Brown University
Department of Computer Science

Learning to Plan in Complex Stochastic Domains

Author:
David Abel

Supervisor:
Prof. Stefanie Tellex

ScM Project

May 2015

Abstract

Probabilistic planning offers a powerful framework for general problem solving. Historically, probabilistic planning algorithms have contributed to a variety of critical application areas and technologies, including conservation biology [42], self-driving cars [48, 37], and space exploration [14, 6, 20]. However, optimal planning is known to be P-Complete with respect to the size of the state-action space [41], imposing a harsh constraint on the types of problems that may be solved in real time.

One way of enabling probabilistic planning algorithms to efficiently solve more complex problems is to provide knowledge about the task of interest; this knowledge might be in the form of an abstracted representation that reduces the dimensionality of the problem, or a heuristic that biases exploration toward the optimal solution. Unfortunately, these forms of knowledge are highly specific to the particular problem instance, and often require significant reworking in order to transfer between slight variants of the same task. As a result, each new problem instance requires a newly engineered body of knowledge.

Alternatively, we propose *learning to plan*, in which planners acquire useful domain knowledge about how to solve families of related problems from a small training set of tasks, eliminating the need for hand engineering knowledge. The critical insight is that problems that are too complex to solve efficiently often resemble much simpler problems for which optimal solutions may be computed. By extracting relevant characteristics of the simple problems' solutions, we develop algorithms to solve the more complex problems by learning about the structure of optimal behavior in the training tasks.

In particular, we introduce *goal-based action priors* [2], that guide planners according to which actions are likely to be useful under different conditions. The priors are informed during a training stage in which simple, tractable tasks are solved, and whose solutions inform the planner about optimal behavior in much more complex tasks from the same domain. We demonstrate that goal-based action priors dramatically reduce the time taken to find a near-optimal plan compared to baselines, and suggest that *learning to plan* is a compelling means of scaling planning algorithms to solve families of complex tasks without the need for hand engineered knowledge.

Acknowledgements

This work would not have been possible without the help of my Advisor Professor Stefanie Tellex, whose guidance, support, and wealth of knowledge were essential for carrying out this research. Additionally, I would like to thank Ellis Hershkowitz, James MacGlashan, and Gabriel Barth-Maron for their critical contributions to this project, and for the many wonderful discussions that led to the central ideas introduced in this document. A special thanks to my mother, father, and brother for letting me bounce ideas off of them for the past two years, and for always supporting me. Lastly, I would like to thank the other members of the Humans to Robots laboratory, including David Whitney, Dilip Arumagum, Emily Wu, Greg Yauney, Izaak Baker, Jeremy Joachim, John Oberlin, Kevin O'Farrell, Professor Michael Littman, Miles Eldon, Nakul Gopalan, Ryan Izant, and Stephen Brawner.

Contents

1 Introduction

Planning offers a powerful framework for general problem solving. Historically, planning algorithms have contributed to a variety of critical application areas and technologies, ranging from conservation biology [42] to self-driving cars [48, 37] to space exploration [14, 6, 20]. It is not altogether surprising that planning algorithms have contributed to such disparate fields - the planning framework is an extremely versatile approach to generic problem solving.

Ultimately our goal is to deploy planning algorithms onto systems that operate in highly complex environments, such as the actual world, and not just in deterministic simulations. As a result, this investigation focuses on *probabilistic* planning algorithms in order to better capture the stochasticity inherent in our experience of reality. Critically, classical deterministic planning is equivalent to the problem of search in a graph, while probabilistic planning assumes that inter-state transitions (i.e. edge traversals) are non-deterministic. That is, if our algorithm intended to traverse the edge between state u and v, with some non-zero probability the environment may instead transition to a different state, w. Consequently, the problem of probabilistic planning is significantly more difficult than deterministic planning, but allows for more accurate models of the real world (and other domains of interest).

1.1 Planning as Sequential Decision Making

Probabilistic planning problems may be formulated as a stochastic sequential decision making problem, modeled as a Markov Decision Process (MDP). In these problems, an agent must find a mapping from states to actions for some subset of the state space that enables the agent to maximize reward over the course of the agent's existence. Of particular interest are Goal-Directed MDPs, whose execution terminates when the agent reaches a terminal or goal state. We treat the problem of an agent operating in an Goal-Directed MDP as equivalent to the probabilistic planning problem.

Computing optimal solutions to MDPs is known to be P-Complete with respect to the size of the state-action space [41] imposing a harsh constraint on the types of problems that may be solved in real time. Furthermore, the state-action spaces of many problem spaces of interest grow exponentially with respect to the number of objects in the environment. For instance, when a robot is manipulating objects, an object can be placed anywhere in a large set of locations. The size of the state space explodes exponentially with the number of objects and locations, which limits the placement problems that the robot is able to expediently solve. Bellman called a version of this problem the "curse of dimensionality" [10].

To confront the state-action space explosion that naturally accompanies difficult planning tasks, prior work has explored giving the agent prior knowledge about the task or domain, such as options [47] and macro-actions [12, 39]. However, while these methods allow the agent to search more deeply in the state space, they add non-primitive actions to the planner which *increase* the branching factor of the state-action space. The resulting augmented space is even larger, which can have the paradoxical effect of increasing the search time for

a good policy [27]. Deterministic forward-search algorithms like hierarchical task networks (HTNs) [38], and temporal logical planning (TLPlan) [4, 5], add knowledge to the planner that greatly increases planning speed, but do not generalize to stochastic domains, and require a great detail of hand-engineered, task-specific knowledge.

In this work, we develop a general framework for *learning to plan*, in which a planning algorithm is given access to optimal solutions to simple tasks, and then asked to solve complex tasks from the same domain. The key insight is that the planner may transfer knowledge about optimal behavior from the simple tasks to the more difficult tasks. In the psychology literature, this concept has been called *scaffolding*, introduced by [50]. To support this strategy, we introduce the notion of a Task Generator, that defines a distribution over tasks belonging to a given domain, subject to a set of constraints.

To demonstrate the power of scaffolding, we develop *goal-based action priors*, which maintain a probability distribution on the optimality of each action relative to the agent's current state. During training, the agent is able to query for optimal behavior in a series of simple tasks - the results of these queries are used to inform the priors. During testing, the agent uses the priors to prune away irrelevant actions in each explored state, consequently reducing state-action space exploration while still finding a near optimal policy.

We evaluate our approach in the 3D blocks world Minecraft. Minecraft is a voxel-based simulation in which the user-controlled agent can place, craft, and destroy blocks of different types. Due to its complexity, Minecraft serves as a compelling simulator for the real world. The Minecraft world is rich enough to offer many interesting challenges, but still gives designers of the tasks full control over what information the agent has access to. Additionally, the massive space of blocks allows for a gradual increase in problem complexity (with AI-Complete tasks in the upper limit), but also allows for simple challenges like classical 2D Grid World (i.e. don't let the Minecraft agent jump). We conduct experiments on several difficult problem types, including constructing a bridge over a trench, digging underground to find a gold block, destroying a wall to reach a goal, collecting gold ore and smelting it in a furnace, and traversing vast, lava-covered terrain to find a goal.

1.2 Domains of Interest

We use Minecraft as a test bed due to its relation to real world problems of interest. These include robotic navigation tasks that involve manipulation of the environment, such as pressing buttons, opening doors, and moving obstacles, as well as tackling more general problem solving strategies that include planning with 3D printers and programmable matter; a composite robot-3D printer system would dramatically increase the scope of what robots can achieve. Robots could construct entire buildings on other planets, such as structures that offer protection from the harsh environments of foreign-atmospheres. The European Space Agency is already investigating using 3D printers to construct protective domes on the moon [17, 18] – however, a 3D printer alone is stationary. The physical capabilities of a robot combined with the tool generation of a 3D printer offer many compelling advances in space exploration. If a part breaks on Mars, we need not send another entire mission to Mars, our robot can simply print another one for use in construction tasks. However, the space of

printable objects is so massive that searching through possible futures is computationally intractable, calling for an advanced planning system that can reason over huge spaces. Generally, we are interested in domains in which a decision making agent has a lot of power to manipulate the environment. This often translates to a large action space, but may also include environments that contain many objects, resulting in a exponential number of possible configurations of the objects involved.

2 Background

Planning for optimal behavior in the real world must account for the uncertainty inherent in our experience of reality. As a result, probabilistic planning problems may be viewed as a stochastic sequential decision making problem, formalized as a Markov Decision Process (MDP).

2.1 Markov Decision Processes

A specific instance of a Markov Decision Process defines a probabilistic planning problem.

