time it takes an action.

## 2.6 OO-MDPs

One the notable shortcomings of the MDP is its highly tabular nature; every state is related to other states by the machinery of the MDP only in so far as one state might transition to the other given some action or series of actions. Consequently, the classic MDP is incapable of accommodating transfer of knowledge across similar states. Unlike the MDP, the OO-MDP, the object-oriented Markov Decision Process (OO-MDP) [11] provides an opportunity for transfer of knowledge across similar states – in particular it accommodates this knowledge transfer by exploiting the object-oriented structure of planning and reinforcement learning problems to create propositional functions.

### 2.6.1 OO-MDP Formalism

An OO-MDP is identical to an MDP except in the added mechanisms it provides for state representation and propositional functions.

An OO-MDP defines a set of $c$ *object classes*, $\mathcal{C} = \{C_1, \ldots, C_c\}$. Each $C_i \in C$ has a set of *attributes*, $Att(C_i) = \{C_i.a_1, \ldots, C_i.a_a\}$. Each attribute $a_j$ for each object class $C_i$ has some *domain*, $Dom(C_i.a_j)$, that

defines the values that $a_j$ can adopt. A single state $s \in \mathcal{S}$ state in an OO-MDP consists of $o$ instantiations of object classes, $\mathcal{O} = \{o_1 \ldots, o_o\}$, wherein each instantiation is an assignment to the attributes of the instantiated object class; that is, the OO-MDP state $s = \cup_{i=1}^{o} o_i$.

This state refactorization, in turn, allows one to define a space of highly generalized *propositional functions*, $\mathcal{P}$, that act on collections of objects and therefore OO-MDP states. That is $p \in \mathcal{P} : s \in \mathcal{S} \rightarrow \{\text{true}, \text{false}\}$.[8] These propositional functions are added to the OO-MDP and provide planners with a means of identifying similar states and transferring knowledge accordingly.

I also let $\mathcal{R}$ be a function that acts generically on collections of objects, i.e. it acts on $s \in \mathcal{S}$. This means that multiple OO-MDPs can share a single $\mathcal{R}$ provided that they share a set of object classes, $\mathcal{O}$.[9] This assumption will become important later when we start defining domains of related OO-MDPs.

Thus an OO-MDP is a seven-tuple: $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \gamma, \mathcal{O}, \mathcal{P} \rangle$ where $s \in \mathcal{S}$ is notable for its object-oriented representation using $\mathcal{O}$ and $\mathcal{P}$ defines propositional functions which provide planners with a means of relating similar states.

### 2.6.2 Grid World as an OO-MDP

The OO-MDP formalization of grid world is the same as the MDP formalization except we now define our object classes and have propositional functions:

$\mathcal{A}, \mathcal{T}, \mathcal{R}$ and $\gamma$ are identical to those provided in section 2.2.2

$\mathcal{O}$: { agent, wall }, $Att(agent) = Att(wall) = \{\text{xLocation}, \text{yLocation}\}$, $Dom(\text{xLocation}) = Dom(\text{yLocation}) = [0, d) \in \mathcal{Z}^+$ where d is the dimension of the width or height of the map, assuming a square map.

$\mathcal{P}$: wallToNorth, wallToSouth, wallToEast, wallToWest. Each propositional function is true iff there is a wall to the cell appropriate cell adjacent to the agent.

$\mathcal{S}$: $s \in \mathcal{S}$ is a set of collections of objects. The collection of objects consists of one instantiation of an agent and some number of walls where the xLocation and yLocation of the agent are free to vary over their domain for different states but the xLocation and yLocation of the walls stay fixed across states.

I now move to the original contributions sections of my paper.

# 3 Leveraging Propositional Functions: Goal-Based Action Priors

Many subspaces of large stochastic state spaces are highly irrelevant to optimal behavior. For instance, a robot trying to plan over actions in a cooking domain need not concern itself with all those configurations of its problem that follow from it boiling flour since no reasonable recipe will involve boiling flour. What we would like, then, is a means by which we can discard portions of the state space unlikely to be involved in an optimal plan.

I present such a pruning approach that leverages propositional functions in the following section to perform state space pruning in large stochastic state spaces by learning so called *goal-based action priors*. This work appears in full in [1]. Note that I in particular was involved in coding a simulation of Minecraft, generating Minecraft training worlds as well as the theory behind the methodology for action pruning.

**Section 3.1**: We first clarify our notions of *domain* and and *goals*.

**Section 3.2**: There are a number of existing approaches to intelligently reducing those states that a planner must examine though they fail to fill the niche that our approach does. I examine, temporally extended

[8] Really, it maps from $s \in$ any state space producible by instantiations of $\mathcal{O}$ rather than a particular state space $\mathcal{S}$ since these PFs act on collections of objects.

[9] It is unclear to me whether this is standard procedure in the OO-MDP literature. However given that $\mathcal{R}$ is meant to map from $\mathcal{S}$ to $\mathbb{R}$, and that OO-MDPs change $s \in \mathcal{S}$ to be collections of objects, it seems uncontentious to consider $\mathcal{R}$ as acting on a collection of objects.


Figure 6: Two problems from the same domain, where the agent's goal is to smelt the gold in the furnace.

actions [15, 29], existing action prior work for action pruning [27] as well as the deterministic approach hierarchical task networks (HTNs) [12]. However, these approaches fail to fill the planning niche which our methodology fills.

**Section 3.3:** Our approach begins by training on a series of small state spaces of problems representative of the larger state space problems we hope to solve. From these training sessions we learn a model parameter, $\theta$, which informs a distribution parameterized by propositional functions (supplied by the OO-MDP) on the optimality of actions.

**Section 3.4:** Lastly we carry this model parameter $\theta$ into novel large state spaces and prune away action on a state by state basis; pruned actions are those actions which are highly unlikely to be optimal according to the distribution dictated by $\theta$. By intelligently pruning actions, we limit those states examined by the planner and therefore intelligently reduce the number of the states that the planner must examine.

**Section 3.6** We conduct experiments on two OO-MDPs characterized by large state spaces: the game Minecraft and a robotic cooking assistant domain. We apply our method to RTDP – when $\theta$ is supplied by an expert we term our approach expert prior RTDP (EP-RTDP) and when it is learned on $W$ we term our approach learned prior RTDP (LP-RTDP). Our Minecraft experiments demonstrate significant planning time speedups, reduction in planning computation performed and improvement in the quality of inferred policies for EP-RTDP and LP-RTDP. Our cooking experiments demonstrate the same for EP-RTDP.

## 3.1 Terminology

Throughout this section we employ the following terminology.

A set of related OO-MDPs is termed a *domain*. In particular, OO-MDPs in a single domain share all aspects of the OO-MDP seven-tuple except for a state space $\mathcal{S}$. For instance, Minecraft can be thought of as a domain since it defines a series of similar OO-MDPs each of which have different state spaces.

A *goal* is a propositional function that is used to define a subset of states in $\mathcal{S}$ that are terminal. A goal specifies the sort of problem the agent is trying to solve, such as retrieving an object of a certain type from the environment, reaching a particular location, or creating a new structure. We let the space of all goals be denoted $\mathcal{G}$. These propositional functions are taken to not be included in $\mathcal{P}$. Figure 6 visualizes two OO-MDPs characterized by the same goal (smelt gold) in the same domain (Minecraft).

## 3.2 Related Work

There is a great deal of literature concerning accelerating planning. I highlight a couple of approaches that extend to stochastic state spaces like our presented approach as well as a deterministic approach which fails to generalize to stochastic state spaces.

**Stochastic Approaches:**
Temporally extended actions – *macro-actions* [15] and *options* [29] – are actions which are available to planners just as normal actions in $\mathcal{A}$ but which consist of the execution of multiple actions in $\mathcal{A}$.[10] Macro actions are simply a determinate combination of actions in $\mathcal{A}$. For example the moveEastALot macro action in grid world might consist of executing the moveEast action five times. Options are slightly more complex: they consist of an initiation condition under which an option policy $\pi_o$ is executed until the option's terminating condition is met. For example the moveOutOfRoom option for grid world might have an initiation condition that is always true which enters a policy that directs the agent out of the room until the terminating condition noLongerInInitialRoom is met.

However temporally extended actions are certainly not a panacea to planning in large stochastic state spaces. By providing planners with additional actions for every state that need to be considered temporally extended actions can paradoxically increase planning time [17]. Additionally, traditional options do not generalize across a single domain: that is, they are state space specific.[11] Our approach, on the other hand, is notable for reducing planning times and generalizing across domains.

[27] provide a method for learning action priors over a set of related tasks. They compute a Dirichelet distribution over actions by extracting the frequency that each action was optimal in each state. However, these action priors are only active some small $\epsilon$ amount of the time, limiting the degree to which they can improve planners. Moreover, these priors become more uniform as more diverse tasks are introduced, meaning they fail to efficiently handle a variety of tasks. Our methodology, on the other hand informs priors that are always active which are well-suited to learning over a variety of tasks as demonstrated in Table 1.

**Deterministic Approaches:**
The most notable deterministic approach to planning acceleration is the hierarchical task network (HTN)[12]. HTNs decompose the current task into smaller and smaller subgoals. This decomposition continues until immediately achievable primitive tasks are derived. The current task decomposition, in turn, informs constraints which reduce the space over which the planner searches. Nonetheless, HTNs as well as other deterministic approaches fail to generalize to stochastic domains as our approach does.

## 3.3 Inferring Optimal Action Distributions

Our approach begins by learning a distribution over the optimality of actions by exploiting the propositional functions in the OO-MDP.

### 3.3.1 Problem Setup

We define the optimal action set, $\mathcal{A}^*$, for a given state $s$ and goal $g$ as:

$$\mathcal{A}^*_{s,G} = \left\{ a \mid \underset{a \in \mathcal{A}}{\text{argmax}}\ Q^{\pi^*_G}(s,a) \right\}, \tag{2}$$

where $\pi^*_G$ is the optimal policy given goal $G$ and $Q^{\pi^*}(s,a)$ represent the optimal Q function.

What we would like to do is learn a probability distribution over the optimality of each action for a given state $s$ and goal $G$. Thus, we want to infer a Bernoulli distribution for each action where a "success" corresponds

[10] Also note that we actually apply our framework to temporally extended actions in [1] to demonstrate the complementarity of both methods but I do not cover that here since it does not fit our present narrative.

[11] However, note that creating so called "portable" options is a topic of active research [21, 9, 19, 26, 8, 2, 20].

to that action being optimal:

$$\Pr(a_i \in \mathcal{A}^*_{s,G} \mid s, G) \tag{3}$$

for $i \in \{1, \ldots, |\mathcal{A}|\}$.

Note that we are not learning a single distribution over all actions for a single $s$ and $G$ because multiple actions may well be optimal which would not be reflected by the use of a single distribution.

In order to learn this distribution we supply the agent with a set of training worlds from the domain ($W$) wherein each training world is small enough that the optimal policy $\pi_i$ has been solved for using value iteration.

### 3.3.2 Solving the Problem

Let $\delta_j \in (\mathcal{P} \times \mathcal{G})$ denote a propositional function paired with a goal.

We introduce the indicator function $f$, which returns 1 if and only if the given $\delta$'s predicate is true in the provided state $s$, and $\delta$'s goal is entailed by the agent's current goal, $G$:

$$f(\delta, s, G) = \begin{cases} 1 & \delta.p(s) \wedge \delta.G \models G \\ 0 & \text{otherwise.} \end{cases} \tag{4}$$

Evaluating $f$ for a single $\delta_j$ given the current state $s$ and goal $G$ gives rise to a binary feature, $\phi_j = f(\delta_j, s, G)$. Evaluating it for every $\delta_j \in (\mathcal{P} \times \mathcal{G})$ gives rise to a set of binary features which we use to approximate our probability distribution:

$$\Pr(a_i \in \mathcal{A}^*_{s,G} \mid s, G) \approx \Pr(a_i \in \mathcal{A}^*_{s,G} \mid \phi_1, \ldots, \phi_n) \tag{5}$$

This approximation may be modeled in a number of ways, making this approach quite flexible. We employ a Naive Bayes model from this point on.

