Goal-Based Action Priors

Paper ID: 85

Abstract

Robots that interact with people must flexibly respond to their requests by planning complex actions in very large state spaces, where the reward function may not be known in advance. However, by providing the robot a large range of capabilities that are necessary for solving a variety of problems, planning problems become prohibitively difficult because of the exponential number of ways the robot can affect the world. In this work, we develop a framework for goal and state dependent action priors that can be used to prune away irrelevant actions based on the robot's current goal, thereby greatly accelerating planning in a variety of complex stochastic environments. Because these action priors depend on the current state and the goal, we refer to them as *affordances*. Our framework allows affordances to be easily specified by an expert or to be learned from prior experience in related problems. Due to the large complexity of the space, we use the video game Minecraft as a robot simulator in which an agent learns useful affordances from a set of training worlds and then uses those affordances to successfully find solutions to otherwise intractable problems without them. We also use expert-provided affordances to accelerate planning in a robot cooking domain that is executed on a two-handed manipulator robot.

Introduction

Robots operating in unstructured, stochastic environments such as a factory floor or a kitchen face a difficult planning problem due to the large state space and the very large set of possible tasks [5, 15]. A powerful and flexible robot such as a mobile manipulator in the home has a very large set of possible actions, any of which may be relevant depending on the current goal (for example, robots assembling furniture [15] or baking cookies [5].) When a robot is manipulating objects in an environment, an object can be placed anywhere in a large set of locations. The size of the state space increases exponentially with the number of objects, which bounds the placement problems that the robot is able to expediently solve. Depending on the reward function (which is unknown before runtime), any of these states and actions may be relevant to the solution, but for any specific reward function, most of them are irrelevant. For instance, when making brownies, the oven and flour are important, while the soy sauce and sauté pan are not. For a different task, such as stir-frying broccoli, the robot must use a different set of objects and actions.

Figure 1: Two different problems from the same domain, where the agent's goal is to smelt the gold 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 (since it can place the gold block anywhere). After learning, it can quickly solve the larger problem.

Robotic planning tasks are often formalized as a stochastic sequential decision making problem, modeled as a Markov Decision Process (MDP) [29]. In these problems, the agent must find a mapping from states to actions for some subset of the state space that enables the agent to achieve a goal while minimizing costs along the way. However, many robotics problems correspond to a family of related MDPs; following STRIPs terminology, these problems come from the same domain, but each task may have a different reward function or goal condition. For example, figure 1 shows an example of two problems from the same domain in the game Minecraft [20].

To confront this state-action space explosion, prior work has explored adding knowledge to the planner, such as options [28] and macro-actions [6, 22]. However, while these methods can 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 [14]. Deterministic forward-search algorithms like hierarchical task networks (HTNs) [21], and tem-

poral logical planning (TLPlan) [2, 3], add knowledge to the planner that greatly increases planning speed, but do not generalize to stochastic domains. Additionally, the knowledge provided to the planner by these methods is quite extensive, reducing the agent's autonomy and must be manually supplied by the designer.

To address these issues, we augment an Object Oriented Markov Decision Process (OO-MDP) with a specific type of action prior conditioned on the current state and an abstract goal description. This goal-based action prior enables the robot to prune irrelevant actions on a state-by-state basis based on the agent's current goal and focus on the most promising parts of the state space. Because we condition on both the state and goal description, we refer to this type of action prior as a knowlege base of *affordances*. Affordances were originally proposed by Gibson [11] as action possibilities prescribed by an agent's capabilities in an environment. **stefie10: Talk about Chemmaro here.** Our goal-based action priors can be specified by hand or alternatively learned through experience in related problems, making them a concise, transferable, and learnable means of representing useful planning knowledge. Our experiments demonstrate that affordances provide dramatic improvements for a variety of planning tasks compared to baselines in simulation, and are applicable across different state spaces. Moreover, while manually provided affordances outperform baselines, affordances learned through experience yield even greater improvements. We conduct experiments in the game Minecraft, which has a very large state-action space, and on a real-world robotic cooking assistant. Figure 1 shows an example of two problems from the same domain in the game Minecraft; the agent learns on randomly generated problems and tests on new problems from the same domain that it has never previously encountered. All associated code with this paper may be found at `http://h2r.cs.brown.edu/affordances`. **stefie10: Make this link real.**