Definition 1. *A* Markov Decision Process (MDP) *is a five-tuple:* $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \gamma \rangle$*, where:*

- $\mathcal{S}$ *is a finite set of states, also called the* state space*.*
- $\mathcal{A}$ *is a finite set of actions, also called the* action space*.*
- $\mathcal{T}$ *denotes* $\mathcal{T}(s' \mid s, a)$*, the transition probability of an agent applying action* $a \in \mathcal{A}$ *in state* $s \in \mathcal{S}$ *and arriving in state* $s' \in \mathcal{S}$
- $\mathcal{R} : \mathcal{S} \mapsto \mathbb{R}$ *denotes the real valued reward received by the agent for occupying state* s*.*
- $\gamma \in [0, 1)$ *is a discount factor that defines how much the agent prefers immediate rewards over future rewards (the agent prefers to maximize immediate rewards as* γ *decreases).*

A solution to an MDP is referred to as a *policy*, which we denote, π.

Definition 2. *A* Policy*, denoted '*π*', is a mapping from a state* $s \in \mathcal{S}$ *to an action* $a \in \mathcal{A}$*.*

Solutions to MDPs are policies. That is, a planning algorithm that solves a particular MDP instance returns a policy π. We can evaluate a policy according to its associated value function:

Definition 3. *A* Value Function $V^{\pi}(s)$ *is the expected cumulative reward an agent receives from occupying state* s *and following policy* π *thereafter.* For the above definition of an MDP, the Value Function associated with following policy π from state s onward is:

$$V^{\pi}(s) = \mathbf{E}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \;\middle|\; s_t = s\right] \tag{1}$$

Definition 4. *The* optimal value function V^**, (also referred to as the Bellman Equation), is:*

$$V^*(s) = \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} \mathcal{T}(s' \mid s, a) \left[\mathcal{R}(s') + \gamma V^*(s) \right] \tag{2}$$

We also introduce a *Q-function*, which is relative to a state-action pair:

Definition 5. *A* Q-Function*, $Q^\pi(s, a)$, is the value of taking action a in state s and following policy π thereafter.* Again, for the above definition of an MDP, the Q-Function associated with following policy π, starting in state s and applying action a is:

$$Q^\pi(s, a) = \mathbf{E} \left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \;\middle|\; s_t = s, a_t = a \right] \tag{3}$$

2.2 Solving Markov Decision Processes

There are a variety of methods for solving MDPs. Here we introduce some of the basic methods, with a special emphasis on those methods used during our experiments.

2.2.1 Value Iteration

Value Iteration [11] effectively takes the Bellman Equation from Equation 2 and turns it into an update rule. The agent considers applying each action in each state, and performs a full update on its approximation of the value function. Value Iteration terminates when the change in the value function from one iteration to the next is tiny (i.e. below some provided valued ε). Psuedocode for Value Iteration is provided in Algorithm 1.

Algorithm 1 Value Iteration

INPUT: An MDP instance, $M = \langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \gamma \rangle$, and a parameter ε dictating convergence conditions (a small positive number).
OUTPUT: A policy π.

$V(s) \leftarrow 0$, for all $s \in \mathcal{S}$.
$\Delta \leftarrow 0$
while $\Delta > \varepsilon$ **do**
    for $s \in S$ **do**
        $v \leftarrow V(s)$
        $V(s) \leftarrow \max_a \sum_{s' \in S} \mathcal{T}(s' \mid s, a) \left[\mathcal{R}(s') + \gamma V(s') \right]$
        $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
for $s \in S$ **do**
    $\pi(s) \leftarrow \arg\max_a \sum_{s' \in S} \mathcal{T}(s' \mid s, a) \left[\mathcal{R}(s') + \gamma V(s') \right]$
return π

Value Iteration notably solves for the optimal policy; the algorithm exhaustively explores the entire state-action space and solves for the optimal value function. As a result, Value

Iteration is not applicable for real-time planning. However, it is useful when optimality is extremely important.

2.2.2 Real-Time Dynamic Programming

Real-Time Dynamic Programming (RTDP) [8] is a sampling-based algorithm for solving MDPs that does not require exhaustively exploring all states. RTDP also uses the Bellman Equation to update its value function approximation, but explores the space by subsequent *rollouts* - that is, by repeatedly returning the agent to the start state, sampling actions greedily, and exploring out to a specified depth. From these rollouts RTDP approximates the optimal value function quite well. RTDP is notably quite faster than Value Iteration since it does not explore the entire state space. Instead, it explores until the same convergence criteria as Value Iteration is satisfied, or if the algorithm has exceeded its budgeted number of rollouts. The psuedocode for RTDP is provided in Algorithm 2

Algorithm 2 Real Time Dynamic Programming

INPUT: An MDP instance, $M = \langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \gamma \rangle$, and three parameters:

1. `max-depth`, denoting the maximum rollout depth
2. `numRollouts`, denoting the maximum number of rollouts
3. ε, specifying the convergence criteria.

OUTPUT: A policy π.

$V(s) \leftarrow 0$, for all $s \in \mathcal{S}$.
`rollout` $\leftarrow 0$
$visited \leftarrow queue(\varnothing)$
$\Delta \leftarrow \infty$
while ($\Delta > \varepsilon \wedge$ `rollout` $<$ `numRollouts`) **do**
    $depth \leftarrow 0$
    $visited.Clear()$
    $s \leftarrow M.\mathcal{S}.initialState$
    $V_{prev} \leftarrow V$
    $\Delta \leftarrow 0$
    while ($s \notin G \wedge depth <$ `max-depth`) **do** ▷ Rollout
        $depth \leftarrow depth + 1$
        $visited.Push(s)$
        $V(s) \leftarrow \max_{a \in \mathcal{A}}(Q(s, a))$
        $\Delta \leftarrow \max(\Delta, |V_{prev}(s) - V(s)|)$
        $a \leftarrow \arg\max_a \sum_{s' \in S} \mathcal{T}(s' \mid s, a) [\mathcal{R}(s') + \gamma V(s')]$
        $s \sim \mathcal{T}(s' \mid s, a)$
for $s \in S$ **do**
    $\pi(s) \leftarrow \arg\max_a \sum_{s' \in S} \mathcal{T}(s' \mid s, a) [\mathcal{R}(s') + \gamma V(s')]$
return π

Also of note is that the resulting policy is a *partial policy*, that is, the policy only specifies behavior for a subset of the state space. This is essential for getting planning algorithms

to work in large state spaces. It is also worth noting that RTDP converges to the optimal policy in the limit of trials [8]. In practice we expect a slight tradeoff of optimality for speed, but for simple tasks this tradeoff is negligible. In massive state spaces such as those considered during our experimentation, the tradeoff is significantly more noticeable (i.e. in order to get real-time solutions, optimality is sacrificed).

During experimentation, we also test with Bounded RTDP [36], an extension of RTDP that introduces upper and lower bounds on the value function during exploration, encouraging faster convergence. Notably, BRTDP produces partial policies with strong anytime performance guarantees while only exploring a fraction of the state space. There are known methods for finding reasonable initial lower and upper bounds, but performance is reasonable when bounds are selected naively. The algorithm for BRTDP is extremely similar - the convergence conditions are typically modified to check that the difference between the upper bound and the lower bound is less than some small value, ε. During rollouts, BRTDP updates its lower bound on the value function and its upper bound on the value function based on the minimal Q-value for an action during each rollout and the smallest maximal Q-value during each rollout.

2.3 Object-Oriented Markov Decision Process

An Object-Oriented Markov Decision Process (OO-MDP) [22] efficiently represents the state of an MDP through the use of objects and predicates.

Definition 6. *An* Object-Oriented Markov Decision Process (OO-MDP) *is an eight tuple,* $\langle \mathcal{C}, \textsc{Att}(c), \textsc{Dom}(a), \mathcal{O}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \gamma \rangle$*, where:*

- $\mathcal{C}$ *is a set of object classes.*
- $\textsc{Att}(c)$ *is a function* $\mathcal{C} \mapsto A$ *that specifies the attributes associated with class* $c \in \mathcal{C}$*.*
- $\textsc{Dom}(a_i)$ *is a function* $A \mapsto [x, y]$*, s.t.* $\{n \in \mathbb{N} \mid x \leq n \leq y\}$*, that specifies the space*[1] *of possible values for an attribute* a_i*.*
- $\mathcal{O}$ *is a collection of objects,* $o \in \mathcal{O}$*, where each object belongs to a class,* $\mathcal{C}$*. The* state *of an object* $o.state$ *is a value assignment to all of the attributes of* o*.*
- $\mathcal{S}$ *is a finite set of states, where a state is uniquely identified by* $\bigcup_{o \in \mathcal{O}} o.state$
- $\mathcal{A}$ *is a finite set of actions, also called the* action space*.*
- $\mathcal{T}$ *denotes* $\mathcal{T}(s' \mid s, a)$*, the transition probability of an agent applying action* $a \in \mathcal{A}$ *in state* $s \in \mathcal{S}$ *and arriving in* $s' \in \mathcal{S}$
- $\mathcal{R} : \mathcal{S} \mapsto \mathbb{R}$ *denotes the real valued reward received by the agent for occupying state* s*.*
- $\gamma \in [0, 1)$ *is a discount factor that defines how much the agent prefers immediate rewards over future rewards (the agent prefers to maximize immediate rewards as* γ *decreases).*

[1] The space need not be the natural numbers.