We first factor using Bayes' rule, introducing a parameter vector $\theta_i$ of feature weights which is unique to each action $a_i \in \mathcal{A}$:

$$= \frac{\Pr(\phi_1, \ldots, \phi_n, \mid a_i \in \mathcal{A}^*_{s,G}, \theta_i) \Pr(a_i \in \mathcal{A}^*_{s,G} \mid \theta_i)}{\Pr(\phi_1, \ldots, \phi_n | \theta_i)} \tag{6}$$

Also note that all $\theta_i$s are stored in $\theta$.

Next we assume that each feature is conditionally independent of the others, given whether the action is optimal:

$$= \frac{\prod_{j=1}^{n} \Pr(\phi_j \mid a_i \in \mathcal{A}^*_{s,G}, \theta_i) \Pr(a_i \in \mathcal{A}^*_{s,G} \mid \theta_i)}{\Pr(\phi_1, \ldots, \phi_n | \theta_i)} \tag{7}$$

We can drop our denominator for proportionality since it is the same whether or not the action is, in fact, optimal to yield our final estimate:

$$\alpha \prod_{j=1}^{n} \Pr(\phi_j \mid a_i \in \mathcal{A}^*, \theta_i) \Pr(a_i \in \mathcal{A}^* \mid \theta_i) \tag{8}$$

Additionally, we store the following counts in $\theta_i$ for each action:

$$C(a_i) = \sum_{w \in W} \sum_{s \in w} (a_i \in \pi_w(s)) \tag{9}$$

$$C(\bar{a_i}) = \sum_{w \in W} \sum_{s \in w} (a_i \notin \pi_w(s)) \tag{10}$$

$$C(\phi_j, a_i) = \sum_{w \in W} \sum_{s \in w} (a_i \in \pi_w(s) \wedge \phi_j == 1) \tag{11}$$

$C(a_i)$ is the number of observed occurrences where $a_i$ was optimal across all worlds $W$, $C(\bar{a_i})$ is the number of observed occurrences where $a_i$ was not optimal, and $C(\phi_j, a_i)$ is the number of occurrences where $\phi_j = 1$ and $a_i$ was optimal. We determined optimality using the synthesized policy using value iteration for each training world.

These counts conveniently can be used to recreate maximum likelihood estimators for our values of interest in Equation 8:

$$\Pr(\phi_j \mid a_i \in \mathcal{A}^*, \theta_i) = C(\phi_j, a_i) \tag{12}$$

$$\Pr(a_i \in \mathcal{A}^* \mid \theta_i) = \frac{C(a_i)}{C(a_i) + C(\bar{a_i})} \tag{13}$$

$$\tag{14}$$

Thus our final formula for the optimality of action $a_i$ for a particular goal $G \in \mathcal{G}$ in state $s \in \mathcal{S}$ is

$$\Pr(a_i \in \mathcal{A}^* \mid s, G) \approx \alpha C(\phi_j, a_i) \frac{C(a_i)}{C(a_i) + C(\bar{a_i})} \tag{15}$$

where these counts for each action are stored in $\theta$. Equation 15 is the final equation used for our goal-based action priors.

## 3.4 Pruning with Optimal Action Distributions

### 3.4.1 Problem Setup

We would now like to leverage our goal-based action priors to expedite planning. In particular, given a state, $s$, goal $G$ and action set $\mathcal{A}$, we would like to pick some subset of actions for $s$ given $G$, $A_{s,G} \subseteq \mathcal{A}$, according to our goal-based action prior, namely a Bernoulli distribution on $\Pr(a_i \in \mathcal{A}^*_{s,G} \mid s, G)$.

### 3.4.2 Solving the Problem

Our solution is fairly straightforward; we simply choose a probability threshold and then prune those actions which do not meet our threshold. That is, given some threshold $\eta \in (0, 1)$:

$$A_{s,G} = \{a \in \mathcal{A} | Pr(a \in \mathcal{A}^*_{s,G} \mid s, G) > \eta\} \tag{16}$$

Empirically we found the heuristic of setting $\eta$ to $\frac{0.2}{|\mathcal{A}|}$ to be effective and so this is the $\eta$ we use.

## 3.5 Applying Action Pruning to Planners

Any planners can straightforwardly employ our approach: when examining state $s$ given goal $G$, we only allow the planner to access $A_{s,G}$ rather than the full OO-MDP action space $\mathcal{A}$. As a result of this state by state limited action set the planner is cut off from large swaths of the state space which are unlikely to inform the most relevant states in the optimal policy.

Our approach is particularly notable for its generalizability: not only are we able to limit states examined in large state spaces but we are able to do so across a variety of large state spaces defined by a single domain. Moreover, since our approach is goal specific our priors do not collapse into uniform distributions if the agent is trained on a variety of test worlds with very different goals.

## 3.6 Experimental Results

### 3.6.1 Planners Used

We empirically demonstrate the virtues of our approach by applying it to RTDP (see Section 2.4.2 for more on RTDP). In particular, when RTDP executes, we only allow it to consider $A_{s,G}$ rather than the full action set $\mathcal{A}$ when examining $s$ and trying to complete goal $G$ as informed by thresholding the distribution dictated by $\theta$. When $\theta$ is provided by an expert we term the resulting planner expert-prior RTPD (EP-RTDP) and when $\theta$ is learned on test worlds we term the resulting planner learned-prior RTDP (LP-RTDP). We terminate each planner when the maximum change in the value function is less than 0.01 for 100 consecutive policy rollouts, or the planner fails to converge after 1000 rollouts.

### 3.6.2 Minecraft Results

**Experimental Setup:**
OO-MDPs in our Minecraft domain are characterized by the following. We had five goals: bridge construction, gold smelting, tunneling through walls, digging to find an object, and path planning. The terminal states for a particular problem instance are specified by the goal. Our set of propositional functions in our OO-MDP, $\mathcal{P}$, consists of of 51 propositional functions we thought likely to aid in prediction of action optimality. The reward function is $-1$ for all transitions, except transitions to states in which the agent is in lava, where we set the reward to $-10$. To introduce non-determinism into our problem, movement actions (move, rotate, jump) in all experiments have a small probability (0.05) of incorrectly applying a different movement action. This noise factor approximates noise faced by a physical robot that attempts to execute actions in a real-world domain and can affect the optimal policy due to the existence of lava pits that the agent can fall into. The discount factor is $\gamma = 0.99$.

The training set (i.e. $W$) for LP-RTDP consists of 20 distinct problem instances of each goal, for a total of 100 instances. Each instance is extremely simple: 1,000-10,000 states (small enough to solve with tabular approaches). The output of our training process is the model parameter $\theta$, which informs our goal-based action prior. The full training process takes approximately one hour run in parallel on a computing grid, with the majority of time devoted to computing the optimal value function for each training instance.

The test set for both RTDP, LP-RTDP and EP-RTDP consists of 20 randomly generated instances of the same goal, for a total of 100 instances. Each instance is extremely complex: 50,000-1,000,000 states (which is far too large to solve with tabular approaches).

For RTDP, EP-RTDP and LP-RTDP, we report the number of Bellman updates executed, the accumulated reward of the average plan, and the CPU time taken to find a plan. Table 1 shows the average Bellman updates, accumulated reward, and CPU time for RTDP, LP-RTDP and EP-RTDP after planning in the test set. Figure 7 shows the results averaged across all 100 problem instances. We report CPU time for completeness, but our results were run on a networked cluster where each node had differing computer and

| Planner | Bellman | Reward | CPU |
|---|---|---|---|
| *Mining Task* | | | |
| RTDP | 17142.1 (±3843) | **-6.5** (±1) | **17.6s** (±4) |
| EP-RTDP | 14357.4 (±3275) | **-6.5** (±1) | 31.9s (±8) |
| LP-RTDP | **12664.0** (±9340) | -12.7 (±5) | 33.1s (±23) |
| *Smelting Task* | | | |
| RTDP | 30995.0 (±6730) | **-8.6** (±1) | 45.1s (±14) |
| EP-RTDP | 28544.0 (±5909) | **-8.6** (±1) | 72.6s (±19) |
| LP-RTDP | **2821.9** (±662) | -9.8 (±2) | **7.5s** (±2) |
| *Wall Traversal Task* | | | |
| RTDP | 45041.7 (±11816) | -56.0 (±51) | **68.7s** (±22) |
| EP-RTDP | 32552.0 (±10794) | -34.5 (±25) | 96.5s (±39) |
| LP-RTDP | **24020.8** (±9239) | **-15.8** (±5) | 80.5s (±34) |
| *Trench Traversal Task* | | | |
| RTDP | 16183.5 (±4509) | **-8.1** (±2) | 53.1s (±22) |
| EP-RTDP | **8674.8** (±2700) | -8.2 (±2) | **35.9s** (±15) |
| LP-RTDP | 11758.4 (±2815) | -8.7 (±1) | 57.9s (±20) |
| *Plane Traversal Task* | | | |
| RTDP | 52407 (±18432) | -82.6 (±42) | 877.0s (±381) |
| EP-RTDP | 32928 (±14997) | -44.9 (±34) | 505.3s (±304) |
| LP-RTDP | **19090** (±9158) | **-7.8** (±1) | **246s** (±159) |

Table 1: RTDP vs. EP-RTDP vs. LP-RTDP

memory resources. As a result, the CPU results have some variance not consistent with the number of Bellman updates in Table 1.

**Observed Results:**

Figure 7 shows that LP-RTDP on average finds a comparably better plan (10.6 cost) than EP-RTDP (22.7 cost) and RTDP (36.4 cost), in significantly fewer Bellman updates (14287.5 to EP-RTDP's 24804.1 and RTDP's 34694.3), and in less CPU time (93.1s to EP-RTDP's 166.4s and RTDP's 242.0s). These results indicate that while learned priors provide the largest improvements, expert-provided priors can also significantly enhance performance. Thus, depending on the domain, expert-provided priors can add significant value in making large state spaces tractable without the overhead of supplying training worlds.

Table 1 shows similar improvements across all goal types. For some task types, LP-RTDP finds a slightly worse plan on average than RTDP (*e.g.* the mining task). This worse convergence is due to the fact that LP-RTDP occasionally prunes actions that are in fact optimal (such as pruning the `destroy` action in certain states of the mining task). Additionally, RTDP occasionally achieved a faster clock time because EP-RTDP and LP-RTDP also evaluate several OO-MDP predicates in every state, adding a small amount of time to planning – note that we reevaluate each predicate every time the agent visits a state, which could be optimized by caching predicate evaluations, likely bridging this CPU gap.

#### 3.6.3 Robotic Cooking Assistant Results

The practical robotics applications of our goal-based action priors were also demonstrated on a robotic assistant cooking task. I was not directly involved with this work so I defer to 3rd person citations of [1] for this section when talking about this work.