Technical Approach

We define affordances as formal knowledge added to a family of related Markov Decision Processes (MDP) from the same domain. An MDP is a five-tuple: $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \gamma \rangle$, where $\mathcal{S}$ is a state space; $\mathcal{A}$ is the agent's set of actions; $\mathcal{T}$ 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 $s' \in \mathcal{S}$; $\mathcal{R}(s, a, s')$ denotes the reward received by the agent for applying action a in state s and transitioning to state s'; and $\gamma \in [0, 1)$ 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).

Our representation of affordances builds on Object-Oriented MDPs (OO-MDPs) [9]. An OO-MDP efficiently represents the state of an MDP through the use of objects and predicates. 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

- Object Classes:
  - Agent
    - Location (x, y, z)
    - Inventory
  - Block
    - Type (Wood, Lava, ...)
    - Location (x, y, z)
    - Destructible (True, False)
- Actions: DestroyBlock, UseBlock, TurnLeft, TurnRight, MoveForward, LookUp, LookDown, PlaceBlock, Jump
- Transition Dynamics: Movement actions (Look, Turn, Move) incorrectly apply a different move action 5% of the time.

(a) Domain

Robot is standing at (x, y, z).
Robot has smelted gold.
Robot has acquired ore.
Robot has constructed a tower of height K.
Robot has built a cube with edge length K.
Robot has destroyed K blocks.

(b) Example goals.

Figure 2: Part of the OO-MDP Domain for the Minecraft.

to use predicates over classes of objects. That is, the OO-MDP definition also includes a set of predicates $\mathcal{P}$ that operate on the state of objects to provide additional high-level information about the MDP state.

OO-MDP predicates provide state space independence. For a given planning domain, OO-MDP objects often appear across tasks. Since predicates operate on collections of objects, they generalize beyond specific state spaces within the domain. For instance, in Minecraft, a predicate checking the contents of the agent's inventory generalizes beyond any particular Minecraft task. We capitalize on this state space independence by using OO-MDP predicates as features for action pruning. Figure 2 shows part of the definition of our OO-MDP for the Minecraft domain.

Following STRIPs-like terminology, we define a domain as a family of related MDPs with the same state space, action space, and transition dynamics, but different reward functions, terminal states, and initial states. Analogously, a STRIPs problem is a specific MDP with a known reward function, set of terminal states, and an initial state. We are concerned with MDPs where the reward function is goal-directed, that is the agent receives a negative reward at each timestep that motivates it to reach the goal terminal state. A goal, G is defined by a reward function plus the set of terminal states. The aim of our approach is to learn knowledge about a domain from experience that enables an agent to quickly infer a solution to a new problem in the same domain that it has never previously encountered.

(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 the Minecraft domain

Modeling the Optimal Actions

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

$$\mathcal{A}^* = \{a \mid Q^*_G(s,a) = V^*_G(s)\}, \quad (1)$$

where $Q^*_G(s,a)$ and $V^*_G(s)$ represent the optimal Q function and value function, respectively.

We aim to learn a probability distribution over the optimality of each action for a given state (s), goal (G). Thus, we want to infer a Bernouli distribution for each action's optimality:

$$\Pr(a_i \in \mathcal{A}^* \mid s, G) \quad (2)$$

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

To generalize across specific low-level states, we abstract the state and goal into a set of n paired preconditions and goal types, $\{(p_1, g_1) \ldots (p_n, g_n)\}$. We abbreviate each pair (p_j, g_j) to δ_j for simplicity. Each precondition $p \in \mathcal{P}$ is a *predicate* in predicate space, $\mathcal{P}$, defined by the OO-MDP domain, and g is a *goal type* which is a logical expression that defines a subset of goals. For example, a predicate might be $nearTrench(agent)$ which is true when the agent is standing near a trench. In general a precondition could be an arbitrary logical expression of the state; in our experiments we used unary predicates defined in the OO-MDP domain. A goal type 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. Depending on the agent's current goal, the relevance of each action changes dramatically. We rewrite Equation 2:

$$\Pr(a_i \in \mathcal{A}^* \mid s, G) = \Pr(a_i \in \mathcal{A}^* \mid \delta_1 \ldots \delta_n). \quad (3)$$