An OO-MDP state is a collection of objects, $O = \{o_1, \ldots, o_o\}$. Each object o_i belongs to a class, $c_j \in \{c_1, \ldots, c_c\}$. Every class has a set of attributes, $Att(c) = \{c.a_1, \ldots, c.a_a\}$, each of which has a domain, $Dom(c.a)$, of possible values.

OO-MDPs enable planners to use predicates over classes of objects. That is, the OO-MDP definition also allows us to create a set of predicates $\mathcal{P}$ that operate on the state of objects to provide high-level information about the underlying state.

The representative power of OO-MDP predicates generalize across specific tasks. As we will see, OO-MDP objects often appear across tasks from the same domain. Since predicates operate on collections of objects, they generalize beyond specific tasks within the domain. For instance, in Minecraft, a predicate checking the contents of the agent's inventory generalizes beyond any particular Minecraft task, so long as an agent object exists in each task.

3 Learning To Plan

We now introduce the conceptual framework that lets us talk precisely about agents that learn to plan.

3.1 Definitions

Decision making agents are often designed to deal with families of related MDPs. Here, we introduce the notion of a *domain*, which specifies precisely what we mean by 'related problems':

Definition 7. *A* Domain*, denoted D, is a five tuple, $\langle \mathcal{C}, \textsc{Att}(c), \mathcal{A}, \mathcal{T}, \textsc{Dom}(a) \rangle$, where each element is defined as in the OO-MDP definition.*

We introduce an intermediary representation relative to the agent, termed *Agent Space*, first introduced by [31].

Definition 8. *Let* Agent Space *refer to a collection of predicates $\mathcal{P}_{agent}$ that relate the agent object to other objects in the domain.*

Since all of our decision making problems are at their core, planning problems, we will be interested in MDPs where an agent is trying to satisfy a specific goal. Specifically, the reward function associated with each problem will be a *goal-directed reward function*:

Definition 9. *A* Goal-Directed Reward Function *is a reward function defined by a characteristic predicate, p. That is, the output of the reward function is one of two values, r_{goal}, or $r_\varnothing$. We say that the predicate p defines the reward function $\mathcal{R}$ relative to r_{goal} and $r_\varnothing$ when:*

$$\mathcal{R}_p(s) = \begin{cases} r_{goal} & p(s) \\ r_\varnothing & \neg p(s) \end{cases} \tag{4}$$

Now we define a *task*, indicating a specific problem instance an agent is supposed to solve:

Definition 10. *A* Task *is a fully specified OO-MDP. A task τ belongs to the domain D just in case $\tau.\mathcal{C} = D.\mathcal{C}, \tau.\mathcal{T} = D.\mathcal{T}, \tau.\textsc{Att} = D.\textsc{Att}, \tau.\mathcal{A} = D.\mathcal{A}$.*

Tasks are specific problem instances. That is, given a goal, and a fully specified world, the agent is tasked with computing a policy that maximizes their expected long term (discounted) reward.

3.2 Task Generators

Next, we introduce the *Task Generator*, which is central to the consideration of transferring knowledge among related problems.

Definition 11. *A* Task Generator *is a randomized polynomial-time turing machine that takes as input a domain D and a set of constraints φ, and outputs a task, $\tau \in D$, such that the constraints specified by φ are satisfied by τ.*

In short, task generators enable us to generate random tasks from a domain, D. Critically, task generators are *randomized* - this ensures that repeated queries to a task generator with the same domain and constraints produce different tasks.

The constraints, φ, consists of two sets:

- $\varphi.A$ is a set of constraints on attribute ranges. Namely, $\varphi.A = \{(a_1, x_1, y_1), \ldots, (a_k, x_k, y_k)\}$, where each triple denotes the range of possible values that attributes $a_1, \ldots, a_k$ may take on.
- $\varphi.P$ is a set of logical constraints. Namely, $\varphi.P = \{p_1, \ldots p_n\}$, where p_i is a formula of first order logic.

The attribute constraints $\varphi.A$ modify the function $\textsc{Dom}(a)$ in the following way:

- $\textsc{Dom}_\theta(a_i)$ is modified such that, $A \mapsto [\varphi.A(a_i).x_i, \varphi.A(a_i).y_i]$, s.t. $\{n \in \mathbb{N} \mid \varphi.A(a_i).x_i \leq n \leq \varphi.A(a_i).y_i\}$, that specifies the space of possible values for an attribute a_i.

The logical constraints φ_P may modify any other aspect of the OO-MDP representing the task τ. For instance, we might imagine a constraint necessitating the existence of an object of a particular class: $\exists_{o \in \mathcal{O}} (class(o) = \texttt{agent})$, indicating that there must exist at least one agent. We could imagine extending these constraints to specify some interesting properties of a given task τ:

- The reward function of τ is goal-directed w.r.t the predicate p.
- There are no more than n objects of class c in τ.
- The only attributes that change across states are $\{a_1, \ldots, a_k\}$.

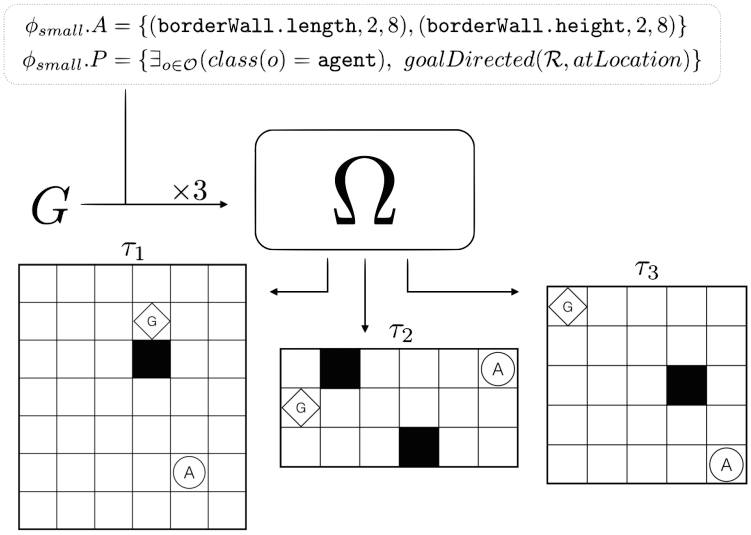


Figure 1: Three example tasks from the Grid World domain generated by Ω.

3.3 Example: Grid World

Consider the Grid World domain, G, defined as follows:

- $\mathcal{C} = \{\texttt{agent}, \texttt{wall}, \texttt{borderWall}\}$
- $\textsc{Att}(c) = \{(\texttt{agent} : x, y), (\texttt{wall} : x, y), (\texttt{borderWall} : length, height)\}$
- $\mathcal{A} = \{\texttt{north}, \texttt{east}, \texttt{south}, \texttt{west}\}$
- $\textsc{Dom}(a) = \{x \in \mathbb{N}, y \in \mathbb{N}, length \in \mathbb{N}, height \in \mathbb{N}\}$
- $\mathcal{T} =$

$$\texttt{north} : \texttt{agent}.y = \begin{cases} \texttt{agent}.y & \texttt{agent}.y + 1 > \texttt{borderWall}.height) \vee \\ & \exists_{o \in \mathcal{O}}(class(o) = \texttt{wall} \wedge o.(x, y) = \texttt{agent}.(x, y + 1)) \\ \texttt{agent}.y + 1 & \text{otherwise} \end{cases}$$

The remaining actions have the same dynamics as `north`, but with the conditions and effects as one would expect (movement in each direction, impeded only by the border or walls).

Note that the grid world domain G contains infinitely many tasks, since $\textsc{Dom}(c)$ allows for an infinite space of attribute assignments. With a set of constraints, φ, the space is constrained, so there are only finitely many tasks. Consider the following set of constraints:

- $\varphi_{small}.A = \{(\texttt{borderWall.length}, 2, 8), (\texttt{borderWall.height}, 2, 8)\}$
- $\varphi_{small}.P = \{\exists_{o \in \mathcal{O}}\left(class(o) = \texttt{agent}\right),\ goalDirected(\mathcal{R}, atLocation)\}$.

