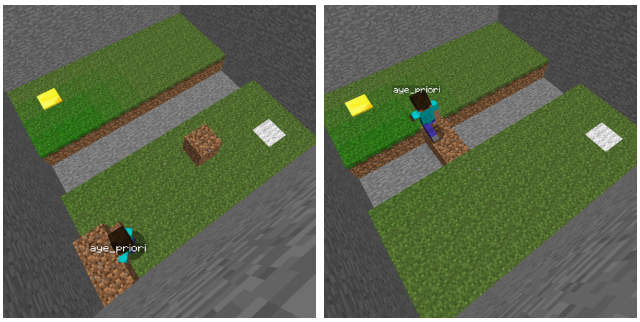
Planning with Affordances

Abstract

Planning algorithms for non-deterministic domains are often intractable in large state and action spaces due to the well-known "curse of dimensionality." Approaches to address this problem by providing the system with formally encoded knowledge still fail to prevent the system from considering many actions which would be obviously irrelevant to a human solving the same problem. We introduce a novel, state- and reward-general approach to limiting the branching factor in large domains by encoding knowledge about the domain in terms of *affordances* [11]. Our affordance formalism can be coupled with a variety of planning frameworks to create "affordance-aware planning," allowing an agent to efficiently prune its action space based on domain knowledge. This pruning significantly reduces the number of state/action pairs the agent needs to evaluate in order to act optimally. We demonstrate our approach in the Minecraft domain, showing significant increase in speed and reduction in state-space exploration compared to the standard versions of these algorithms.

1 INTRODUCTION

As robots move out of the lab and into the real world, planning algorithms need to scale to domains of increased noise, size, and complexity. A classic formalization of this problem is a stochastic sequential decision making problem in which the agent must find a policy (a mapping from states to actions) for some subset of the state space that enables the agent to achieve a goal from some initial state, while minimizing any costs along the way. Increases in planning problem size and complexity directly correspond to an explosion in the state-action space. Current approaches to solving sequential decision making problems cannot tackle these problems as the state-action space becomes too large [12].



(a) Planning with VI. (b) Planning with affordances.

Figure 1: Scenes from a Minecraft agent planning using Value Iteration (VI) compared to affordance-aware VI in a bridge building task. VI considers states which would be obviously irrelevant to a human solving the same problem, such as stacking blocks in the corner. Our affordance agent, in contrast, focuses on placing blocks in the trench, which are much more relevant to reaching the goal.

To address this state-space explosion, prior work has explored adding knowledge to the planner to enable it to solve problems in these massive domains. However, previous approaches such as options and macro-actions work by providing additional high-level actions to the agent, which *increases* the size of the state/action space (while also allowing the agent to search more deeply within the space). The resulting augmented space is even larger, which can have the paradoxical effect of increasing the search time for a good policy.

Instead, we propose a formalism that enables an agent to focus on problem-specific aspects of the environment, guiding the search toward the most relevant

and useful parts of the state-action space. This approach reduces the size of the explored state action space, leading to dramatic speedups in planning. Our approach is a formalization of *affordances*, introduced by Gibson [11] as "what [the environment] offers [an] animal, what [the environment] provides or furnishes, either for good or ill."

We formalize the notion of an affordance as a piece of planning knowledge provided to an agent operating in a Markov Decision Process (MDP). We explain how affordances can be leveraged by a variety of planning algorithms to prune the action set the agent uses dynamically based on the agent's current goal. We call any planning algorithm that uses affordances to prune the action set an *affordance-aware* planning algorithm. Affordances are not specific to a particular reward function or goal, and thus, provide the agent with transferable knowledge that is effective in a wide variety of problems.

We use Minecraft as our planning and evaluation domain. Minecraft is a 3-D blocks world game in which the user can place and destroy blocks of different types. Minecraft players have constructed complex worlds, including models of a scientific graphing calculator [1]; scenes from a Minecraft world appear in Figure 1.

Minecraft serves as an effective parallel for the actual world, both in terms of approximating the complexity and scope of planning problems, as well as modeling the uncertainty and noise presented to an real world agent (e.g. a robot). For instance, robotic agents are prone to uncertainty all throughout their system, including noise in their sensors (cameras, LIDAR, microphones, etc.), odometry, control, and actuation. In order to accurately capture some of the inherent difficulties of planning under uncertainty, the Minecraft agent (and the API through which it is controlled) as well as the planning tasks we experiment on all capture the stochasticity that affect robotics agents. We have chosen to give the Minecraft agent non-noisy sensory data about the Minecraft world, as that is outside the scope of this project.