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

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

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:

$$\Pr(a_i \in \mathcal{A}^* \mid s, G, \delta_1 \ldots \delta_n) = \Pr(a_i \in \mathcal{A}^* \mid \phi_1, \ldots, \phi_n) \quad (5)$$

This distribution may be modeled in a number of ways making this approach quite flexible. One model that can be easily specified by an expert is an OR model that we refer to as *hard affordances*. In the hard affordance 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,

$$\Pr(a_i \in \mathcal{A}^* \mid \phi_1, \ldots, \phi_n) = \phi^i_1 \vee \ldots \vee \phi^i_m, \quad (6)$$

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. For example, Table 1 shows example expert-provided hard affordances that we used in our Minecraft domain.

Because the hard affordance 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. Learning has the potential to outperform hand-coded affordances by more flexibly adapting itself to the features that predict optimal actions over a large training set. An alternative more expressive model that does lend itself to learning is Naive Bayes. To implement a Naive Bayes affordance model, we first factor using Bayes' rule, introducing a parameter vector θ_i of feature weights:

$$= \frac{\Pr(\phi_1, \ldots, \phi_n, \mid a_i \in \mathcal{A}^*, \theta_i) \Pr(a_i \in \mathcal{A}^* \mid \theta_i)}{\Pr(\phi_1, \ldots, \phi_n | \theta_i)} \quad (7)$$

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

$$= \frac{\prod_{j=1}^{n} \Pr(\phi_j \mid a_i \in \mathcal{A}^*, \theta_i) \Pr(a_i \in \mathcal{A}^* \mid \theta_i)}{\Pr(\phi_1, \ldots, \phi_n | \theta_i)} \quad (8)$$

Finally, we define the prior on the optimality of each action to be the fraction of the time each action was optimal during training. This approach allows model parameters to be learned from optimal policies during training, and information from different state precondition functions to be combined to infer a distribution on affordances. Although we only explore hard affordance and Naive Bayes models in this work, other affordance models, like logistic regression and Noisy-Or, could also be used.

Learning the Optimal Actions

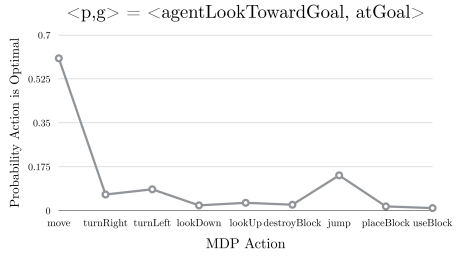
Our Naive Bayes affordance model allows affordances to be learned through experience. To learn affordances, we provide a set of training worlds from the domain (W), for which the optimal policy, π, may be tractably computed using existing planning methods. We compute model parameters using the small training worlds, and then evaluate performance on a different set of much harder problems at test time. To compute model parameters using Naive Bayes, we compute the maximum likelihood estimate of the parameter vector θ_i for each action using the policy.