Consider a Task Generator Ω. Given φ_{small} and the Grid World domain G, the task generator randomly creates tasks where the grid dimensions are between 2 and 8, and there

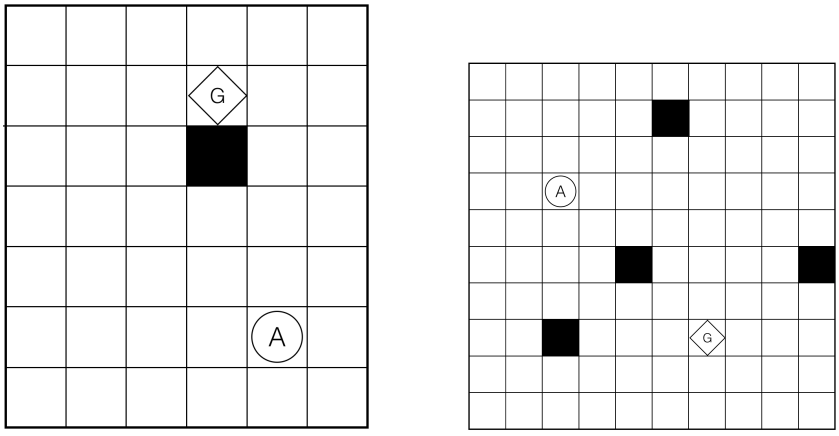
is an agent placed randomly in the grid. Additionally, the reward function is goal-directed with respect to an *atLocation* predicate, meaning that the agent receives positive reward when it is at a particular randomized location in the grid. Figure 1 illustrates the full process. Given a set of constraints, φ_{small}, and a domain G, we make three queries to Ω to generate three tasks, τ_1, τ_2, τ_3.

Now consider a second set of constraints, φ_{large}:

- $\varphi_{large}.A = \{(\texttt{borderWall.length}, 8, 16), (\texttt{borderWall.height}, 8, 16)\}$
- $\varphi_{large}.P = \varphi_{small}.P$

With these larger constraints, we could query the task generator Ω to create larger grid worlds, between 8×8 and 16×16. Note that the logical constraints of φ_{large} are the same as in φ_{small} (i.e. there will be an agent and a goal location). Using these two different sets of constraints, we can generate tasks from the domain G. Example tasks generated from each constraint set are shown in Figure 2.

Using task generators, we investigate families of related planning problems.



(a) A task in the Grid World domain generated with constraints φ_{small}

(b) A task in the Grid World domain generated with constraints φ_{large}

Figure 2: Different randomly generated tasks from the Grid World domain.

3.4 Computational Learning Theory

Alternatively, we may view a Task Generator instance as a probability distribution over the space of tasks defined by the domain D with respect to some distribution over the constraints $\varphi.A$. That is, consider the space of tasks where the constraints $\varphi.P$ are all satisfied. Then $\varphi.A$ defines the space of possible values for a random variable. This consideration allows interesting extensions to the Probably Approximately Correct (PAC) framework [49]. Note that different values of $\varphi.A$ (e.g. φ_{small} vs. φ_{large}) will lead to two different and possibly disjoint distributions over tasks. This problem structure lets us consider variants to the

PAC problem wherein the training distribution is disjoint from the test distribution, but is related in virtue of the tasks belonging to the same domain. [9] performed a theoretical investigation of the PAC framework where the training and test distributions are disjoint. In future work, we are interested in furthering this investigation in the context of agents that learn to plan.

3.5 Agent Space

Consider two tasks τ_1 and τ_2, both belonging to the same domain D. The critical observation is that knowledge acquired by solving τ_1 is useful when solving τ_2. Consider the case that $\tau_1 = \tau_2$. Then clearly knowing the optimal policy for τ_1 (trivially) helps to compute the optimal policy for τ_2.

Now suppose $\tau_1 \neq \tau_2$, but that they share a domain, D. Consider that τ_1 and τ_2 both have Goal-Based Reward Functions operating under predicate p. Furthermore, since $\tau_1, \tau_2 \in D$, we know that the space of classes, $D.\mathcal{C}$, the action space, transition dynamics, and space of attributes are shared between the two tasks.

Thus, the idea is to use our knowledge of what is shared between two tasks belonging to the same domain in order to *translate* between them using an intermediary representation. Since the both τ_1 and τ_2 share object classes, we know that predicates operating on objects are guaranteed to preserve meaning across tasks. Critically, relations among objects will still be relevant across tasks. For instance, predicates that operate on the attributes of the agent will necessarily transfer between τ_1 and τ_2, assuming these tasks were generated under the constraint that an agent object exists in the task. In the future, we are interested in investigating learning representations within this framework, as generality across tasks is an essential characteristic of a good representation.

In this work we learn goal-based action priors from a series of training tasks, sampled from the domain D. These priors are represented in agent space, enabling transfer across tasks from the same domain. These priors are used to prune away actions on a state by state basis for any task belonging to the target domain, reducing the number of state-action pairs the agent needs to evaluate in order to perform optimally. Ultimately, planning algorithms equipped with these priors are capable of solving significantly more difficult problems than algorithms without them. We present these priors inside the framework of an agent learning to plan.

4 Goal-Based Action Priors

To address state-action space explosions in planning tasks, we investigate learning an action prior conditioned on the current state and an abstract goal description. This *goal-based action prior* enables an agent to prune irrelevant actions on a state-by-state basis according to the agent's current goal, focusing the agent on the most promising parts of the state

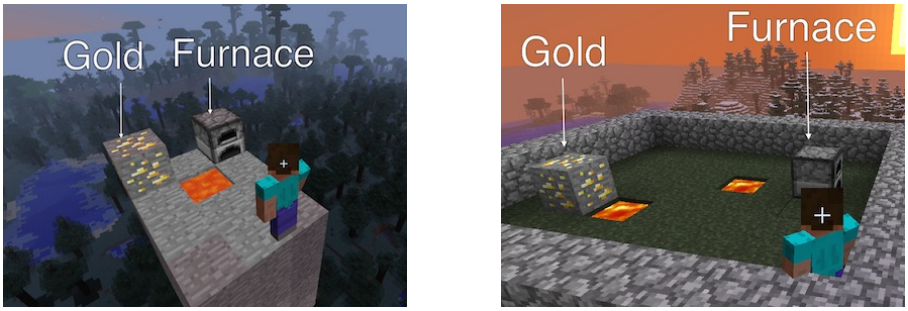


Figure 3: Two problems from the Minecraft domain, where the agent's goal is to collect the gold ore and smelt it in the furnace while avoiding the lava. Our agent is unable to solve the problem on the right before learning because the state-action space is too large. After learning on simple problems like the one on the left, the agent can quickly solve the larger problem.

space. The agent will learn these priors from solving simple task instances and use this knowledge on more complex tasks generated from the same domain.

Goal-based action priors can be specified by hand or learned by repeated queries to a Task Generator, Ω, making them a concise, transferable, and learnable means of representing useful planning knowledge.

Our results demonstrate that these priors provide dramatic improvements for a variety of planning tasks compared to baselines in simulation, and are applicable across different tasks. Moreover, while manually provided priors outperform baselines on difficult problems, our approach is able to learn goal-based action priors on simple, tractable, training problems that yield even greater performance on the test problems than manually provided priors.

We conduct experiments in Minecraft. Figure 3 shows an example of two problems from the Minecraft domain; the agent learns on simple tasks, (like the problem in the left image) and tests on more challenging tasks from the same domain that it has never previously encountered (like the problem in the right image).

4.1 Approach

We define a *goal-based action prior* as knowledge provided to a planning algorithm to help reduce problem complexity. These priors are used to prune actions on a state by state basis, which naturally reduces the number of state-action pairs the agent needs to evaluate. The key observation is that for action-rich domains (i.e. $\mathcal{A}$ is large), many actions are not relevant in every state, but are still relevant at some point in the task. Using goal-based action priors, an agent will be biased toward the most relevant action applications for each state, encouraging the agent to explore the most promising parts of the state space.

These priors are inspired by affordances. Affordances were originally proposed by Gibson as action possibilities prescribed by an agent's capabilities in an environment [24], and have recently received a lot of attention in robotics research [33, 32]. In a recent review on

the theory of affordances, Chemero suggests that an affordance is a relation between the features of an environment and an agent's abilities [19]. It is worth noting that the formalism proposed by Chemero differs from the interpretation of affordance that is common within the robotics community. Our goal-based action priors are analogously interpreted as a grounding of Chemero's interpretation of an affordance, where the features of the environment correspond to the goal-dependent state features, and the agent's abilities correspond to the OO-MDP action set. In earlier versions of this work, we refer to these priors as affordances [7, 1].