**Experimental Setup:**
[1]'s robotic cooking assistant's domain is characterized by the following. A brownie recipe is divided into

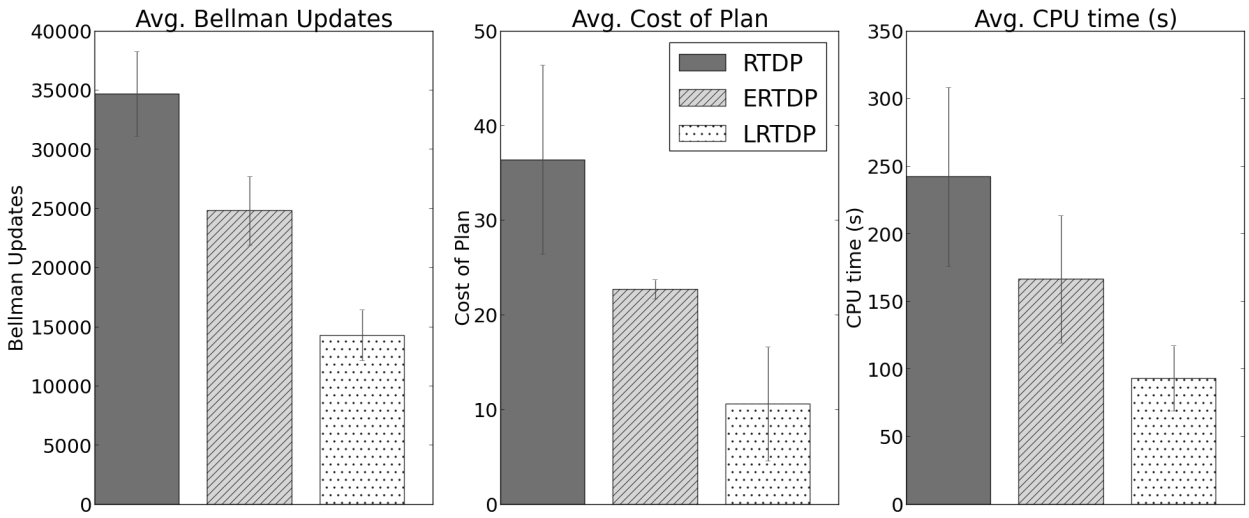



Figure 7: Average results from all maps.

Figure 8: Goal-based action priors enable a robot to efficiently infer helpful actions in very large state spaces, such as a kitchen.

three subgoals: combining and mixing the dry ingredients, combining and mixing the wet ingredients, and combining these two mixtures into a batter. Each subgoal is treated as a goal in the earlier sense – that is, [1] provide action priors specific for each subgoal. The human participant and the robotic companion are each modeled as separate OO-MDPs. From the robot's perspective, the human is just a stochastic element of its OO-MDP's transition dynamics. The robot's OO-MDP contained three spaces: human counter, robot counter, sink; four ingredient bowls, two mixing bowls, and two tools that could be in any of the three spaces, in any configuration. Additionally, the robot's OO-MDP contains the following ingredients: cocoa powder, sugar, eggs, and flour. Each container/tool can occupy one of three spaces, and each ingredient in one of the containers is either mixed or unmixed. Because of its fine-grained nature, our cooking state space has whopping $47,258,883$ states when configured with the ingredients and tools necessary to make brownies. The robot's action consist of things like handing off the whisk to the person to mix the wet ingredients. EP-RTDP is used to search for the least-cost plan to complete the recipe.

**Observed Results:**

[1] report the same quantities as in Section 3.6.2 – Bellman updates executed, the accumulated reward of the average plan, and the CPU time taken to find a plan. Unlike previous CPU time results, these experiments

| Planner | Bellman | Reward | CPU |
|---|---|---|---|
| *Dry Ingredients* | | | |
| RTDP | 20000 (±0) | -123.1 (±0) | 56.0s (±2.9) |
| EP-RTDP | **2457.2** (±53.2) | **-6.5** (±0) | **10.1s** (±0.3) |
| *Wet Ingredients* | | | |
| RTDP | 19964 (±14.1) | -123.0 (±0) | 66.6s (±9.9) |
| EP-RTDP | **5873.5** (±53.7) | **-6.5** (±0) | **15.6s** (±1.2) |
| *Brownie Batter* | | | |
| RTDP | 20000 (±0) | -123.4 (±0.7) | 53.3s (±2.4) |
| EP-RTDP | **6642.4** (±36.4) | **-7.0** (±0) | **31.9s** (±0.4) |

Table 2: RTDP vs. EP-RTDP for robotic kitchen tasks

were conducted on the same multi-core computer.

In Table 2 [1] compare between standard RTDP and EP-RTDP planning for each of the three subgoals. Because the state-action space is reduced significantly, EP-RTDP can plan successfully in a short amount of time. Standard RTDP always encounters the maximum number of rollouts specified at the maximum depth each time, even with a relatively small number of objects.

[1] provide a video[12] showing how EP-RTDP running on a robot can help a person cook by dynamically replanning through constant observations. After observing the placement of a cocoa container in the robot's workspace, the robot fetches a wooden spoon to allow the person to mix. After observing an egg container, the robot fetches a whisk to help beat the eggs. The robot dynamically resolves failures and accounts for unpredictable user actions; in the video, the robot fails to grasp the wooden spoon on the first attempt and must retry the grasp after it observed no state change.

# 4 Learning Propositional Functions: DOORMAX Style Inference

Propositional functions in OO-MDPs are of crucial importance in planning and reinforcement learning algorithms: as demonstrated in Section 3 they can be used to greatly reduce the computational complexity of planning problems and as demonstrated in Section 4.2.3 they can be incorporated into the powerful reinforcement learning technique deterministic object-oriented RMAX (DOORMAX). They also are of crucial importance in a number of other techniques such as linear function approximations for value functions [14]. However, propositional functions are often difficult to supply: they might be strictly obfuscated to the algorithm developer or else equivalently simply onerous to specify. Developing a means by which propositional functions can be learned then is of immediate importance to planning and reinforcement learning.

**Section 4.1:** Existing techniques fail to provide a means by which propositional functions may be learned for OO-MDPs. I examine traditional machine learning techniques for supervised learning, auto-encoders [16] and Batch incremental feature dependency discovery (Batch-iFDD) [13].

**Section 4.2:** My technique works by building on the algorithmic framework provided by DOORMAX and so I begin with an exposition of it.

**Section 4.3:** I begin my presentation of my own technique by presenting a means of relationally featurizing states in an OO-MDP in a manner that is likely to inform transition dynamics.

**Section 4.4:** I next discuss how use this state featurization and a novel DOORMAX-like algorithm can be combined to produce labeled datasets of featurized states given a series of (initialState, action, resultingState) observations of an RL agent.

**Section 4.5:** Lastly, I discuss how each generated dataset is turned into a propositional function.

[12]Watch at `https://vimeo.com/106226282`

**Section 4.6:** I also derive a novel DOORMAX algorithm which requires no expert-provided propositional functions and which uses propositional functions learned in my manner instead.

**Section 4.7:** Empirical results consist of applying my methodology to learn propositional functions in the taxi domain. The used taxi domain is detailed in Section 4.7.1. Learned propositional functions are demonstrated to correspond well with the standard taxi domain propositional functions in Section 4.7.2. Moreover, my novel DOORMAX algorithm is shown to perform essentially as well as standard DOORMAX in Section 4.7.3.

## 4.1 Related Work

There are a number of existing methods in machine learning, planning and reinforcement learning related to my methodology but which fail to accomplish what my methodology does.

*Machine learning methods for binary classification* in supervised learning settings such as support vector machines (SVMs) [7] can be thought of as learning a propositional function given some labeled dataset, namely the propositional function which is true when an input is of some class. Consequently, the learnable propositional functions by this methodology is highly robust: if a dataset can be mustered and meaningfully annotated, these methods can produce meaningful propositional functions. However, these methods alone cannot produce propositional functions for states in an OO-MDP: in particular they require both a means of deriving a dataset and meaningfully annotating said dataset. My work can be understood as providing both datasets and annotations for these machine learning methods

*Autoencoders* [16] are closer to accomplishing what my methodology does. An autoencoder is simply an artificial neural network like any other. Given sample inputs, autoencoders are trained to learn the identity function – that is, they update weights between nodes in such a way that their output is identical to their input. The hidden layers of the autoencoder have fewer nodes than the input layer, thereby imposing a compression bottleneck which forces the autoencoder to learn "how to think" about its input. In this manner internal neurons of the autoencoder can be thought of as learning useful, high-level abstractions about their input much like the propositional functions that I learn. However, it is, at the very least, nontrivial how one would go about directly applying autoencoders to learning propositional functions for OO-MDPs.

*Batch incremental feature dependency discovery (Batch-iFDD)* [13] also offers a means of deriving novel propositional functions in planning domains. In particular, it learns useful conjunctions of more atomic propositional functions in batch. However, this methodology requires an initial pool of atomic propositional functions and so does not fill the niche of my method – namely generating novel propositional functions for planning and reinforcement learning without use of provided propositional functions. I am, nonetheless, interested in using my learned propositional functions as those supplied to Batch-iFDD.

## 4.2 DOORMAX

My methodology uses a DOORMAX-like framework and so I now cover the details of the algorithm. DOORMAX is the OO-MDP reinterpretation of RMAX for non-stochastic state spaces. That is, it is a model-based reinforcement learning algorithm for OO-MDPs in which $\mathcal{T}$ only ever outputs one state given any $(\mathcal{S} \times \mathcal{A})$ tuple. DOORMAX's critical contribution is its capitalization on the propositional functions that OO-MDPs offer.

However, given its exceeding complexity we are well served by first defining some DOORMAX lingo.

### 4.2.1 Lingo

**Conditions:**
A condition is a bit string where each bit represents the truth value of one of the propositional functions in the OO-MDP. The propositional functions are always evaluated in the same order to produce the condition

– that is, indices of the bit string uniquely correspond to propositional functions of the OO-MDP. Figure 9 provides an example of a taxi domain problem where only one propositional function is true, namely wallToWestOfAgent, yielding a condition of (000100) since the wallToWestOfAgent propositional function is indexed by 3.

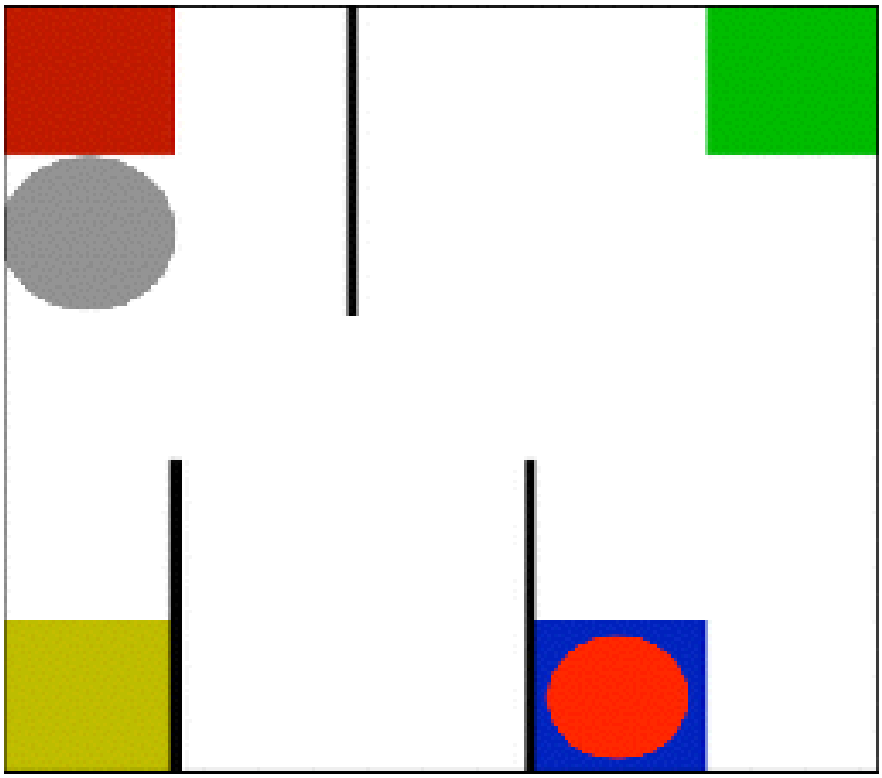


Figure 9: An example of a taxi problem with condition (000100)

Bits may also be * in a condition to indicate that a propositional function "doesn't matter" for that condition.