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 $j \in$

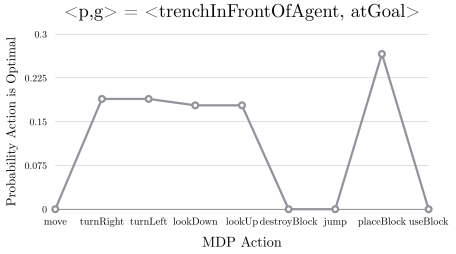
$\{1, \ldots, n\}$, where the maximum likelihood estimates are:

$$\theta_{i,0} = \frac{C(a_i)}{C(a_i) + C(\bar{a}_i)} \tag{9}$$

$$\theta_{i,j} = \frac{C(\phi_j, a_i)}{C(a_i)} \tag{10}$$



(a) agentLookTowardGoal



(b) trenchInFrontOfAgent

Figure 4: When the agent is looking toward the goal, it is generally better to move forward than do anything else. Alternatively, when the agent is faced with a trench, the agent should walk along the trench to look for gaps, or build a bridge across by looking down and placing blocks.

Here, $C(a_i)$ 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 for each training world, π_w. More formally:

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

$$C(\bar{a}_i) = \sum_{w \in W} \sum_{s \in w} (a_i \notin \pi_w(s)) \tag{12}$$

$$C(\phi_j, a_i) = \sum_{w \in W} \sum_{s \in w} (a_i \in \pi_w(s) \wedge \phi_j == 1) \tag{13}$$

During the learning phase, the agent learns which actions are useful under different conditions. For example, consider the three different problems shown in Figure 3. 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 ingots. 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 another affordance suggests using `destroy`). Example learned affordances are shown in .

Precondition	Goal Type	Actions
`lookingTowardGoal`	`atLocation`	{`move`}
`lavaInFront`	`atLocation`	{`rotate`}
`lookingAtGold`	`hasGoldOre`	{`destroy`}

Table 1: Examples of expert-provided affordances.

At test time, the agent will see different, randomly generated worlds from the same domain, and use the learned affordances to increase its speed at inferring a plan. For simplicity, our learning process uses a strict separation between training and test; after learning is complete our model parameters remain fixed.

Affordance-Aware Planning

An *affordance-aware planner* uses affordances to prune actions from the action set under consideration by the planner on a state and goal-specific basis. When a hard affordance model is used, defining the action pruning method is trivial; when $\Pr(a_i \in \mathcal{A}^* \mid \phi_1, \ldots, \phi_n) = 0$ action a_i is pruned from the planner's consideration. When $\Pr(a_i \in \mathcal{A}^* \mid \phi_1, \ldots, \phi_n) = 1$, action a_i remains in the action set to be searched by the planner.

When a non-deterministic affordance model, like Naive Bayes, is used, a less restrictive approach must be taken. In this work, we prune actions whose probability is below some threshold and keep the rest. Empirically, we found the heuristic of setting the threshold to $\frac{0.2}{|\mathcal{A}|}$ to be effective (where $|\mathcal{A}|$ is the size of the full action space of the OO-MDP). This threshold is quite conservative and means that only actions that are extremely unlikely to be optimal are pruned. In the future, we plan on exploring stochastic action pruning methods with anytime planning algorithms, but an advantage to this threshold pruning approach is that it can be used in a large range of different planning algorithms including A* (for deterministic domains), Value Iteration, and Real-time Dynamic Programming (RTDP) [4]. In this work, we present results using RTDP.

Results

We evaluate our approach using the game Minecraft and a collaborative robotic cooking task. Minecraft is a 3-D blocks game in which the user can place, craft, and destroy blocks of different types. Minecraft's physics and action space allow users to create complex systems, including logic gates and functional scientific graphing calculators[1]. Minecraft serves as a model for robotic tasks such as

[1] https://www.youtube.com/watch?v=wgJfVRhotlQ

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

Table 2: RTDP vs. affordance-aware alternatives.

cooking assistance, assembling items in a factory, object retrieval, and complex terrain traversal. As in these tasks, the agent operates in a very large state-action space in an uncertain environment. Figure 3 shows three example scenes from Minecraft problems that we explore. Additionally, we used expert-provided affordances to enable a manipulator robot to infer helpful actions in response to a person working on a kitchen task, shown in Figure 6.