4.2 Modeling the Optimal Actions

The goal is to formalize planning knowledge that allows an agent to avoid searching suboptimal actions in each state based on the agent's current goal. This knowledge must be defined in a way that it is applicable across tasks from the same domain (i.e. in agent space).

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

$$\mathcal{A}^* = \{a \mid Q^*_G(s,a) = V^*_G(s)\}, \tag{5}$$

where $Q^*_G(s,a)$ and $V^*_G(s)$ represent the optimal Q function and value function relative to the goal G.

We 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's optimality:

$$\Pr(a_i \in \mathcal{A}^* \mid s, G) \tag{6}$$

for $i \in \{1, \ldots, |\mathcal{A}|\}$, where $\mathcal{A}$ is the OO-MDP action space for the domain.

To generalize across tasks, we abstract the state and goal into a set of n paired predicates and goals, $\{(p_1, g_1) \ldots (p_n, g_n)\}$. We abbreviate each pair (p_j, g_j) to δ_j for simplicity. Each predicate is an agent space predicate, $p \in \mathcal{P}_{agent}$ ensuring transferability between tasks. For example, an agent space predicate might be $nearTrench(agent)$ which is true when the agent is standing next to a trench object. In general these could be arbitrary logical expressions of the state; in our experiments we used unary predicates. G is a *goal* which is a predicate on states that is true if and only if a state is terminal. A goal specifies the sort of problem the agent is trying to solve, such as the agent retrieving an object of a certain type from the environment, reaching a particular location, or creating a new structure. These correspond directly to the predicates that serve as characteristic predicates for Goal-Based Reward Functions. Goals are included in the features since the relevance of each action changes dramatically depending on the agent's current goal.

We rewrite Equation 6:

$$\Pr(a_i \in \mathcal{A}^* \mid s, G) = \Pr(a_i \in \mathcal{A}^* \mid s, G, \delta_1 \ldots \delta_n) \tag{7}$$

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

$$f(\delta, s, G) = \begin{cases} 1 & \delta.p(s) \wedge \delta.g(G) \\ 0 & \text{otherwise} \end{cases} \tag{8}$$

Evaluating f for each δ_j given the current state and goal gives rise to a set of binary features, $\phi_j = f(\delta_j, s, G)$, which we use to reformulate our probability distribution:

$$\Pr(a_i \in \mathcal{A}^* \mid s, G, \delta_1 \ldots \delta_n) = \Pr(a_i \in \mathcal{A}^* \mid \phi_1, \ldots, \phi_n) \tag{9}$$

This equation models how optimal each action is given a state, and goal. Critically, we can rewrite the lefthand side of the equation in terms of $\phi_1, \ldots, \phi_n$, which provides an agent space representation of the current state. This intermediary representation is exactly what enables our agent to transfer these priors from a set of training tasks to arbitrary tasks from the same domain.

This distribution may be modeled in a number of ways, making this approach extremely flexible.

4.2.1 Expert Model

One model that can be specified by an expert is an OR model. In the OR model some subset of the features ($\phi^i \subset \phi$) are assumed to cause action a_i to be optimal; as long as one of the features is on, the probability that a_i is optimal is one. If none of the features are on, then the probability that a_i is optimal is zero. More formally,

$$\Pr(a_i \in \mathcal{A}^* \mid \phi_1, \ldots, \phi_n) = \phi^i_1 \vee \ldots \vee \phi^i_m \tag{10}$$

where m is the number of features that can cause a_i to be optimal ($m = |\phi^i|$).

In practice, we do not expect such a distribution to be reflective of reality; if it were, then no planning would be needed because a full policy would have been specified. However, it does provide a convenient way for a designer to provide conservative background knowledge. Specifically, a designer can consider each precondition-goal pair and specify the actions that could be optimal in that context, ruling out actions that would be known to be irrelevant or dependent on other state features being true.

Because the OR model is not expected to be reflective of reality and because of other limitations (such as not allowing support for an action to be provided when a feature is off), the model is not practical for learning.

Our real goal is to learn from a series of simple tasks, and use this knowledge to solve much more challenging problems from the same domain. Learned priors have the potential to outperform hand-coded priors by more flexibly adapting to the features that predict optimal actions over a large training set. We consider two models for learning: Naive Bayes and Logistic Regression.

4.2.2 Naive Bayes

We first factor Equation 9 using Bayes' rule, introducing a parameter vector θ_i of feature weights:

$$\Pr(a_i \in \mathcal{A}^* \mid \phi_1, \ldots, \phi_n) = \frac{\Pr(\phi_1, \ldots, \phi_n, \mid a_i \in \mathcal{A}^*, \theta_i) \Pr(a_i \in \mathcal{A}^* \mid \theta_i)}{\Pr(\phi_1, \ldots, \phi_n \mid \theta_i)} \tag{11}$$

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

$$= \frac{\prod_{j=1}^{n} \Pr(\phi_j \mid a_i \in \mathcal{A}^*, \theta_i) \Pr(a_i \in \mathcal{A}^* \mid \theta_i)}{\Pr(\phi_1, \ldots, \phi_n \mid \theta_i)} \tag{12}$$

Finally, we define the prior on the optimality of each action to be the fraction of the time each action was optimal during training.

4.2.3 Logistic Regression

Under the Logistic Regression model, classification is computed by a logistic threshold function. That is, for each action, we learn a vector of weights $\vec{w}$ that determines the optimal decision boundary:

$$\textsc{LogReg}_{a_i}(s, G) = \frac{1}{1 + e^{-\vec{w}_{a_i} \cdot \vec{\phi}(s,G)}} \tag{13}$$

Where $\phi(s, G)$ denotes the agent space features extracted from the state s introduced above. Rewriting in terms of our feature vector, $\vec{\phi}$:

$$\textsc{LogReg}_{a_i}(\vec{\phi}) = \frac{1}{1 + e^{-\vec{w}_{a_i} \cdot \vec{\phi}}} \tag{14}$$

Then, our decision rule for classification is simply:

$$\begin{cases} 1 & \textsc{LogReg}_{a_i}(\vec{\phi}) \geq 0.5 \\ 0 & \text{otherwise} \end{cases} \tag{15}$$

4.3 Learning the Optimal Actions

For learning we consider a Task Generator Ω, a domain D, and a set of constraints, φ_{train}. These constraints will force the tasks output by Ω to be sufficiently small (i.e. small enough so that tabular approaches like Value Iteration can solve for the optimal policy).

We generate n training tasks, $\tau_1, \ldots, \tau_n$, and solve for the optimal policy in each task, $\pi_1, \ldots, \pi_n$.

Using the optimal policies across these tasks, we can learn the model parameters for the Naive Bayes model, or inform the weights for the Logistic Regression model.

To compute model parameters using Naive Bayes, we compute the maximum likelihood estimate of the parameter vector θ_i for each action using the optimal policies for the training tasks.

Under our Bernouli Naive Bayes model, we estimate the parameters $\theta_{i,0} = \Pr(a_i)$ and $\theta_{i,j} = \Pr(\phi_j | a_i)$, for $j \in \{1, \ldots, n\}$, where the maximum likelihood estimates are:

$$\theta_{i,0} = \frac{C(a_i)}{C(a_i) + C(\overline{a_i})} \tag{16}$$

$$\theta_{i,j} = \frac{C(\phi_j, a_i)}{C(a_i)} \tag{17}$$

Here, $C(a_i)$ is the number of observed occurrences where a_i was optimal across all worlds W, $C(\overline{a_i})$ 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 for each training world, π_w. More formally:

$$C(a_i) = \sum_{w \in W} \sum_{s \in w} (a_i \in \pi_w(s)) \tag{18}$$

$$C(\overline{a_i}) = \sum_{w \in W} \sum_{s \in w} (a_i \notin \pi_w(s)) \tag{19}$$

$$C(\phi_j, a_i) = \sum_{w \in W} \sum_{s \in w} (a_i \in \pi_w(s) \wedge \phi_j == 1) \tag{20}$$

To compute the optimal decision boundary for Logistic Regression, we compute gradient descent using the L_2 loss function, resulting in the following update rule:

$$w_{ij} \leftarrow w_{ij} + \alpha(y_a - \textsc{LogReg}_{a_i}(\vec{\phi})) \times \textsc{LogReg}_{a_i}(\vec{\phi})(1 - \textsc{LogReg}_{a_i}(\vec{\phi})) \tag{21}$$

Where $y_a = 1$ if action a was optimal in state s under goal G represented by the feature vector $\vec{\phi}$, and $y_a = 0$ otherwise. We optimize until convergence.