One of the most significant difficulties of dealing with an agent planning in a noisy world (such as a robot) is that many of its actions can have unintended consequences, often leading to disastrous results. We model these potentially dangerous situations by requiring that the actions in the Minecraft domain be non-deterministic. Furthermore, we add pits of lava to many (but not all) of our experiments, which force the agent to plan conservatively in order to avoid falling in.

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

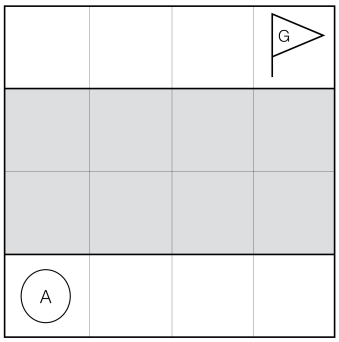


Figure 2: In the above Minecraft planning problem `BRIDGEWORLD`, the agent must place a block in the trench in order to reach the goal (the trench is too wide to jump over).

As a running example, we will consider the problem of an agent attempting to cross a trench in a $4 \times 4 \times 2$ Minecraft world shown in Figure 2. The floor (at $z = 1$) [2] is composed of 8 solid blocks, with horizontal empty trenches at $y = 2$ and $y = 3$. The agent is at the starting location $(1, 1, 2)$ and needs to reach the goal at $(4, 4, 2)$

To solve the problem, the agent must place a block in the trench to form a bridge, then cross the bridge to reach the goal. This task is challenging for planning algorithms to solve because the reachable state space in Minecraft is so large. For example, the number of places an agent can place and destroy blocks alone can result in a combinatorial explosion of the state space. Given an agent capable of placing and destroying blocks in a world with dimensions $w \times l \times h$, there are:

$$O\left(\sum_{n=1}^{w \cdot l \cdot h} \binom{w \cdot l \cdot h}{n}\right) \tag{1}$$

states, which is too large for a standard planner to explore in a reasonable time.

An affordance-aware planner, however, (when equipped with the proper affordances) will only attempt to place a or destroy a block when it is useful. The usefulness of a given action is congruent to how effective that action is at moving the agent closer to the goal (akin to a heuristic). Thus, when the agent is in states that are not considered useful (i.e. the predicate p is false), the agent will not have access to the block placement action. This prevents the agent from trying countless applications of actions that would ultimately not contribute towards reaching the goal.

[2] The z-axis is the height of the Minecraft world. Similarly, the x-axis is its width and the y-axis is its length.

2 BACKGROUND

We define affordances in terms of propositional functions on states. Our definition builds on the Object-Oriented Markov Decision Process (OO-MDP) [10]. OO-MDPs [10] are an extension of the classic Markov Decision Process (MDP). A classic 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 and transitioning to state s'; and $\gamma \in [0, 1)$ is a discount factor that defines how much the agent prefers immediate rewards over distant rewards (the agent more greatly prefers to maximize more immediate rewards as γ decreases).

A classic way to provide a factored representation of an MDP state is to represent each MDP state as a single feature vector. In contrast, an OO-MDP represents the state space as 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 $Dom(c.a)$. Upon instantiation of an object class, its attributes are given a state $o.state$ (an assignment of values to its attributes). The underlying MDP state is the set of all the object states: $s \in \mathcal{S} = \cup_{i=1}^{o} \{o_i.state\}$.

There are two advantages to using an object-oriented factored state representation instead of a single feature vector. First, different states in the same state space may contain different numbers of objects of varying classes, which is useful in domains like Minecraft in which the agent can dynamically add and remove blocks to the world. Second, MDP states can be defined invariantly to the specific object references. For instance, consider a Minecraft world with two block objects, b_1 and b_2. If the agent picked up and swapped the position of b_1 and b_2 (and then returned to the agent's previous position in the world), the MDP state before the swap and after the swap would be the same, because the MDP state definition is invariant to which object holds which object state. Formally, if there exists a bijection between two sets of objects that maps each object in one set to an object in the other set with the same object state, then the two sets of objects define the same MDP state. This object reference invariance results in a smaller state space compared to representations like feature vectors in which changes to value assignments always result in a different state.