Minecraft

Our experiments consisted of five common tasks in Minecraft, including constructing bridges over trenches, smelting gold, tunneling through walls, basic path planning, and digging to find an object. We tested on randomized worlds of varying size and difficulty. The generated test worlds varied in size from tens of thousands of states to hundreds of thousands of states. The agent learned affordances from a training set consisting of 25 simple state spaces of each map type (100 total maps), each approximately a 1,000-10,000 state world. We conducted all tests with a single knowledge base. Learning this knowledge base took approximately one hour run in parallel on a computing grid.

We fix the number of features at the start of training based on the predicates, $|P|$ and goal types, $|G|$; we had a total of 51 features. In the limit, if there was a feature for each state, it would learn an affordance for each state in the space, corresponding to a policy for the entire domain (since affordances are conditioned on goal descriptions). Because the number of features we are using is significantly smaller than the size of the state/action space, the agent learns general knowledge that is not specific to a single problem.

We use Real-Time Dynamic Programming (RTDP) [4]

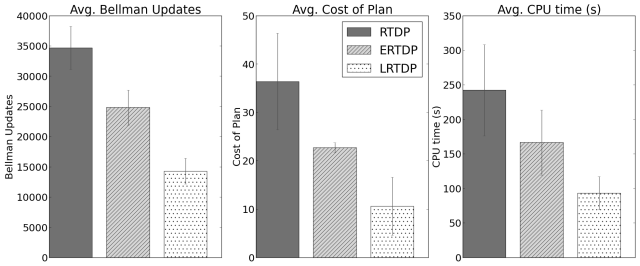


Figure 5: Average results from all maps.

as our baseline planner, a sampling-based algorithm that does not require the planner to visit all states. We compare RTDP with learned affordance-aware RTDP (LA-RTDP), and expert-defined affordance-aware RTDP (EA-RTDP). We terminated each planner when the maximum change in the value function was less than 0.01 for 100 consecutive policy rollouts, or the planner failed to converge after 1000 rollouts. The reward function was -1 for all transitions, except transitions to states in which the agent was in lava, where we set the reward to -10. The goal was set to be terminal and the discount factor was $\gamma = 0.99$. To introduce non-determinism into our problem, movement actions (move, rotate, jump) in all experiments had 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.

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 2 shows the average Bellman updates, accumulated reward, and CPU time for RTDP, LA-RTDP and EA-RTDP after planning in 25 different maps of each goal type (100 total). Figure 5 shows the results averaged across all maps. 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 2. Despite this noise, overall the average CPU time shows statistically significant improvement overall with affordances, 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 EA-RTDP and LA-RTDP.

Because the planners were forced to terminate after only 1000 rollouts, they did not always converge to the optimal policy. LA-RTDP on average found a comparably better plan (10.6 cost) than EA-RTDP (22.7 cost) and RTDP (36.4 cost), found the plan in significantly fewer Bellman updates (14287.5 to EA-RTDP's 24804.1 and RTDP's 34694.3) and in less CPU time (93.1s to EA-RTDP's 166.4s and RTDP's 242.0s). These results indicate that while learned affordances gave the largest improvements, expert-provided affordances can also significantly enhance performance, and,

depending on the domain, could add significant value in making large state-spaces tractable without the overhead of supplying training worlds.

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

Temporally Extended Actions and Affordances

We compared our approach to Temporally Extended Actions: macro-actions and options. We conducted these experiments with the same configurations as our Minecraft experiments. Domain experts provided the option policies and macro-actions.

Table 3 indicates the results of comparing RTDP equipped with macro-actions, options, and affordances across 100 different executions in the same randomly generated Minecraft worlds. The results are averaged across tasks of each type presented in Table 2. Both macro-actions and options add a significant amount of time to planning due to the fact that that the options and macro-actions are being reused in multiple 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 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 affordances, the planner found a better plan in less CPU time, and with fewer Bellman updates. These results support the claim that affordances can handle the augmented action space provided by temporally extended actions by pruning away unnecessary actions, and that options and affordances provide complementary information.