During the learning phase, the agent learns when actions are useful with respect to the agent space features. For example, consider the three different problems shown in Figure 4. During training, we observe that the `destroy` action is often optimal when the agent is looking at a block of gold ore and the agent is trying to smelt gold bars. Likewise, when the agent is not looking at a block of gold ore in the smelting task we observe that the `destroy` action is generally not optimal (i.e. destroying grass blocks is typically irrelevant to smelting). This information informs the distribution over the optimality of the `destroy` action, which is used at test time to encourage the agent to destroy blocks when trying to smelt gold and looking at gold ore, but not in other situations (unless the prior suggests using `destroy`).

At test time, we query Ω with a different set of constraints, φ_{test} but the same domain D. Notably, φ_{test} will necessitate larger, more complicated tasks than those trained on. For simplicity, our learning process uses a strict separation between training and test; after learning is complete our model parameters/weights remain fixed.

(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 4: Three different problems from the Minecraft domain.

4.4 Action Pruning with Goal-Based Action Priors

A planner using a goal-based action prior will prune actions on a state-by-state basis.

Under the expert specified OR model, when $\Pr(a_i \in \mathcal{A}^* \mid \vec{\phi}) = 0$ action a_i is pruned from the planner's consideration. When $\Pr(a_i \in \mathcal{A}^* \mid \vec{\phi}) = 1$, action a_i remains in the action set to be searched by the planner.

Under Naive Bayes, we found that the optimal decision rule resulted in poor performance as a consequence of pruning away too many actions. Instead, we imposed a more conservative threshold, only pruning away actions if they were extremely unlikely to be sub-optimal.

Under Logistic Regression, we used the optimal decision rule to determine which actions were to be considered in each state. We bias each distribution by normalizing each distribution's weight with respect to the maximally likely action. Specifically:

$$\Pr(a_i \in \mathcal{A}^* \mid \vec{\phi}) = \frac{\textsc{LogReg}_{a_i}(\vec{\phi})}{\max_j \textsc{LogReg}_{a_j}(\vec{\phi})} \tag{22}$$

This ensures that the maximally likely action will *always* be selected.

5 Related Work

In this section, we discuss the differences between goal-based action priors and other forms of knowledge engineering that have been used to accelerate planning, as well as other models and frameworks that resemble the notions introduced in this document.

5.1 Stochastic Approaches

Temporally extended actions are actions that the agent can select like any other action of the domain, except executing them results in multiple primitive actions being executed in succession. Two common forms of temporally extended actions are *macro-actions* [26]

and *options* [47]. Macro-actions are actions that always execute the same sequence of primitive actions. Options are defined with high-level policies that accomplish specific sub tasks. For instance, when an agent is near a door, the agent can engage the 'door-opening-option-policy', which switches from the standard high-level planner to running a policy that is crafted to open doors. Although the classic options framework is not generalizable to different state spaces, creating *portable* options is a topic of active research [30, 28, 43, 3, 29].

Since temporally extended actions may negatively impact planning time [27] by adding to the number of actions the agent can choose from in a given state, combining our priors with temporally extended actions allows for even further speedups in planning, as demonstrated in Table 3. In other words, goal-based action priors are complementary knowledge to options and macro-actions.

Sherstov and Stone [46] considered MDPs for which the action set of the optimal policy of a source task could be transferred to a new, but similar, target task to reduce the learning time required to find the optimal policy in the target task. Goal-based action priors prune away actions on a state-by-state basis, enabling more aggressive pruning whereas the learned action pruning is on a per-task level.

Rosman and Ramamoorthy [45] provide a method for learning action priors over a set of related tasks. Specifically, they compute a Dirichlet distribution over actions by extracting the frequency that each action was optimal in each state for each previously solved task. These action priors can only be used with planning/learning algorithms that work well with an ϵ-greedy rollout policy, while our goal-based action priors can be applied to almost any MDP solver. Their action priors are only active for a fraction ϵ of the time, which is quite small, limiting the improvement they can make to the planning speed. Finally, as variance in tasks explored increases, the priors will become more uniform. In contrast, goal-based action priors can handle a wide variety of tasks in a single prior, as demonstrated by Table 1.

Heuristics in MDPs are used to convey information about the value of a given state-action pair with respect to the task being solved and typically take the form of either value function initialization [25], or reward shaping [40]. However, heuristics are highly dependent on the reward function and state space of the task being solved, whereas goal-based action priors are state space independent and may be learned easily for different reward functions. If a heuristic can be provided, the combination of heuristics and our priors may even more greatly accelerate planning algorithms than either approach alone.

5.2 Deterministic Approaches

There have been several attempts at engineering knowledge to decrease planning time for deterministic planners. These are fundamentally solving a different problem from what we are interested in since they deal with non-stochastic problems, but there are interesting parallels nonetheless.

Hierarchical Task Networks (HTNs) employ *task decompositions* to aid in planning [23]. The agent decomposes the goal into smaller tasks which are in turn decomposed into smaller

tasks. This decomposition continues until immediately achievable primitive tasks are derived. The current state of the task decomposition, in turn, informs constraints which reduce the space over which the planner searches. At a high level HTNs and goal-based action priors both achieve action pruning by exploiting some form of supplied knowledge. We speculate that the additional action pruning provided by our approach is complementary to the pruning offered by HTNs.

One significant difference between HTNs and our planning system is that HTNs do not incorporate reward into their planning. Additionally, the degree of supplied knowledge in HTNs far exceeds that of our priors: HTNs require not only constraints for sub-tasks but a hierarchical framework of arbitrary complexity. Goal-based action priors require a domain specification, a task generator, and arbitrarily many sets of constraints, each of which is arguably necessary for planning across related tasks, regardless of what knowledge is being learned.

An extension to the HTN is the probabilistic Hierarchical Task Network (pHTN) [34]. In pHTNs, the underlying physics of the primitive actions are deterministic. The goal of pHTN planning is to find a sequence of deterministic primitive actions that satisfy the task, with the addition of matching user preferences for plans, which are expressed as probabilities for using different HTN methods. As a consequence, the probabilities in pHTNs are in regard to probabilistic search rather than planning in stochastic domains, as we do.

Bacchus and Kabanza [4, 5] provided planners with domain dependent knowledge in the form of a first-order version of linear temporal logic (LTL), which they used for control of a forward-chaining planner. With this methodology, a STRIPS style planner may be guided through the search space by pruning candidate plans that falsify the given knowledge base of LTL formulas, often achieving polynomial time planning in exponential space. LTL formulas are difficult to learn, placing dependence on an expert, while we demonstrate that our priors can be learned from experience. Our approach is related to preferred actions used by LAMA [44] in that our agent learns actions which are useful for a specific problem and expands those actions first. However our approach differs in that it generalizes this knowledge across different planning problems, so that the preferred actions in one problem influence search in subsequent problems in the domain.

5.3 Models

Our planning approach relies critically on the the ability of the OO-MDP to express properties of objects in a state, which is shared by other models such as First-Order MDPs (FOMDPs) [13]. As a consequence, a domain that can be well expressed by a FOMDP may also benefit from our planning approach. However FOMDPs are purely symbolic, while OO-MDPs can represent states with objects defined by numeric, relational, categorical, and string attributes. Moreover, OO-MDPs enable predicates to be defined that are evaluative of the state rather than attributes that define the state, which makes it easy to add high-level information without adding complexity to the state definition and transition dynamics to account for them.

5.4 Frameworks

The conceptual framework developed by Konidaris serves as inspiration for the underlying notation and structures of task generators, domains, and tasks introduced here [31]. The critical difference is that Konidaris is interested in the reinforcement learning problem as opposed to planning. Later iterations of Konidaris' work focused on skill acquisition and behavior transfer [30]; while skill acquisition and transfer is critical, as discussed, adding high level actions increases the branching factor, consequently making planning more difficult. Our results indicate that goal-based action priors are actually complementary knowledge to temporally extended actions.

Brunskill and Li recently investigated transfer learning for lifelong reinforcement learning [16, 15]. They provide sample complexity guarantees about lifelong RL agents under the assumption that these agents are restricted to solving MDPs that share a state space, which is much more restricted than the domain-level abstraction presented here. In future work, we are interested in furthering this investigation within a broader conceptual framework that allows for more variation between tasks.

6 Evaluation

We evaluate our approach using the game Minecraft. Minecraft is a voxel-based simulation in which the user-controlled agent can place, craft, and destroy blocks of different types. Minecraft's physics and action space are extremely expressive and allow users to create complex objects and systems, including logic gates and functional scientific graphing calculators. Minecraft serves as a model for complicated real world systems such as robots traversing complex terrain, and large scale construction projects involving highly malleable environments. As in these tasks, the agent operates in a very large state-action space in an uncertain environment. Figure 4 shows three example scenes from Minecraft problems that we solve.