Conditions are a compact means of summarizing when actions cause changes in the OO-MDP. For instance, I might (correctly) hypothesize that the *west* action has an effect only when there is no wall to the west which I could summarize with the condition (***0**). Consequently, we will use conditions to summarize when we think particular effects will take place in the OO-MDP given some action.

I let my full space of conditions be denoted $CON$.

**$\oplus$ of Conditions:**
$\oplus$ is a binary operator that acts on two conditions, returning a condition of the same length. Each index of the returned condition is 1 if both input conditions were 1 at that index, 0 if both input conditions were 0 at that index and * otherwise. For instance:

$$(000) \oplus (010) = (0 * 0)$$

$$(111) \oplus (111) = (111)$$

$$(111) \oplus (000) = (* * *)$$

More formally

$$cond_1 \oplus cond_2 = \bigparallel_{i=0}^{|cond_1|} cond_1[i] \text{ if } cond_1[i] = cond_2[i] \text{ else } *$$

where $||$ is a concatenation.

We will use the $\oplus$ operator to update the conditions under which we think particular effects will take place for particular actions.

**$\models$ (matching conditions):**
One condition matches another condition, denoted $cond_1 \models cond_2$, if for every index either both conditions have the same bit or the first condition has a * at that index. Note that this means that matching is not a symmetric operator. For instance:

$$000 \models 000$$
$$000 \not\models 100$$
$$*** \models 100$$
$$100 \not\models ***$$
$$*0* \models 100$$
$$*0* \not\models 110$$

More formally

$$cond_1 \models cond_2 \text{ iff } \overset{|cond_1|}{\underset{i=0}{AND}} [cond_1[i] = cond_2[i] \text{ or } cond_1[i] = *]$$

We determine which effects we think will take place by looking at whether those conditions we think effects take place under match the current state.

**Overlapping Conditions:**
We say that two conditions overlap if one condition matches the other or vice versa. That is,

$$cond_1 \text{ overlaps } cond_2 \text{ iff } cond_1 \models cond_2 \text{ or } cond_2 \models cond_1$$

Overlapping conditions will be used to rule out contradictory predictions (more on predictions soon).

**Effects:**
An effect is defined by a type, an attribute, an object class and a real number. The type of the effect dictates how the effect changes the attribute of the object class. The two types examined here are assignment and arithmetic effects: *assignment effects* set attributes of the effect's object class to a fixed value and *arithmetic effects* add a value to the attribute of the object class of the efFfect. I let $\mathcal{Y} = \{arithmetic, assignment\}$ denote the set of effects used. Lastly, the real number indicates what value is added (arithmetic effects) or assigned (assignment effects). Thus, effects are 4-tuples $\langle type \in \mathcal{Y}, att \in \cup_{O \in \mathcal{O}} Att(O), O \in \mathcal{O}, r \in \mathbb{R} \rangle$.

I let my space of effects be noted $\mathcal{E} = (\mathcal{Y} \times \cup_{O \in \mathcal{O}} Att(O) \times \mathcal{O} \times \mathbb{R})$

Effects are a means of hypothesizing how actions affect attributes of one object class and so we will use effects for exactly this. For instance, I might (correctly) hypothesize that the effect of the west action is an arithmetic effect of adding -1 to the xLocation attribute of the agent class. In more formal notation my effect would be: $\langle \text{arithmetic}, xLocation, agentClass, -1 \rangle$. For this reason effects are treated as functions that input states in $\mathcal{S}$ and output states in $\mathcal{S}$. That is, effect $e \in \mathcal{E} : \mathcal{S} \rightarrow \mathcal{S}$.

Given a particular object class $oClass$, attribute $att$ and states $s$ and $s'$ I let $\text{eff}_{oClass,att}(s, s') : (\mathcal{S} \times \mathcal{S}) \rightarrow (\text{effects} \subseteq \mathcal{E})$ return all effects that capture how $oClass$'s $att$ changes from $s$ to $s'$ for all effect types in $\mathcal{Y}$.

**Incompatible effects:**
Two effects are said to be *incompatible* for a particular state $s \in \mathcal{S}$ when they both act on the same attribute and object class but they would cause different values to be assigned to the attribute of the object class if applied in state $s$. That is, two effects, $e_1$ and $e_2$, are incompatible for state $s$ iff $e_1(s) \neq e_2(s)$.

**Predictions:**
Predictions are uniquely defined by an action, an effect and a condition. They are the unit of DOORMAX

which summarizes the condition under which an effect is thought to take place when a particular action is taken.

For example, a (correct) prediction for west's effect on the x attribute of the taxi is the arithmetic effect of adding -1 to the x attribute of the taxi provided the state meets the condition (***0**), i.e. whenever there is no wall to the west.

More formally predictions are a tuple of an action, a condition and an effect: $pred \in (\mathcal{A} \times CON \times \mathcal{E})$.

It is worth noting that the condition of a prediction will be updated during learning but the effect is always static for a given prediction.

**RelatedPredictions:**
I define the set of predictions related to $pred \in (\mathcal{A} \times CON \times \mathcal{E})$ to be those predictions $\in (\mathcal{A} \times CON \times \mathcal{E})$ which have the same action as *pred* and have effects of the same type that act on the same object class and attribute.

### 4.2.2 Algorithm

DOORMAX's input consists of an OO-MDP without $\mathcal{T}$, a parameter $k$ which dictates the maximum number of related predictions with non-overlapping conditions allowed, an initial state $s_0$ and any parameters for planners that it uses – here $\epsilon$ for VI. It outputs a policy $\pi$.

The algorithm by which DOORMAX works is as follows. DOORMAX begins by initializing its data structures. For each triple of action, object class and attribute, DOORMAX initializes a set, $F_{a,att,oClass}$, to empty for the conditions under which that action has been observed to not effect that attribute of that object instance. Each of these sets is essentially a tabular set of failure conditions for when $a$ does not change $att$ of any instances of $oClass$. DOORMAX also initializes an empty set, $\alpha$, for our body of predictions and a set, $\omega$ for the contradictory (effect types, object class, attribute, action) tuples. DOORMAX's primary loop is divided into a prediction phase and a learning phase. In this loop DOORMAX predicts a model of the transition dynamics $\mathcal{T}$ and feeds this to VI[13] which in turn produces a policy, $\pi$. DOORMAX takes an action from its current state according to $\pi$. Based on the state that results from taking the action dictated by $\pi$ in the current state, DOORMAX learns by updating its model. This prediction and learning cycle repeats until some specified termination conditions are met. The top-level pseudo-code for DOORMAX can be seen in algorithm 4.

[13]Though as with RMAX any planning algorithm could be used.

**Algorithm 4** DOORMAX

INPUT: a partial OO-MDP $\langle \mathcal{S}, \mathcal{A}, \mathcal{R}, \gamma, \mathcal{O}, \mathcal{P} \rangle$, the maximum number of effects $k$, an initial state $s_0$, $\epsilon$ for VI
OUTPUT: a policy, $\pi$

//Initialize failure conditions $F$, prediction set $\alpha$ and ruled out prediction set $\omega$
**for all** $(a, att, oClass) \in (S \times \cup_{O \in \mathcal{O}} Att(O) \times \mathcal{O})$ **do**
  $F_{act,att,oClass} \leftarrow \emptyset$
**end for**
$\omega \leftarrow \emptyset$
$\alpha \leftarrow \emptyset$

// Run prediction, learning loop
currentState = $s_0$
**while** Termination conditions not met **do**
  $\mathcal{T} \leftarrow DOORMAXPrediction()$
  $\pi \leftarrow$ ValueIteration($\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \gamma \rangle, \epsilon$)
  nextAction$\leftarrow \pi$(currentState)
  newState $\leftarrow$ nextAction(currentState)
  DOORMAXLearn(currentState, nextAction, newState)
  currentState $\leftarrow$ newState
**end while**
**return** $\pi$

I now discuss in more detail what exactly the prediction and learning phases of DOORMAX amount to.

**DOORMAX Learning:**
DOORMAX's learning is input a state $s$, an action $a$ and a state $s'$. $s'$ was observed to result from taking $a$ in $s$. It then loops over every tuple of object class, attribute pairs. If the values of the attribute for the object class are unchanged in both states then DOORMAX adds the condition representation of $s$ to our failure conditions for this action, object class and attribute triple. If it was not a failure condition for this triple then we update the conditions of any predictions that predict the appropriate effect by taking the $\oplus$ of the prediction's condition and $s$'s condition representation. If there are no such predictions we initialize a new prediction with the corresponding effect and $s$'s condition representation for the new prediction's condition (provided we have not ruled out this sort of prediction as dictated by $\omega$). Lastly, we rule out contradictory object class, attribute, action, effect type tuples: namely those for which there are more than $k$ predictions and those which have predictions with overlapping conditions. Pseudocode can be seen in algorithm 5.

**Algorithm 5** DOORMAXLearn

INPUT: a state $s$, an action $a$ and the state $s'$ that resulted from taking $a$ in $s$ and the state from top-level DOORMAX
OUTPUT: none

**for all** $(oClass, att) \in (\mathcal{O} \times \cup_{O \in \mathcal{O}} Att(O))$ **do**
  //Check for failure condition
  **if** $\underset{o_1 \in s, o_2 \in s', o_1 \text{ and } o_2 \text{ of } oClass}{AND} o_1.att == o_2.att$ **then**
    $F_{a,att,oClass}.add(s.toCondition())$
  **else**
    **for all** hypEffect $\in \text{eff}_{oClass,att}(s, s')$ **do**
      //Check for existing predictions to update
      **if** $\exists$ pred $\in \alpha$ s.t. pred.effect = hypEffect **then**
        pred.condition $\leftarrow$ pred.condition $\oplus$ s.toCondition()
        //Rule out related predictions if there are overlaps
        relatedPreds $\leftarrow \{p \mid p$ is related to pred$\}$
        **if** $\exists$ otherPred $\in$ {relatedPreds - pred} s.t. otherPred.condition matches pred **then**
          $\omega$.add((pred.effect, oClass, att, a))
          $\alpha$.removeAll(relatedPreds)
        **end if**
      **else**
        **if** (hypEffect.type, oClass, att, a) $\notin \omega$ **then**
          newPrediction $\leftarrow (a, s.toCondition, \text{hypEffect})$
          $\omega$.add(newPrediction)
          //Rule out predictions if more than $k$ related predictions
          relatedPreds $\leftarrow \cup p \in \alpha$ s.t. p is related to pred
          **if** |relatedPreds|$> k$ **then**
            $\omega$.add((pred.effect, oClass, att, a))
            $\alpha$.removeAll(relatedPreds)
          **end if**
        **end if**
      **end if**
    **end for**
  **end if**
**end for**

**DOORMAX Learning Example:**
Figures 10 and 11 illustrate how the DOORMAX learning phase takes place for a single set of related predictions. The algorithm is input a tuple of a state (with condition (000100) – i.e. only wallToWestOfTaxi is true), an action, east, and the resulting state of taking east in the initial state (with condition (010000) – i.e. only wallToEastOfTaxi is true) as depicted at the top of Figure 10. Before learning, the set of related predictions for east, taxi, and xLocation consist of two predictions as depicted at the bottom of Figure 10: one has an assignment of 1 effect, one an arithmetic of +1 effect and both have conditions (101011).

Since the two input states are different we do not add to our failure conditions, rather we hypothesize all effects that explain the transition from the initial state to the resulting state – one is an assignment of 1 to the xLocation of the taxi and the other is an arithmetic of adding 1. Since we have no predictions of the former effect we add a new prediction for assignment of 1 to the xLocation of the taxi and since we already have a prediction of the latter effect we update its condition using the $\oplus$ operation. We now have two predictions of a single effect type with contradictory effects so we eliminate these predictions and add to $\omega$. The end result is that we predict east will add 1 to the xLocation of the taxi on (*0****), namely when there is no wall to the east which is exactly what we would hope to predict!