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

Table 3: Affordances with Temporally Extended Actions

Planner	Bellman	Reward	CPU
Dry Ingredients			
`RTDP`	20000 (±0)	-123.1 (±0)	56.0s (±2.9)
`EA-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)
`EA-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)
`EA-RTDP`	**6642.4** (±36.4)	**-7.0** (±0)	**31.9s** (±0.4)

Table 4: RTDP vs. EA-RTDP for robotic kitchen tasks

Cooking Robot

To assess affordances applied to a real-world robotic task, we created a cooking domain that requires the robot to choose helpful actions for a person following a recipe.

There were three spaces: human counter, robot counter, sink. We had four ingredient bowls, two mixing bowls, and two tools that could be in any of the three spaces, in any configuration. There were four ingredients: cocoa powder, sugar, eggs, and flour. Each container/tool could be on one of three spaces, and each ingredient in one of the containers in an either mixed or unmixed state. (e.g., the state of all the ingredients in a mixing bowl is different than the state of the dries mixed with the wets mixed individually in a bowl.) Althoug this fine-grained state-space is much larger than needed for any one recipe, it enables support for a variety of different recipes, ranging from brownies to mashed potatoes. Because of its fine-grained nature, our cooking state space has 4.73×10^7 states when configured with the ingredients and tools necessary to make brownies.

We divided a brownie recipe into three subgoals: combining and mixing the dry ingredients, combining and mixing the wet ingredients, and combining these two mixtures into a batter. For each subgoal, we provided affordances specific to the objects used in that subgoal; for example, a whisk should only be used to mix wet ingredients. We used EA-RTDP to search for the least-cost plan to complete the recipe. The robot inferred actions such as handing off the whisk to the person to mix the wet ingredients.

Figure 6: Affordances enable a robot to efficiently infer helpful actions in very large state spaces, such as a kitchen.

In Table 4 we compare between standard RTDP and EA-RTDP planning for each of the three subgoals. Because the state-action space is reduced significantly, EA-RTDP can plan successfully in a short amount of time. Standard RTDP always encountered the maximum number of rollouts specified at the maximum depth each time, even with a relatively small number of objects. (Unlike our previous CPU time results, these results took place on the same multi-core computer.)

We provide a video[2] showing how this affordance-aware planner 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.

Related Work

In this section, we discuss the differences between affordance-aware planning and other forms of knowledge engineering that have been used to accelerate planning. This paper builds on previous work[3] published at two workshops (redacted for review).

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* [13] and *options* [28]. 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 [18, 16, 24, 7, 1, 17].

Since temporally extended actions may negatively impact planning time [14] by adding to the number of actions the agent can choose from in a given state, combining affordances with temporally extended actions allows for even further speedups in planning, as demonstrated in Table 3. In other words, affordances are complementary knowledge to options and macro-actions.

Sherstov and Stone [27] 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. Affordances prune away actions on a state-by-state basis, enabling more aggressive pruning whereas the learned action pruning is on per-task level.

Rosman and Ramamoorthy [26] 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. Action priors can only be used with planning/learning algorithms that work well with an ϵ-greedy rollout policy, while affordances can be applied to almost any MDP solver. Action priors are only active for a fraction ϵ of the time steps, 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, affordance-aware planning can handle a wide variety of tasks in a single knowledge base, as demonstrated by Table 2.

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 [12], or reward shaping[23]. However, heuristics are highly dependent on the reward function and state space of the task being solved, whereas affordances are state space independent and may be learned easily for different reward functions. If a heuristic can be provided, the combination of heuristics and affordances may even more greatly accelerate planning algorithms than either approach alone.

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 [10]. 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 affordances both achieve action pruning by exploiting some form of supplied knowledge.

However HTNs do not incorporate reward into their planning. Consequently, they lack any guarantees of the quality of any induced plan. Additionally, the degree of supplied knowledge in HTNs far exceeds that of affordances: HTNs require not only constraints for sub-tasks but a hierarchical framework of arbitrary complexity. Affordances require either simple symbolic knowledge, as illustrated in Table 1, or a set of predicates for use as features and a means of generating candidate state spaces.