6.1 Experiments

Our experiments consist of five common tasks in Minecraft: bridge construction, gold smelting, tunneling through walls, digging to find an object, and path planning.

The training set consists of 20 randomly generated tasks for each goal, for a total of 100 instances. Each instance is guaranteed to be extremely simple: 1,000-10,000 states (small enough to solve with tabular approaches). We specified a set of constraints φ_{train} that ensured that the generated tasks would be small. The output of our training process is the model parameter θ or the weight vector $\vec{w}$, which inform our goal-based action prior depending on which model we are using. 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.

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)
NBP-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)
NBP-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)
NBP-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)
NBP-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)
NBP-RTDP	**19090** (±9158)	**-7.8** (±1)	**246s** (±159)

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

The test set consists of 20 randomly generated tasks from the same domain, with the same five goals, 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). We specified a set of constraints φ_{test} that ensured that the generated tests would be massive.

We fix the number of features at the start of training based on the number predicates defined by the OO-MDP, $|\mathcal{P}|$, and the number of goals, $|G|$. We provide our system with a set of 51 features that are likely to aid in predicting the correct action across instances.

We compare RTDP with priors learned under Naive Bayes used with RTDP (NBP-RTDP), and expert priors RTDP (EP-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. 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. The goal specifies terminal states, and the discount factor is $\gamma = 0.99$. 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.

We report the number of Bellman updates executed by each planning algorithm, 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, NBP-

RTDP and EP-RTDP after planning in 20 different tasks of each goal (100 total). Figure 5 shows the results averaged across all tasks. We report CPU time for completeness, but our results were run on a networked cluster where each node had differing computer and memory resources. As a result, the CPU results have some variance not consistent with the number of Bellman updates in Table 1. Despite this noise, overall the average CPU time shows statistically significant improvement overall with our priors, as shown in Figure 5. Furthermore, we reevaluate each predicate every time the agent visits a state, which could be optimized by caching predicate evaluations, further reducing the CPU time taken for EP-RTDP and NBP-RTDP.

6.2 Results

Because the planners terminate after a maximum of 1000 rollouts, they do not always converge to the optimal policy. NBP-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. Expert-provided priors can add significant value in making large state spaces more tractable, though as we hypothesized, learned priors generally outperform expert provided priors, reinforcing that creating general planning knowledge by hand is rather difficult.

For some task types, NBP-RTDP finds a slightly worse plan on average than RTDP (*e.g.* the mining task). This worse convergence is due to the fact that NBP-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 NBP-RTDP also evaluate several OO-MDP predicates in every state, adding a small amount of time to planning.

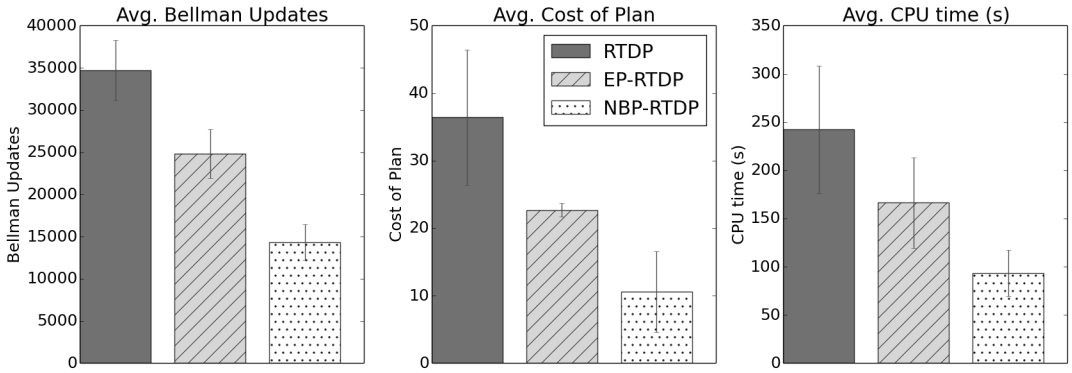


Figure 5: Average results from all tasks.

Planner	Bellman	Reward	CPU
BRTDP	35180.1 (±2223.5)	-81.8 (±18.1)	474.1 (±32.3)
LRP-BRTDP	**11389.4** (±1758.94)	**-10.7** (±0.9)	**78.1** (±13.0)

Table 2: Logistic Regression Priors vs. Bounded-RTDP

6.2.1 Logistic Regression Results

We followed this set of experiments by comparing Bounded RTDP (BRTDP) with and without priors learned with the Logistic Regression model (LRP-BRTDP). In these experiments, we provided a smaller feature set (only 6 features were used), and only tested on tasks of a single goal type. These experiments were conducted as a proof of concept to verify that Logistic Regression can effectively learn useful priors, too. All relevant parameters were set as in the previous set of experiments.

We evaluated on significantly larger tasks than with Naive Bayes. The training tasks are each of a similar size - roughly 1,000-10,000 states. The test tasks are all larger than 1,000,000 states, with several approaching 10,000,000. Each task was provided a goal-directed reward function, defined by the predicate that evaluates whether an agent is at a particular coordinate of the world. In other words, these tasks were large obstacle courses. Lava was scattered randomly throughout the world, and the agent was given blocks to plug up the lava if it desired (one could imagine cases where this is optimal behavior). We conducted tests on 100 randomly generated tasks.

The results are summarized in Table 2. Clearly, the priors improve planning dramatically. Since BRTDP is also designed to only explore a fraction of the state space and is said to have strong anytime performance guarantees, it is significant that BRTDP with goal-based action priors performs far better than without. The results show that in these much larger tasks, BRTDP rarely computes an optimal policy (again, we cut off all planners after 1000 rollouts). If BRTDP were given more rollouts, it would eventually compute a better policy, but at the cost of more planning time. The policy computed by LRP-BRTDP produced an average of cost 17.1, compared to BRTDP's 81.8. Additionally, the policies were computed in dramatically less time (roughly 1 minute to 8 minutes, and 11,000 Bellman updates to 35,000).

6.2.2 Temporally Extended Actions and Goal-Based Action Priors

The primary defect with including temporally extended actions in the action space is that the branching factor increases, consequently increasing planning time. With the use of goal-based action priors, the agent will learn to prune away irrelevant action applications, including options and macro-actions. As a result, goal-based action priors and temporally extended actions are quite complementary.

We conduct experiments with the same configurations as our earlier Minecraft experiments. Domain experts provide useful option policies (e.g. walk forward until hitting a wall, dig

Planner	Bellman	Reward	CPU
RTDP	27439 (±2348)	-22.6 (±9)	107 (±33)
NBP-RTDP	**9935** (±1031)	**-12.4** (±1)	**53** (±5)
RTDP+Opt	26663 (±2298)	-17.4 (±4)	129(±35)
NBP-RTDP+Opt	**9675** (±953)	**-11.5** (±1)	**93** (±10)
RTDP+MA	31083 (±2468)	-21.7 (±5)	336 (±28)
NBP-RTDP+MA	**9854** (±1034)	**-11.7** (±1)	**162** (±17)

Table 3: Priors with Temporally Extended Actions

until looking at gold ore) and macro-actions (e.g. move forward twice, turn around). Priors are learned from 100 training tasks generated by a task generator for the Minecraft domain.

Table 3 indicates the results of comparing RTDP equipped with macro-actions, options, and goal-based action priors across 100 different tasks in the same domain. The results are averaged across goals of each type presented in Table 1. Both macro-actions and options add a significant amount of time to planning due to the fact that the options and macro-actions are being reused in multiple OO-MDPs that each require recomputing the resulting transition dynamics and expected cumulative reward when applying each option/macro-action (a cost that is typically amortized in classic options work where the same OO-MDP state space and transition dynamics are used). This computational cost might be reduced when using a Monte Carlo planning algorithm that does not need the full transition dynamics and expected cumulative reward. Furthermore, the branching factor of the state-action space significantly increases with additional actions, causing the planner to run for longer and perform more Bellman updates. Despite these extra costs in planning time, earned reward with options was higher than without, demonstrating that our expert-provided options add value to the system.

With goal-based action priors, the planner finds a better plan in less CPU time, and with fewer Bellman updates. These results support the claim that priors can handle the augmented action space provided by temporally extended actions by pruning away unnecessary actions, and that options and goal-based action priors provide complementary information.

7 Conclusion

We propose a framework where decision making agents learn to plan by acquiring useful domain knowledge about how to solve families of related problems from a small training set of tasks, eliminating the need for hand engineering knowledge. The critical insight is that problems that are too complex to solve efficiently often resemble much simpler problems for which optimal solutions may be computed. By extracting relevant characteristics of the simple problems' solutions, we introduce strategies for solving the more complex problems by learning about the structure of optimal behavior in the training tasks.