Note that a full iteration of learning on a $(\mathcal{S} \times \mathcal{A} \times \mathcal{S})$ tuple involves repeating this process for every set of

related predictions.

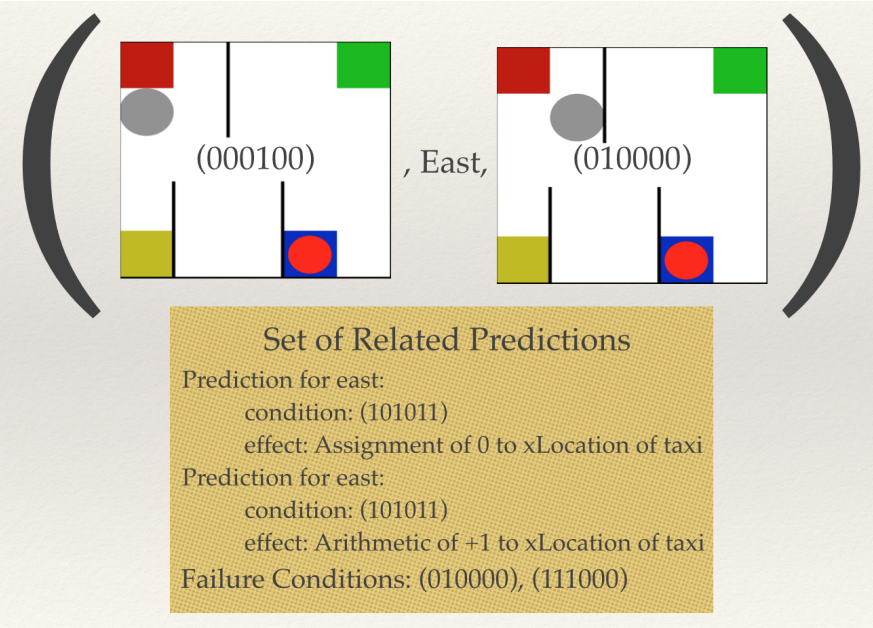



Figure 10: Example of DOORMAX Learning Input with State of One Set of Related Predictions Before Learning

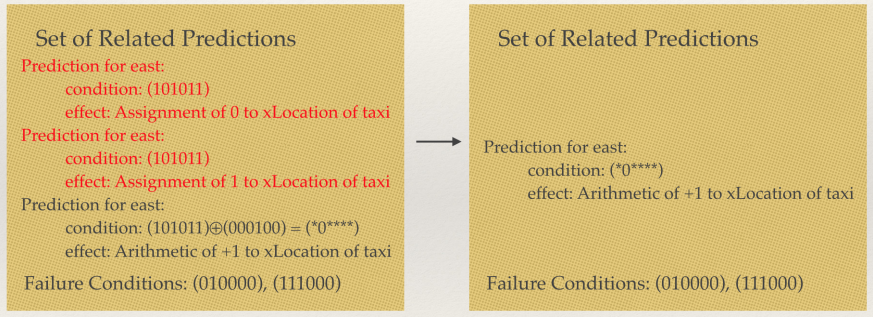



Figure 11: Updating the predictions using the input and initial predictions of figure 10

**DOORMAX Prediction:**
DOORMAX's prediction phase develops a model $\mathcal{T}$ by inferences on every state, action tuple. For each state, action tuple do the following: generate a pool of effects for each appropriate object class, attribute, effect type tuple – if two effects in any pool are contradictory or any pool is empty and there is no associated failure condition for that pool then predict transition to a hallucinatory state with maximal reward, $RMAX$ – else apply all effects of all pools and predict that state for $\mathcal{T}$. Pseudocode for DOORMAX prediction can be found in Algorithm 6. Note that in practice, we need only perform a prediction on a $(\mathcal{S} \times \mathcal{A})$ pair whenever a query is made of our model of $\mathcal{T}$ rather than predict transitions for every $(\mathcal{S} \times \mathcal{A})$ pair to fully develop $\mathcal{T}$. Also note that we need only predict a single state since DOORMAX is only meant to work in deterministic OO-MDPs for every $(\mathcal{S} \times \mathcal{A})$ pair.

**Algorithm 6** DOORMAXPredict

INPUT: The state from top-level DOORMAX
OUTPUT: A model, $\mathcal{T}$
**for all** $(s, a) \in (\mathcal{S} \times \mathcal{A})$ **do**
  totalPool $\leftarrow \emptyset$
  **for all** $(objectClass) \in \mathcal{O}$ **do**
    currentPool $\leftarrow \emptyset$
    **for all** $eType \in \mathcal{Y}$ **do**
      **for all** $att \in Att(\mathcal{O})$ **do**
        currentPool.addAll(pred $\in \alpha$ s.t. pred.condition $\models$ s.toCondition())
      **end for**
      **if** $\exists(e_1, e_2) \in$currentPool such that $e_1$ contradicts $e_2$ **then**
        $\mathcal{T}(s, a, RMAX) = 1$, continue looping over $(s, a)$
      **end if**
      **if** currentPool.isEmpty() and (eType, oClass, att, a) $\notin \omega$ **then**
        $\mathcal{T}(s, a, RMAX) = 1$, continue looping over $(s, a)$
      **end if**
    **end for**
  **end for**
  //Apply all predicted effects
  resultingState $\leftarrow s$
  **for all** effect $\in$ totalPool **do**
    resultingState $\leftarrow$ effect(resultingState)
  **end for**
  $\mathcal{T}(s, a, resultingState) = 1$
**end for**
**return** $\mathcal{T}$

**Prediction examples:**
Figures 12 and 13 illustrate how DOORMAX's prediction phase for a single state, action pair takes place. As shown in Figure 12 the algorithm is input a state (with condition (000100), meaning only wallToWestOfTaxi is true), an action, east(not pictured) and at this phase has two predictions for east. Both conditions of the predictions match the input state condition so both predictions add their effects to the pool of effects for this set of related predictions. These effects do not contradict (they both set the agent's xLocation to 1) despite being of different types so they are both applied to yield to the input state. Assuming that all other effect pools are empty DOORMAX then (correctly) predicts the output state as shown in Figure 13.[14]

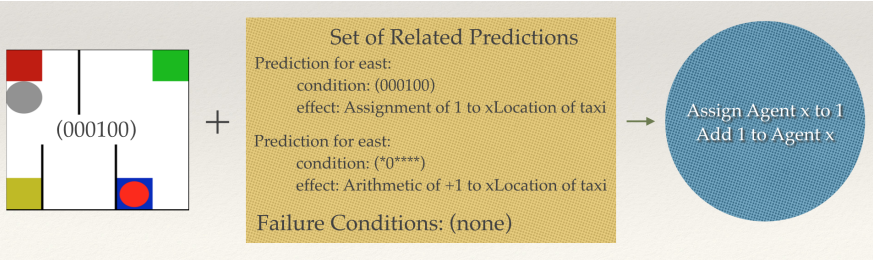



Figure 12: DOORMAX Prediction 1

[14]There are a number of *differences* between my formalization of DOORMAX and that of [11]. In particular, the algorithm presented in [11] only checks for overlapping conditions or an empty pool on what I call the totalPool. This causes the algorithm to fail on even simple taxi domains. I am unsure if this was simply poorly specified pseudocode or else a conceptual mistake on the part of [11] but in any case I present what I have empirically found to be the functioning version of DOORMAX.


Figure 13: DOORMAX Prediction 2

#### 4.2.3 Virtues and Vices

DOORMAX is laudable for a number of its qualities. First, it is a so called Knows-What-It-Knows (KWIK) algorithm[22] meaning that it only ever predicts true transition dynamics if it is not predicting transition to $RMAX$ and fully learns transition dynamics in a polynomial fashion. Moreover, it is noteworthy for its ability to transfer knowledge of transition dynamics across similar states. In particular, when two states have identical conditions if DOORMAX knows the transition dynamics of one for a particular action, then it also knows the transition dynamics of the other for that action. This translates to rapid increases in the rate of learning of transition dynamics which allows DOORMAX to achieve more reward more quickly than RMAX.

Figure 14 demonstrates these rapid speed increases as garnered by my implementation of DOORMAX. I use the taxi domain specified in Section 4.7.1. The x-axis corresponds to number of learning episodes wherein each episode consists of the agent taking actions until the the taxi problem is solved. The y-axis is the number of actions taken across all previous episodes – this directly corresponds to negative reward received up to the current episode given the used uniform negative cost function.

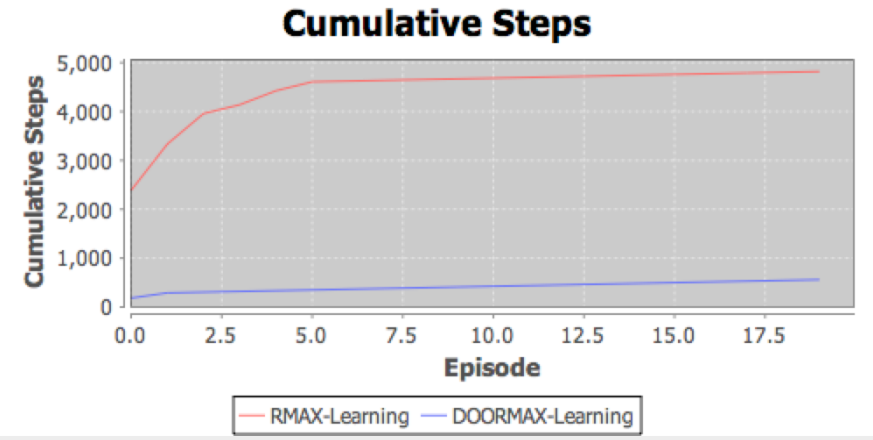



Figure 14: DOORMAX ($k = 2$)VS RMAX on Taxi

However, DOORMAX is not without its vices. What DOORMAX achieves in optimality it very much gives up in generality and ease of use. First of all, not only must all effect types relevant to $\mathcal{T}$ be known and specified to the algorithm but so too must all propositional functions. If either provided set is insufficient the algorithm is often incapable of predicting anything. Moreover, in order for the algorithm to efficiently rule out predictions, $k$ must be set low but not so low that it rules out true predictions. Lastly, as indicated by the name, DOORMAX does not generalize to stochastic domains.

Having covered DOORMAX I now move into my methodology.

## 4.3 Featurizing States without Propositional Functions

### 4.3.1 Problem Setup

My methodology exploits the conditions under which transition dynamics occur as grounds for deriving propositional functions. Consequently a function $v$ which returns a vectorized representation for state $s \in \mathcal{S}$ that is likely to inform transition dynamics is necessary. Moreover, since we are interested in learning general propositional functions we must grapple with not only a single OO-MDP but a domain over which many OO-MDPs are defined. In particular, we would like our vectorized state representation to consist of similar content across OO-MDPs in the same domain. This consistency of meaning across problem instances ensures that $v$ continues to inform transition dynamics across OO-MDPs in a single domain.

Thus we seek a vectorized state representation that satisfies two conditions: (1) it is likely to inform transition dynamics in a particular OO-MDP and (2) it has similar conceptual content across OO-MDPs in the same domain.[15]

### 4.3.2 Problem Solution

Transition dynamics in most any domain are determined by the *relative* values of object attributes. For instance, an agent in grid world fails to move north because it is *1 south* of a wall. Similarly, the taxi in a taxi problem changes the inTaxi attribute of the passenger when it executes the pick up passenger action because both its y and x position are *identical* to those of the passenger. Moreover, transition dynamics are not just relationally determined but they are *minimally* relationally determined: for instance, the agent in grid world cannot go north because the *closest* wall directly north of the agent has a y position *1 greater* than the y position of the agent. My approach leverages these crucial insights.