Bacchus and Kabanza [2, 3] 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

[2]Watch at https://vimeo.com/106226282

[3]A previous version of the proposed approach may be seen planning in Minecraft at https://vimeo.com/88689171

polynomial time planning in exponential space. LTL formulas are difficult to learn, placing dependence on an expert, while we demonstrate that affordances can be automatically learned from experience.

Our approach is related to preferred actions used by LAMA [25] in that our agent learns actions which are useful for a specific problem and expands those actions first. However our approach differsn in that it generalizes this knowledge across different planning problems, so that the preferred actions in one problem influence search in subsequant problems in the domain.

Conclusion

We proposed a novel approach to representing transferable planning knowledge in terms of *affordances* [11]. Affordances allow an agent to efficiently prune actions based on learned or expert provided knowledge, significantly reducing the number of state-action pairs the agent needs to evaluate in order to act near optimally. We demonstrated the effectiveness of the affordance model by comparing RTDP to its affordance-aware equivalents in a series of challenging planning tasks in the Minecraft domain. Further, we designed a learning process that allows an agent to autonomously learn useful affordances that may be used across a variety of task types, reward functions, and state spaces, allowing for convenient extensions to robotic applications. Additionally, we compared the effectiveness of augmenting planners with affordances with temporally extended actions, and the combination of the two. The results suggest that affordances may be combined with temporally extended actions to provide improvements in planning. Lastly, we deployed an affordance-aware planner on a robot in a collaborative cooking task.

In the future, we hope to automatically discover useful state space specific subgoals online—a topic of some active research [19, 8]. Automatic discovery of subgoals would allow affordance-aware planners to take advantage of the goal-oriented focus of affordances, and would further reduce the size of the explored state-action space by improving the effectiveness of action pruning. Finally we are excited to explore learning affordances from other sources besides learning from policies. For example, learning from human feedback via imitation learning or reinforcement learning could allow our agent to bias its actions in promising directions by following a person's example. Another promising direction is to explore continuous learning. In this mode, the agent uses affordances as it learns, allowing it to build a knowledge base that expands as it goes.

Additionally, we hope to explore methods that capitalize on the distribution over optimal actions, such as incorporating affordances with a forward search sparse sampling algorithm [30], or replacing the Naive Bayes model with a more sophisticated model, such as Logistic Regression or a Noisy-OR. We are also investigating methods to stochastically prune actions rather than requiring a hard threshold parameter to be defined (or estimated). These methods will enable agents to acquire and leverage general planning knowledge to infer actions in very large state spaces.

References

[1] 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.

[2] 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.

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

[4] 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.

[5] Mario Bollini, Stefanie Tellex, Tyler Thompson, Nicholas Roy, and Daniela Rus. Interpreting and executing recipes with a cooking robot. In *Proceedings of International Symposium on Experimental Robotics (ISER)*, 2012.

[6] 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.

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

[8] 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*, ICML '05, pages 816–823, New York, NY, USA, 2005. ACM. ISBN 1-59593-180-5. doi: 10.1145/1102351.1102454. URL `http://doi.acm.org/10.1145/1102351.1102454`.

[9] 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.

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

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

[12] 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.

[13] 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.

[14] 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.

[15] Ross A. Knepper, Stefanie Tellex, Adrian Li, Nicholas Roy, and Daniela Rus. Single assembly robot in search of human

partner: Versatile grounded language generation. In *Proceedings of the HRI 2013 Workshop on Collaborative Manipulation*, 2013.

[16] 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.

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

[18] 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.

[19] 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.

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

[21] 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, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc. URL `http://dl.acm.org/citation.cfm?id=1624312.1624357`.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic robotics*. MIT Press, 2008.

[30] Thomas Walsh, Sergiu Goschin, and Michael Littman. Integrating sample-based planning and model-based reinforcement learning, 2010. URL `http://www.aaai.org/ocs/index.php/AAAI/AAAI10/paper/view/1880`.