While the OO-MDP state definition is a good fit for the Minecraft domain, our motivation for using an OO-MDP lies in the ability to formulate 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. For example, in `BRIDGEWORLD`, a `nearTrench(AGENT)` predicate evaluates to true when the singular instance of class `AGENT` is directly adjacent to an empty location at floor level (i.e. the cell beneath the agent in some direction does not contain a block). In the original OO-MDP work, these predicates were used to model and learn an MDP's transition dynamics. In the next section, we use the predicates to define affordances that enable planning algorithms to prune irrelevant actions.

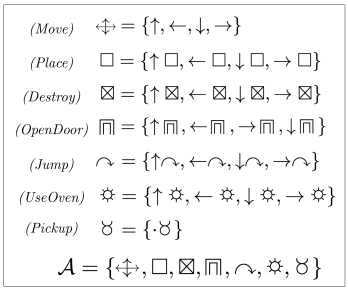


Figure 3: The set of all actions in the Minecraft domain.

3 AFFORDANCES

In many planning scenarios, not all actions are needed in all states. In fact, many applications of actions in states do not contribute toward solving the planning task, but instead, cause the state-space to grow exponentially, especially true in domains in which the agent's actions can drastically shape the environment, such as in Minecraft. We capture this intuition via an affordance Δ, defined as a tuple, $\langle p, g \rangle \longmapsto \alpha$, where:

α is a subset of the action space, $\mathcal{A}$

p is a predicate on states, $s \longrightarrow \{0, 1\}$ representing the *precondition* for the affordance.

g is a lifted goal description, represented as an ungrounded predicate on states, g.

The precondition, p refers to a predicate over an OO-MDP state, where an OO-MDP state is represented as a union of all of the objects' current attribute values: $\cup_{i=1}^{o} o_i.state$. The use of an OO-MDP here makes predicates more general and robust (e.g. OO-MDP predicates may be relational) across tasks and is one of the reasons that affordance-aware planners may be untethered from specific state spaces, as the predicate of p benefit from the richness of the predicates of an OO-MDP. Further, the lifted goal description, g, is an

ungrounded predicate, describing the type of task being solved. In Minecraft, this can range from reaching a particular cell (*reachGoal*), to constructing an object of a certain type (e.g. baking bread, building a pickaxe).

Algorithm 1 pruneActions(*state*, *KB*)
Complexity: $\mathcal{O}(|KB|)$

```
for Δ ∈ KB do
    if Δ.p(state) and Δ.g == state.goal then
        α.update(Δ.p, Δ.g)
    end if
end for
return α
```

We encode a lifted goal description g with each affordance. If the agent is at the trench in `BRIDGEWORLD` and is faced with a *reachGoal* task, then it would not consider placing a block in any arbitrary location, as that would not further its ability to reach the goal. Instead, we endow the agent with the following affordance: $\Delta_1 = \langle nearTrench, reachGoal\rangle\rangle \longmapsto \{\square\}$, that tells the agent to try placing blocks when next to a trench (as the trench could inhibit its progress toward reaching the goal). Thus, in any *reachGoal* task, we would have the following affordances: $\Delta_2 = \langle onPlane, reachGoal\rangle\rangle \longmapsto \{\updownarrow\leftrightarrow\}$,
$\Delta_3 = \langle nearWall, reachGoal\rangle \longmapsto \{\boxtimes\}$. Eliminating the possibility of applying the "destroy" action in every state avoids exploring every consequent state in which that block has been destroyed, enabling an affordance-aware planner in *reachGoal* task types to handle large state spaces.

Further, agent's may be equipped with affordances with differing lifted goal descriptions, thus enabling an agent with a single, relatively minimal knowledge base to tackle planning problems of scale from a variety of goal types. We envision extensions to equipping A* with affordances with applications to a robotic cooking companion, as well as PO-MDP planners with affordances, with applications in robotic care-giver companions.

We propose the notion of an *affordance-aware* planner, which refers to a planning algorithm that prunes the actions set according to affordances. Action set pruning affects different planning algorithms in different ways. In particular, we focus on how action pruning benefits *dynamic programming*, *policy rollout*, and *subgoal* planning paradigms.