I begin by defining a set of functions on a state which are parametrized by pairs of real-valued attributes of pairs of object classes, $\mathcal{L} = \{arith_{oClass_1,oClass_2,att_1,att_2}, geom_{oClass_1,oClass_2,att_1,att_2}\}$[16] where $oClass_1, oClass_2 \in \mathcal{O}$ and $att_1 \in Att(oClass_1)$ and $att_2 \in Att(oClass_2)$:

$$arith_{att_1,att_2,oClass_1,oClass_2}(s) = \min_{o_1 \text{ of } oClass_1 \in s, o_2 \text{ of } oClass_2 \in s} o_1.att_1 - o_2.att \tag{17}$$

$$geom_{att_1,att_2,oClass_1,oClass_2}(s) = \min_{o_1 \text{ of } oClass_1 \in s, o_2 \text{ of } oClass_2 \in s} \frac{o_1.att_1}{o_2.att} \tag{18}$$

$arith_{att_1,att_2,oClass_1,oClass_2}(s)$ is simply the minimal arithmetic difference between $att_1$ and $att_2$ for all object instances of $oClass_1$ and $oClass_2$ in $s$. Similarly, $geom_{att_1,att_2,oClass_1,oClass_2}(s)$ is the same but for a geometric difference.

Each element of my vectorized state representation for state $s$ corresponds to a unique evaluation of a function in $\mathcal{L}$ parameterized by a unique pair of object classes and object attributes. Thus, my state representation for state $s$, $v(s)$ is:

$$v(s) = \Big|\Big| \Big|\Big| l_{att_1,att_2,oClass_1,oClass_2}(s) \tag{19}$$
$$(oClass_1,oClass_2,l)\in(\mathcal{O}^2\times\mathcal{L})(att_1,att_2,)\in(Att(oClass_1)\times Att(oClass_2))$$

where $||$ is a concatenation.[17]

This state featurization satisfies both of the aforementioned desired conditions. First, it lends itself to predicting transition dynamics well. As stated earlier, transition dynamics are determined by minimal

[15] Essentially we wish to determine a featurization for a so called agent-space in the terminology of [21].

[16] Note that in experimental results I only made use of *arith* since no used domains actually have any geometric relationships relevant to transition dynamics.

[17] Apologies for the gross notation.

relational values. This featurization is straightforwardly built out of the minimum of relational differences between object instance attributes. Second, it has the same content across domains: each entry in our vectorized state representation constantly refers to the same relational value across OO-MDPs in a single domain. This consistency results from the fact that a particular entry in our vector always corresponds to the evaluation of a function in $\mathcal{L}$ parametrized by some pair of attributes of some pair of object classes and these object classes and attributes are guaranteed to exist across the domain since OO-MDPs in a single domain share object class declarations, $\mathcal{O}$.

## 4.4 Extracting Annotated Datasets from an RL Agent's Observations

### 4.4.1 Problem Setup

Given some set of observations of (initialState, action, resultingState) of a reinforcement learning agent, $O \subseteq (\mathcal{S} \times \mathcal{A} \times \mathcal{S})$ in an OO-MDP setting and a function which featurizes our states $v(s)$, we would like to infer some set of annotated datasets, $D_v$. $d \in D_v$ is an annotated dataset of the form $d = ((v(s_0), b_0), \ldots, (v(s_{|d|}), b_{|d|}))$ where $s_i$ corresponds to an initial state in $O[i][0]$ and $b_i$ is a boolean in $\{\text{true}, \text{false}\}$.

In the next stage we will convert each of these annotated datasets, $d \in D_v$ into a propositional function. Note that this means that we are learning propositional functions in batch. As always, a batch solution can trivially be converted into an online solution by re-running batch learning at each time step as a new observation is received which is essentially what will be done for the modified version of DOORMAX presented.

### 4.4.2 Problem Solution

High level details of my methodology are as follows. I maintain predictions in the DOORMAX sense. I associate with each prediction an annotated dataset. This dataset consists of a set of states featurized according to $v$. Included states are those states that were an initialState in a (initialState, action, resultingState) observation where the action of the observation matches the prediction's action. The associated label of each state is true if the effect of the prediction was observed to take place between initialState and resultingState. Otherwise the label is false. The end result is that each prediction maintains a dataset of states, noting those states under which its effect took place when its action was executed. Since effect occurrence is dictated by OO-MDP transition dynamics generated datasets are closely informed by the transition dynamics of the OO-MDP.

The actual algorithm which involves reworking DOORMAX's learn function works as follows. We loop over each observation in $O$ paired with an object class and an attribute of that object class. For each observation we hypothesize the effects that could explain the changes in the current object class's attribute (as with DOORMAX this may well be an empty set, meaning we at this point we *continue*). If there exists a dataset with an associated prediction which already predicts the current effect we add the featurized initial state to our dataset with a true label. If there is no such dataset we instantiate a new dataset consisting of the current state featurized and labeled with a true and associate with it the appropriate prediction. In the case of instantiating a new dataset, if the number of datasets with predictions related to our new dataset exceeds $k$ (including the new dataset) we eliminate all such datasets and add the current (effect type, object class, attribute, action) tuple to our "ruled-out" set ($\omega$). We also add the current state to all those relevant datasets that were not updated with a label of false. Pseudocode to generate $D_v$ is show in Algorithm 7.

Note that we since we no longer assume the existence of propositional functions for DOORMAX, we now associate a null condition with predictions as they are instantiated. This means we can no longer rule out predictions on the basis of overlapping conditions (though we can still do so on the basis of $k$ as shown in Algorithm 7).

**Algorithm 7** GenerateDataSets()

INPUT: a set of observations $O$, a state featurizer $v$, a $k$ in the DOORMAX sense
OUTPUT: a set of labeled datasets $D_v$

$D_v \leftarrow \emptyset$
$\omega \leftarrow \emptyset$
//Loop over observations
**for all** $(s, a, s') \in O$ **do**
  **for all** $(oClass, att) \in (\mathcal{O} \times \text{Att}(oClass))$ **do**
    updatedDataSets$\leftarrow \emptyset$
    //Loop over hypothesized effects
    **for all** hypEffect $\in \text{eff}_{oClass,att}(s, s')$ **do**
      //Check for dataset with prediction for this effect
      **if** $\exists$ d $\in D_v$ s.t. d.pred.effect = hypEffect **then**
        d.add($(v(s), \text{true})$)
        updatedDataSets.add(d)
      **else**
        **if** (hypEffect.type, oClass, att, a) $\notin \omega$ **then**
          newPrediction $\leftarrow (a, null, \text{hypEffect})$
          newDataSet $\leftarrow \{(v(s), \text{true})\}$
          newDataSet.prediction = newPrediction
          $D_v$.add(newPrediction)
          //Rule out related datasets if more than k related predictions
          relatedDataSets $\leftarrow \cup d \in D_v$ s.t. d.prediction is related to newPrediction
          **if** |relatedDataSets|$> k$ **then**
            $\omega$.add((newPrediction.effect, oClass, att, a))
            $D_v$.removeAll(relatedDataSets)
          **end if**
          updatedDataSets.add(d)
        **end if**
      **end if**
    **end for**
    //Update all appropriate datasets that did not receive a true with a false
    **for all** d $\in D_v$ and $\notin$ updatedDataSets s.t. d.prediction.action $= a$ **do**
      d.add($(v(s), \text{false})$)
    **end for**
  **end for**
**end for**
**return** $D_v$

## 4.5 Inferring Propositional Functions from Annotated datasets

### 4.5.1 Problem Setup

Given a set of labeled datasets $D_v$ where each dataset $d \in D_v$ consists of states in $\mathcal{S}$ featurized according to some featurization function $v$ with labels of either true or false, we would like to generate some set of propositional functions – i.e. we would like to generate some set of functions which maps from $\mathcal{S}$ to $\{\text{true}, \text{false}\}$

### 4.5.2 Problem Solution

My solution is very straightforward. I treat each dataset as a supervised learning dataset for learning a single propositional function. A separate binary classifier is trained on each dataset. This classifier's classify routine, in turn, acts as a propositional function: it classifies novel state $s$ as true or false. In particular, I use a J48 decision tree implementation as detailed by [25] though other methods can straightforwardly be swapped in (and I have experimented a good deal with a logistic regression implementation for this task).

Thus the end result is $|D_v|$ classifiers whose classify routines define $|D_v|$ propositional functions.

## 4.6 DOORMAX with Learned Propositional Functions

DOORMAX normally requires propositional functions to determine the conditions under which an effect is thought to take place. These conditions are constructed out of those propositional functions supplied to the OO-MDP. The conditions can be thought of as propositional functions unto themselves which return true when they match a state's corresponding condition. I modify DOORMAX by replacing entire conditions with propositional functions learned in my framework.

DOORMAXLearn is modified as follows. All predictions now have an associated dataset of states featurized according to the featurizing function $v$ detailed in 4.3.2 in which featurized states are labeled with a true or false. Those included states in a predictions dataset are those which were states from which action $a$ was observed to be taken where action $a$ is the action of the prediction. A state is labeled with a true for a given state in a prediction's dataset whenever the prediction's effect was observed in the resulting state. A state is labeled with a false in a prediction's dataset if the effect of the prediction did not take place in the observed resulting state. The rest of the DOORMAX learn machinery runs as usual with the exception that, as in the case of producing datasets, we can no longer rule out predictions on the basis of overlapping conditions.

Full pseudocode is shown in Algorithm 8.

**Algorithm 8** DOORMAXLearnWithoutPFs()

INPUT: a state $s$, an action $a$ and the state $s'$ that resulted from taking $a$ in $s$ and the state from top-level DOORMAX
OUTPUT: none

```
for all (oClass, att) ∈ (O × Att(oClass)) do
  updatedPredictions← ∅
  //Loop over hypothesized effects
  for all hypEffect ∈ eff_{oClass,att}(s, s') do
    //Check for prediction for this effect
    if ∃ pred ∈ ω s.t. pred.effect = hypEffect then
      pred.d.add((v(s), true))
      updatedPredictions.add(pred)
    else
      if (hypEffect.type, oClass, att, a) ∉ ω then
        newDataSet ← {(v(s), true)}
        newPrediction ← (a, null, hypEffect)
        newPrediction.dataSet = newDataSet
        ω.add(newPrediction)
        //Rule out predictions if more than k predictions
        relatedPredictions ← ∪p ∈ ω s.t. p is related to newPrediction
        if |relatedPredictions|> k then
          ω.add((newPrediction.effect, oClass, att, a))
          ω.removeAll(relatedDataSets)
        end if
        updatedPredictions.add(newPrediction)
      end if
    end if
  end for
  //Update all appropriate datasets that did not receive a true with a false
  for all p ∈ ω and ∉ updatedPredictions s.t. p.action = a do
    p.d.add((v(s), false))
  end for
end for
```

DOORMAXPredict is very straightforwardly modified. We no longer have a condition associated with our predictions which we can check against the condition representation of our current state. However, we do have a dataset for each prediction labeled with those states (as featurized by $v$) in which the prediction's effect is known to take place. Thus, rather than checking if a given prediction's condition matches that of the current state we now train a classifier on its dataset and add the prediction's effect to the pool of effects if the trained classifier returns true for the current state. I use a J48 decision tree for my classifier. Full pseudocode is shown in Algorithm 9.