Specifically, we introduce *goal-based action priors* [2], that guide planners according to which

actions are likely to be useful under different conditions. The priors are informed during a training stage in which simple, tractable tasks are solved, and whose solutions inform the planner about optimal behavior in much more complex tasks from the same domain. We demonstrate that goal-based action priors dramatically reduce the time taken to find a near-optimal plan compared to baselines, and suggest that *learning to plan* is a compelling means of scaling planning algorithms to solve families of complex tasks without the need for hand engineered knowledge.

In the future, we hope to automatically discover useful state space specific subgoals online—a topic of some active research [35, 21]. Automatic discovery of subgoals would allow goal-based action priors to take advantage of the task-oriented nature of our priors, and would further reduce the size of the explored state-action space by improving the effectiveness of action pruning. Additionally, we hope to investigate ties between learning to plan and computational learning theory. In particular, we are interested in extending the inductive bias learning framework [9] to learning to plan. Lastly, we are interested in further analysis of the learning to plan framework established here, with a special interest in representation learning in the context of scaffolding. We hypothesize that learning hierarchical object-oriented representations is a natural abstraction for planning and reinforcement learning agents to make, and will significantly reduce problem complexity for a variety of interesting domains.

References

[1] David Abel, Gabriel Barth-Maron, James MacGlashan, and Stefanie Tellex. Toward affordance-aware planning. In *First Workshop on Affordances: Affordances in Vision for Cognitive Robotics*, 2014.

[2] David Abel, David Ellis Hershkowitz, Gabriel Barth-Maron, Stephen Brawner, Kevin O'Farrell, James MacGlashan, and Stefanie Tellex. Goal-based action priors. In *Twenty-Fifth International Conference on Automated Planning and Scheduling*, 2015.

[3] 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.

[4] Fahiem Bacchus and Froduald Kabanza. Using temporal logic to control search in a forward chaining planner. In *In Proceedings of the 3rd European Workshop on Planning*, pages 141–153. Press, 1995.

[5] Fahiem Bacchus and Froduald Kabanza. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence*, 116:2000, 1999.

[6] Paul G Backes, Gregg Rabideau, Kam S Tso, and Steve Chien. Automated planning and scheduling for planetary rover distributed operations. In *Robotics and Automation, 1999. Proceedings. 1999 IEEE International Conference on*, volume 2, pages 984–991. IEEE, 1999.

[7] Gabriel Barth-Maron, David Abel, James MacGlashan, and Stefanie Tellex. Affordances as transferable knowledge for planning agents. In *2014 AAAI Fall Symposium Series*, 2014.

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

[9] Jonathan Baxter. A model of inductive bias learning. *J. Artif. Intell. Res.(JAIR)*, 12:149–198, 2000.

[10] R. Bellman and R.E. Bellman. *Adaptive Control Processes: A Guided Tour*. 'Rand Corporation. Research studies. Princeton University Press, 1961.

[11] Richard Bellman. Dynamic programming, 1957.

[12] Adi Botea, Markus Enzenberger, Martin Müller, and Jonathan Schaeffer. Macro-ff: Improving ai planning with automatically learned macro-operators. *Journal of Artificial Intelligence Research*, 24:581–621, 2005.

[13] Craig Boutilier, Raymond Reiter, and Bob Price. Symbolic dynamic programming for first-order mdps. In *IJCAI*, volume 1, pages 690–697, 2001.

[14] John L Bresina, Ari K Jónsson, Paul H Morris, and Kanna Rajan. Activity planning for the mars exploration rovers. In *ICAPS*, pages 40–49, 2005.

[15] Emma Brunskill and Lihong Li. Sample complexity of multi-task reinforcement learning. *arXiv preprint arXiv:1309.6821*, 2013.

[16] Emma Brunskill and Lihong Li. Pac-inspired option discovery in lifelong reinforcement learning. In *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, pages 316–324, 2014.

[17] F Ceccanti, E Dini, X De Kestelier, V Colla, and L Pambaguian. 3d printing technology for a moon outpost exploiting lunar soil. In *61st International Astronautical Congress, Prague, CZ, IAC-10-D3*, volume 3, 2010.

[18] Giovanni Cesaretti, Enrico Dini, Xavier De Kestelier, Valentina Colla, and Laurent Pambaguian. Building components for an outpost on the lunar soil by means of a novel 3d printing technology. *Acta Astronautica*, 93(0):430 – 450, 2014.

[19] Anthony Chemero. An outline of a theory of affordances. *Ecological psychology*, 15(2):181–195, 2003.

[20] Steve Chien, G Rabideau, R Knight, Robert Sherwood, Barbara Engelhardt, Darren Mutz, T Estlin, Benjamin Smith, F Fisher, T Barrett, et al. Aspen–automated planning and scheduling for space mission operations. In *Space Ops*, pages 1–10, 2000.

[21] 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.

[22] C. Diuk, A. Cohen, and M.L. Littman. An object-oriented representation for efficient reinforcement learning. In *Proceedings of the 25th international conference on Machine learning*, ICML '08, 2008.

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

[24] JJ Gibson. The concept of affordances. *Perceiving, acting, and knowing*, pages 67–82, 1977.

[25] Eric A Hansen and Shlomo Zilberstein. Solving markov decision problems using heuristic search. In *Proceedings of AAAI Spring Symposium on Search Techniques from Problem Solving under Uncertainty and Incomplete Information*, 1999.

[26] 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.

[27] 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.

[28] 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.

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

[30] 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.

[31] George D Konidaris. A framework for transfer in reinforcement learning. In *ICML-06 Workshop on Structural Knowledge Transfer for Machine Learning*, 2006.

[32] Hema S. Koppula, Rudhir Gupta, and Ashutosh Saxena. Learning human activities and object affordances from rgb-d videos. *International Journal of Robotics Research*, 2013.

[33] Hema S. Koppula and Ashutosh Saxena. Anticipating human activities using object affordances for reactive robotic response. In *Robotics: Science and Systems (RSS)*, 2013.

[34] Nan Li, William Cushing, Subbarao Kambhampati, and Sungwook Yoon. Learning probabilistic hierarchical task networks to capture user preferences. *arXiv preprint arXiv:1006.0274*, 2010.

[35] 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.

[36] H Brendan McMahan, Maxim Likhachev, and Geoffrey J Gordon. Bounded real-time dynamic programming: Rtdp with monotone upper bounds and performance guarantees. In *Proceedings of the 22nd international conference on Machine learning*, pages 569–576. ACM, 2005.

[37] Michael Montemerlo, Jan Becker, Suhrid Bhat, Hendrik Dahlkamp, Dmitri Dolgov, Scott Ettinger, Dirk Haehnel, Tim Hilden, Gabe Hoffmann, Burkhard Huhnke, et al. Junior: The stanford entry in the urban challenge. *Journal of field Robotics*, 25(9):569–597, 2008.

[38] Dana Nau, Yue Cao, Amnon Lotem, and Hector Munoz-Avila. Shop: Simple hierarchical ordered planner. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'99, pages 968–973, 1999.

[39] M Newton, John Levine, and Maria Fox. Genetically evolved macro-actions in ai planning problems. *Proceedings of the 24th UK Planning and Scheduling SIG*, pages 163–172, 2005.

[40] Andrew Y Ng, Daishi Harada, and Stuart Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *ICML*, volume 99, pages 278–287, 1999.

[41] Christos H Papadimitriou and John N Tsitsiklis. The complexity of markov decision processes. *Mathematics of operations research*, 12(3):441–450, 1987.

[42] Hugh Philip Possingham. State-dependent decision analysis for conservation biology. In *The Ecological Basis of Conservation*, pages 298–304. Springer, 1997.

[43] 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.

[44] Silvia Richter and Matthias Westphal. The lama planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research*, 39(1):127–177, 2010.

[45] 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.

[46] A.A. Sherstov and P. Stone. Improving action selection in mdp's via knowledge transfer. In *Proceedings of the 20th national conference on Artificial Intelligence*, pages 1024–1029. AAAI Press, 2005.

[47] 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.

[48] Sebastian Thrun, Mike Montemerlo, Hendrik Dahlkamp, David Stavens, Andrei Aron, James Diebel, Philip Fong, John Gale, Morgan Halpenny, Gabriel Hoffmann, et al. Stanley: The robot that won the darpa grand challenge. *Journal of field Robotics*, 23(9):661–692, 2006.

[49] Leslie G Valiant. A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142, 1984.

[50] David Wood, Jerome S Bruner, and Gail Ross. The role of tutoring in problem solving*. *Journal of child psychology and psychiatry*, 17(2):89–100, 1976.