**Algorithm 9** DOORMAXPredictWithoutPFs()

```
INPUT: The state from top-level DOORMAX
OUTPUT: A model, 𝒯
for all (s, a) ∈ (𝒮 × 𝒜) do
  totalPool ← ∅
  for all (objectClass) ∈ 𝒪 do
    currentPool ← ∅
    for all eType ∈ 𝒴 do
      for all att ∈ Att(𝒪) do
        for all pred ∈ α s.t. pred.action == a and pred.effect.effectType == eType do
          classifier = J48(pred.d)
          if classifier.classify(s) then
            currentPool.add(pred.effect)
          end if
        end for
      end for
      if ∃(e1, e2) ∈currentPool such that e1 contradicts e2 then
        𝒯(s, a, RMAX) = 1, continue looping over (s, a)
      end if
      if currentPool.isEmpty() and (eType, oClass, att, a) ∉ ω then
        𝒯(s, a, RMAX) = 1, continue looping over (s, a)
      end if
    end for
  end for
  //Apply all predicted effects
  resultingState ← s
  for all effect ∈ totalPool do
    resultingState ← effect(resultingState)
  end for
  𝒯(s, a, resultingState) = 1
end for
return 𝒯
```

A modified version of DOORMAX which requires no propositional functions can be run by running classic DOORMAX with its learn and predict routines replaced by Algorithms 8 and 9 respectively.

## 4.7 Experimental Results

### 4.7.1 Taxi Domain Used

I define the taxi domain used in experiments as follows:

$\mathcal{A}$ : { taxiMoveNorth, taxiMoveEast, taxiMoveSouth, taxiMoveWest, pickupPassenger, dropOffPassenger}

Note that pickupPassenger causes a passenger at the same location of the taxi to move with the taxi until dropOffPassenger is executed. Only one passenger may occupy a single location.

$\mathcal{O}$ : { goalLocation (xLocation, yLocation, goalType), taxi(xLocation, yLocation, passengerInTaxi), passenger(xLocation, yLocation, goalType, inTaxi), verticalWall(wallOffSet, bottomOfWall, topOfWall), horizontalWall (wallOffSet, leftStartOfWall, rightStartOfWall) }

Where values in parenthesis indicate attributes of each of the object classes. Note that goalLocations are locations that passengers with matching goalTypes are trying to reach. The offset of vertical walls indicates the horizontal offset from the origin whereas the bottomOfWall and topOfWall indicate the start and end y

positions of the wall. Symmetric clarification applies to horizontal walls. All attributes' domain range over $\mathcal{Z}$.

$\mathcal{P}$ : { wallToNorthOfTaxi, wallToEastOfTaxi, wallToSouthOfTaxi, wallToWestTaxi, passengerInTaxi, taxiAtPassenger }

Note that passengerInTaxi is true if there is some passenger in the taxi and taxiAtPassenger is true if the taxi is at a passenger location regardless of whether or not the passenger is in the taxi. Also keep in mind that my algorithms do not have access to this $\mathcal{P}$.

$\mathcal{R}$ : uniform -1 for all states.

$\mathcal{T}$ : Transitions are deterministic and actions result in exactly what one would expect.

$\gamma$ : .95

I define the *classic taxi OO-MDP initial state* as follows:

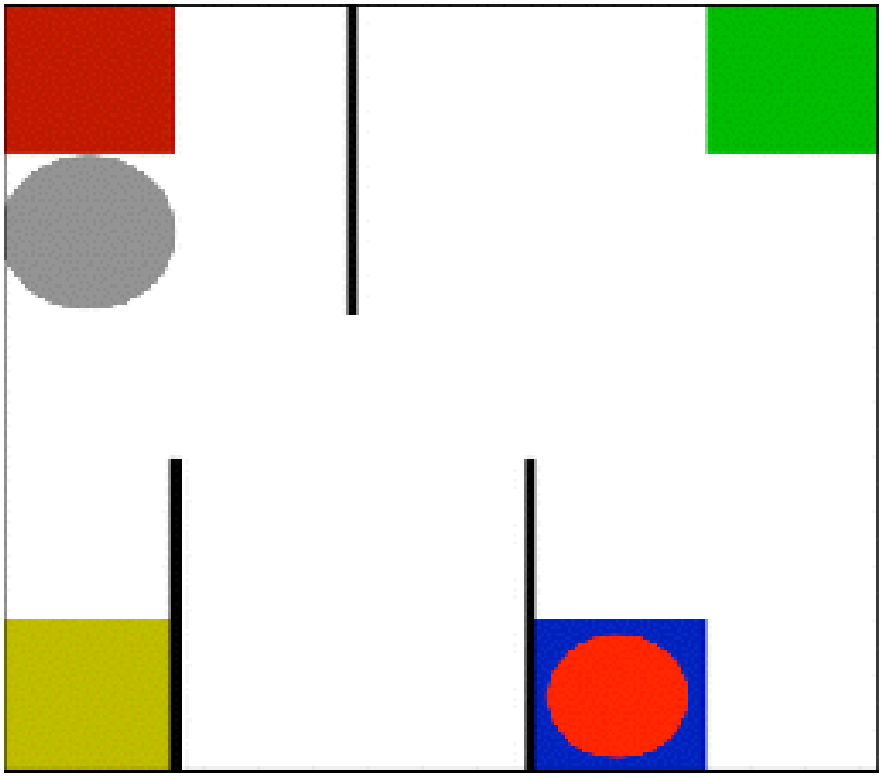


Figure 15: The classic taxi domain initial state

The *classic taxi OO-MDP state space* is all those states reachable from the classic taxi OO-MDP initial state using $\mathcal{A}$ according to $\mathcal{T}$.

State spaces for OO-MDPs in my taxi domain are defined by those states reachable by $\mathcal{A}$ according to $\mathcal{T}$ from an initial state – this initial state is identical to tthe classic taxi OO-MDP initial state but with the three vertical walls, all passengers' initial positions, all goal locations' positions and the taxi's initial position randomized within the boundaries of the map. The width and height of the map also independently vary between 5 and 30 (the map is not necessarily square) for initial states.

### 4.7.2 Agreement with Existing Propositional Functions

I demonstrate correspondence between learned propositional functions according to my methodology and those normally expert provided for the taxi domain. I use taxi domain because 4 of those propositional functions normally supplied for the OO-MDP (i.e. $\mathcal{P}$) directly correspond to 4 propositional functions that will be learned by method – these are the wallToEast, wallToWest, wallToSouth and wallToNorth of agent propositional functions. This correspondence results from the fact that the conditions for 4 predictions in the DOORMAX framework are uniquely determined by one of these propositional functions in $\mathcal{P}$: for instance, the prediction that north adds one to the agent's y position has a condition that only depends on the wallToNorth propositional function. This direct correspondence allows me to readily evaluate how well the learned propositional functions perform by simply examining how well they align with their corresponding propositional functions.

**Experimental Setup:**

I conduct two sets of experiments in my taxi domain. In one the agent is allowed to learn propositional functions based on 10,000 consecutive observations of (initialState, action, resultingState) $\in (\mathcal{S} \times \mathcal{A} \times \mathcal{S})$ in the classic taxi state. In the other the agent samples 10,000 observations of the result of random actions on random states. The random states are sampled uniformly from the union of all states of all OO-MDPs in the domain.

$|D_v| = 6$ in both cases and I examine error rates for the 4 propositional functions which correspond to the a wall being adjacent to the taxi.

Generalization error quantification for both experiments involves 10 trials of sampling 1,000 states uniformly from the union of all states defined by all OO-MDPs in the domain (for 10,000 states total) and measuring the percentage of states for which learned functions predict something different from their corresponding propositional function in $\mathcal{P}$.

**Experimental Results:**

When propositional functions are learned only on the classic taxi state space and then tested across the taxi domain, I observe reasonable correspondence between propositional functions in $\mathcal{P}$ and their corresponding learned propositional functions. The wallToNorth and wallToSouth propositional functions are learned noticeably worse than the wallToEast and wallToWest propositional functions. I hypothesize that this is a result of learned propositional functions for wallToNorth and wallToSouth overfitting to the classic state space since the only horizontal walls in the classic state space are those at the boundaries of the map. For example, the agent may learn that whenever the difference between its y position and a fixed goalLocation's y position dictates that the passenger is at the top of the map, the wallToNorth propositional function should be true. Obviously this will not generalize well across the domain. This is not the case for wallToWest and wallToEast for which there are 3 non-boundary vertical walls which force the agent to learn more generalized propositional functions. Full results are detailed in Figure 16.

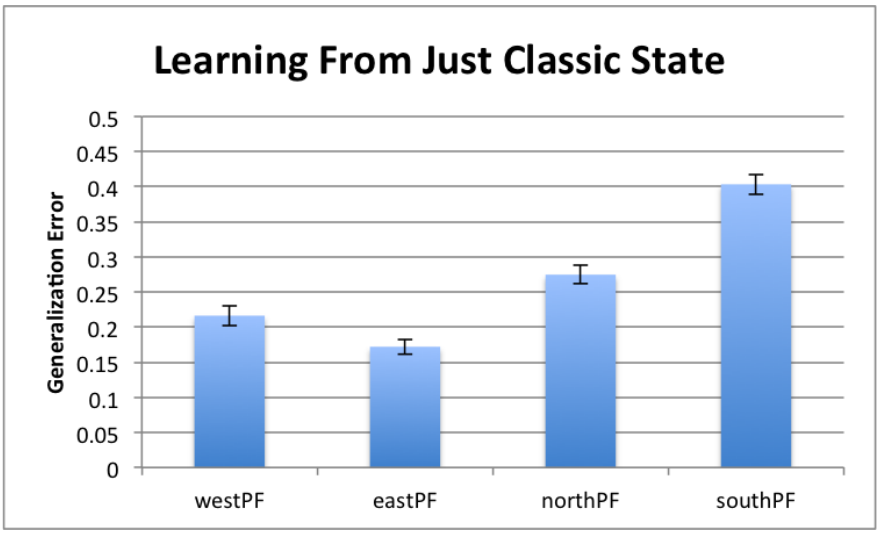



Figure 16: Generalization error for learned propositional functions when learning only on the classic taxi state

When propositional functions are learned across the taxi, we observe a strong correspondence between propositional functions in $\mathcal{P}$ and their corresponding learned propositional functions. The wallToNorth and wallToSouth propositional functions are learned appreciably perfectly. I attribute this to the fact that there are no horizontal walls in my taxi domain other than those at the top or bottom of the map, making learning adjacency to horizontal walls particularly easy when samples are provided across the entire domain. Full results are detailed in Figure 17;

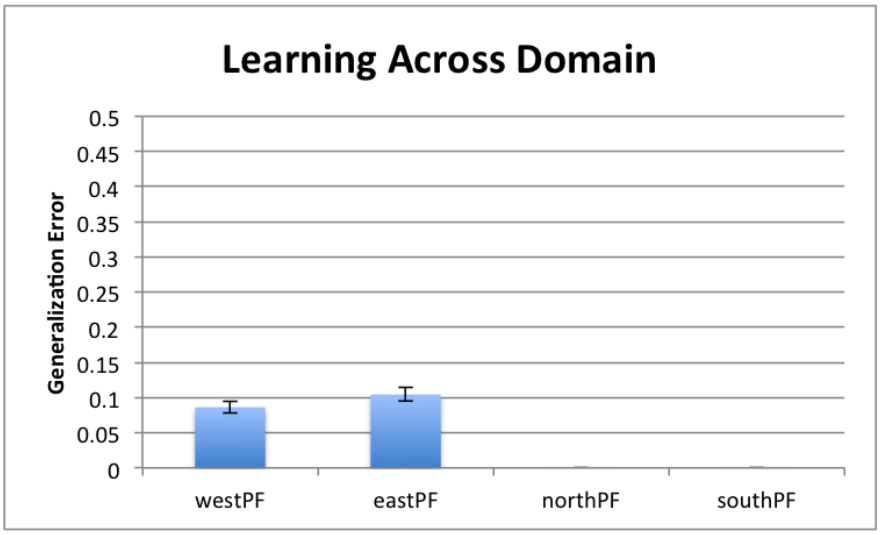



Figure 17: Generalization error for learned propositional functions when learning across the full taxi domain

### 4.7.3 DOORMAX with Learned Propositional Functions

I now test my modified version of DOORMAX which, rather than requiring expert propositional functions, learns the necessary propositional functions.

**Experimental Setup:**

I conduct experiments in a taxi domain problem with the classic taxi domain state space as detailed in Section 4.7.1 .

I run tabular RMAX, classic DOORMAX, as well as this modified DOORMAX. $k = 2$ for both versions of DOORMAX. Each algorithm is allowed to run until a learning episode terminates by arriving in the terminal state. 10 learning episodes are run for all three algorithms. The number of cumulative steps for all previous learning episodes is reported for each learning episode. Since a uniform negative reward is used, the number of cumulative steps corresponds to cumulative cost received (i.e. it is inversely proportional to cumulative reward). When the slope of this plot ceases to change the algorithm has converged on an optimal policy.

**Experimental Results:**
RMAX takes approximately 4,500 actions to converge on the optimal policy which we might reasonably expect given its tabular and slow nature. DOORMAX with expert-provided propositional functions takes approximately 260 actions and my DOORMAX without any expert-provided propositional functions takes about 350 actions. DOORMAX without expert-provided propositional functions, then, performs nearly as well as DOORMAX! The disparity in performance is likely due to the inability of DOORMAX without propositional functions to rule out predictions on the basis of overlapping conditions as classic DOORMAX does. Full results are shown in Figure 18. Figure 19 shows the same results but zoomed in on DOORMAX in the first iteration to better visualize the difference between DOORMAX with learned vs expert provided propositional functions .

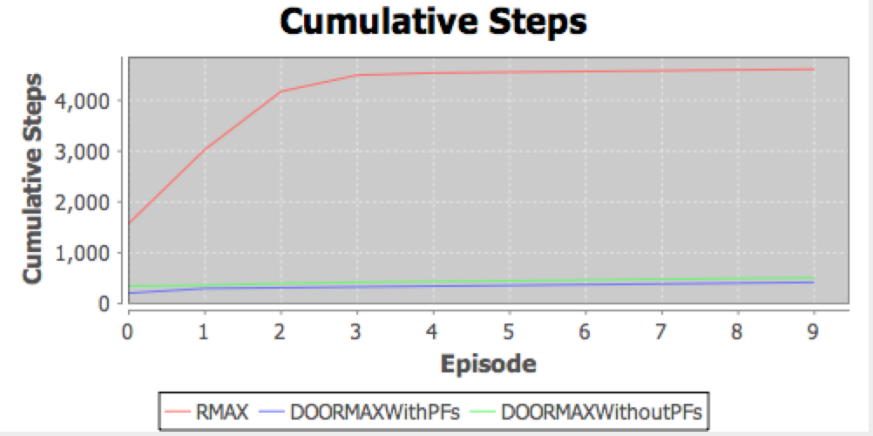



Figure 18: RMAX(RMAX) vs classic DOORMAX (DOORMAXWITHPFs) VS DOORMAX with learned propositional functions(DOORMAXWithoutPFs)

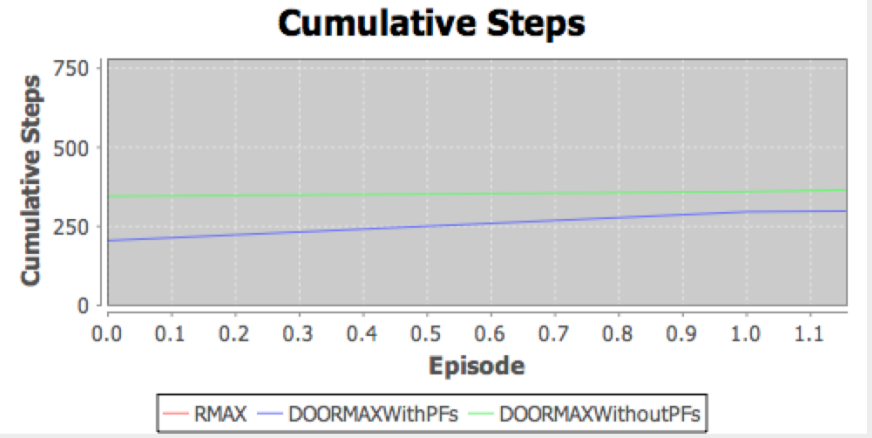



Figure 19: RMAX(RMAX) vs classic DOORMAX (DOORMAXWITHPFs) VS DOORMAX with learned propositional functions(DOORMAXWithoutPFs) zoomed in on the first learning episode

# 5 Conclusion

I present a means of leveraging propositional functions to greatly reduce the number of states which a planner must examine in a large state space. Those propositional functions used are those accommodated by the OO-MDP problem representation. These propositional functions are used to estimate a distribution on the optimality of actions on a state by state basis. If an action is unlikely to be optimal in a state it is barred from the planner. The result is that the planner only considers a small subspace of the large state space.

This methodology is notable for its generalness: once an expert provides model parameters or learning is performed in small state spaces for a domain, the method can be applied to any OO-MDPs in the domain. Learning can also take place over a series of tasks without a collapse to a uniform prior and minimal action pruning since the priors are goal specific. This methodology yields remarkable empirical results: as demonstrated in both a Minecraft and robotics domain, it reaps significant speedups in terms of runtime and the quality of the inferred policy.

In future work, we hope to automatically discover useful state space specific subgoals online—a topic of some active research [23, 9]. Subgoals would synthesize extremely well with our pruning methodology given the goal-based nature of the priors we use to prune. Another promising direction to explore is an on-line approach to learning, as opposed to the batch style presented here. Under an online paradigm, the agent would modify its prior over action optimality after each action execution. We are also investigating methods to stochastically prune actions rather than requiring a hard threshold parameter to be defined.

I also present a means of automatically learning propositional functions for an OO-MDP. This method performs a DOORMAX-like learn function over a set of (initialState, action, resultingState) observations of an RL agent. While learning takes place it associates with every prediction a dataset of those initialStates whose actions match that of the prediction. These states are featurized according to minimal relational values of object class' attributes. Additionally, each state is labeled with a true if the resultingState is explained by the prediction'ss associated effect. Lastly, each dataset associated with a non-eliminated prediction at the end of learning is converted into a propositional function using any standard supervised learning technique – I use J48 decision trees. The end result is a set of propositional functions closely informed by the transition dynamics of the OO-MDP. I also demonstrate how given the intimate relationship between these learned propositional functions and the transition dynamics of the OO-MDP, these propositional functions can supplant those normally provided to DOORMAX. Empirical results are gathered in taxi domain and include demonstrating correspondence between learned propositional functions and those normally expert provided as well as the near DOORMAX level performance of my DOORMAX that runs without propositional functions.

In future work I hope to explore the extent to which these learned propositional functions can be utilized. In particular I am interested in applying these learned propositional functions to the many algorithms which exploit propositional functions for planning and reinforcement learning – our action pruning method included.

# References


[1] David Abel, D Ellis Hershkowitz, Gabriel Barth-Maron, Stephen Brawner, Kevin O'Farrell, James MacGlashan, and Stefanie Tellex. Goal-based action priors. In *Proceedings of the 25th International Conference on Automated Planning and Scheduling*, 2015.

[2] D. Andre and S.J. Russell. State abstraction for programmable reinforcement learning agents. In *Eighteenth national conference on Artificial intelligence*, pages 119–125. American Association for Artificial Intelligence, 2002.

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

[4] Mario Bollini, Stefanie Tellex, Tyler Thompson, Nicholas Roy, and Daniela Rus. Interpreting and executing recipes with a cooking robot. In *Experimental Robotics*, pages 481–495. Springer, 2013.

[5] Blai Bonet and Hector Geffner. Labeled rtdp: Improving the convergence of real-time dynamic programming. In *ICAPS*, volume 3, pages 12–21, 2003.

[6] Ronen I. Brafman and Moshe Tennenholtz. R-max - a general polynomial time algorithm for near-optimal reinforcement learning. *J. Mach. Learn. Res.*, 3:213–231, March 2003.

[7] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[8] T. Croonenborghs, K. Driessens, and M. Bruynooghe. Learning relational options for inductive transfer in relational reinforcement learning. *Inductive Logic Programming*, pages 88–97, 2008.

[9] Özgür Şimşek, Alicia P. Wolfe, and Andrew G. Barto. Identifying useful subgoals in reinforcement learning by local graph partitioning. In *Proceedings of the 22Nd International Conference on Machine Learning*, pages 816–823, 2005.

[10] Thomas G Dietterich. Hierarchical reinforcement learning with the maxq value function decomposition. *J. Artif. Intell. Res.(JAIR)*, 13:227–303, 2000.

[11] Carlos Diuk, Andre Cohen, and Michael L. Littman. An object-oriented representation for efficient reinforcement learning. In *Proceedings of the 25th International Conference on Machine Learning*, ICML '08, pages 240–247, New York, NY, USA, 2008. ACM.

[12] Kutluhan Erol, James Hendler, and Dana S Nau. Htn planning: Complexity and expressivity. In *AAAI*, volume 94, pages 1123–1128, 1994.

[13] Alborz Geramifard, Thomas J. Walsh, Nicholas Roy, and Jonathan P. How. Batch-ifdd for representation expansion in large mdps. *CoRR*, abs/1309.6831, 2013.

[14] Alborz Geramifard, Thomas J. Walsh, Stefanie Tellex, Girish Chowdhary, Nicholas Roy, and Jonathan P. How. A tutorial on linear function approximators for dynamic programming and reinforcement learning. *Foundations and Trends? in Machine Learning*, 6(4):375–451, 2013.

[15] 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. Morgan Kaufmann Publishers Inc., 1998.

[16] Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006.

[17] Nicholas K. Jong. The utility of temporal abstraction in reinforcement learning. In *Proceedings of the Seventh International Joint Conference on Autonomous Agents and Multiagent Systems*, 2008.

[18] R.A. Knepper, S. Tellex, A. Li, N. Roy, and D. Rus. Single assembly robot in search of human partner: Versatile grounded language generation. In *Human-Robot Interaction (HRI), 2013 8th ACM/IEEE International Conference on*, pages 167–168, March 2013.

[19] G. Konidaris and A. Barto. Efficient skill learning using abstraction selection. In *Proceedings of the Twenty First International Joint Conference on Artificial Intelligence*, pages 1107–1112, 2009.

[20] G. Konidaris, I. Scheidwasser, and A. Barto. Transfer in reinforcement learning via shared features. *The Journal of Machine Learning Research*, 98888:1333–1371, 2012.

[21] George Konidaris and Andrew Barto. Building portable options: Skill transfer in reinforcement learning. In *Proceedings of the International Joint Conference on Artificial Intelligence*, IJCAI '07, pages 895–900, January 2007.

[22] Lihong Li, Michael L Littman, Thomas J Walsh, and Alexander L Strehl. Knows what it knows: a framework for self-aware learning. *Machine learning*, 82(3):399–443, 2011.

[23] Amy Mcgovern and Andrew G. Barto. Automatic discovery of subgoals in reinforcement learning using diverse density. In *In Proceedings of the eighteenth international conference on machine learning*, pages 361–368. Morgan Kaufmann, 2001.

[24] Mojang. Minecraft. http://minecraft.net, 2014.

[25] Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers, San Mateo, CA, 1993.

[26] Balaraman Ravindran and Andrew Barto. An algebraic approach to abstraction in reinforcement learning. In *Twelfth Yale Workshop on Adaptive and Learning Systems*, pages 109–144, 2003.

[27] Benjamin Rosman and Subramanian Ramamoorthy. What good are actions? accelerating learning using learned action priors. In *Development and Learning and Epigenetic Robotics (ICDL), 2012 IEEE International Conference on*, pages 1–6. IEEE, 2012.

[28] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2 edition, 2003.

[29] Richard S Sutton, Doina Precup, and Satinder Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1):181–211, 1999.