3.1 Dynamic Programming

In dynamic programming paradigms, the planning algorithm estimates the optimal *value function* for each state. Formally, the optimal value function (V^*) defines the expected discounted return from following the optimal policy in each state:

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

this equation is known as the Bellman equation [4]. Given the optimal value function, the optimal policy is derived by taking the action that maximizes the values of each state. More specifically, by taking the action with the highest optimal state-action value:

$$Q^*(s,a) = \sum_{s'} \Pr(s' \mid s, a)\left[\mathcal{R}(s,a,s') + \gamma V^*(s')\right]. \tag{3}$$

Dynamic programming planning algorithms (such as Value Iteration [4]) estimate the optimal value function by initializing the value of each state arbitrarily and iteratively updating the value of each state by setting its value to the result of the right-hand-side of the Bellman equation using its current estimate of V instead of V^*. Iteratively updating the value function estimate in this way is guaranteed to converge to the optimal value function.

Using a pruned action set in dynamic programming can accelerate its computation in two ways: (1) by reducing the number of actions over which the max operator in the Bellman equation must iterate and (2) by restricting the state space for which the value function is estimated to the states that are reachable with the pruned action set from the initial state. Note that neither of these computational gains come at the cost of solution optimality as long as the pruned action set contains the actions necessary for an optimal policy from the initial state. In the case of the Bellman equation, the max operator makes the value function indifferent to the effects of actions that are not part of the optimal policy; therefore, the action set can be reduced entirely to the actions in the optimal policy without sacrificing optimality. Similarly, since we are only concerned with finding a good policy to dictate behavior from some initial state, the state space for which the value function is computed can be reduced to that which is reachable using only the optimal actions without sacrificing optimality.

3.2 Policy Rollout

In policy rollout planning paradigms, the agent starts with some initial policy and follows it (or rolls out the policy) from an initial/current state to either some maximum time horizon or until a terminal state is reached. Often, these approaches use samples from the policy rollout to improve estimates of the value function and indirectly improve the rollout policy. Ex-

amples of planning algorithms in this paradigm include Monte Carlo methods [7, 26] and temporal difference methods [2, 19, 22, 24, 27, 28]. By using a pruned action set, the policy space, and resulting state space explored from the searched policies, is reduced, thereby reducing the number of rollouts necessary to find a good policy. Similar to dynamic programming paradigms, as long as the pruned action set contains actions necessary for the optimal policy, solution optimality will not be sacrificed.

In this work, we will explore how real time dynamic programming (RTDP) [3] benefits from affordances. RTDP is both a dynamic programming algorithm and a policy rollout algorithm. RTDP starts by initializing the value function optimistically. It then follows a greedy rollout policy with respect to its currently estimated value function. After each action selection in the policy rollout, RTDP updates its estimate of the value function for the last state using the Bellman equation. RTDP is guaranteed to converge to the optimal policy from some initial state and has the advantage that it iteratively refocuses its attention to states that are likely to be on the path of the optimal policy.

In affordance-aware RTDP, the action selection of the rollout policy is restricted to the affordance-pruned action set and the Bellman equation is similarly restricted to operating on the affordance-pruned action set.

3.3 Subgoal Planning

Subgoal planning leverages the intuition that certain goals in planning domains may only be brought about if certain preconditions are first satisfied. For instance, in the bridge problem, the agent must first place a block in the trench to create a bridge before crossing the trench. Branavan et al. [6] explore learning subgoals from the Minecraft wiki and applying them in order to plan through a variety of problems in Minecraft.

Formally, in subgoal planning, the agent is set of subgoals, where each subgoal is a pair of predicates:

$$SG = \langle x_k, x_l \rangle \tag{4}$$

where x_l is the effect of some action sequence performed on a state in which x_k is true. Thus, subgoal planning requires that we perform high-level planning in subgoal space, and low-level planning to get from subgoal to subgoal. The low-level planner may vary, though Metro-FF and A* are popular choices (depending on domain constraints), as is Value Iteration.

In the case of `BRIDGEWORLD`, the agent might consider placing a block somewhere along the trench to be a subgoal. Then, it runs a low-level planner to get from its starting location to the subgoal. Next, it runs the same low-level planner from the first subgoal to the finish. Subgoals enhance an agent's planning abilities when they propose *necessary* claims about the domain. If the subgoals are *contingent* (i.e. true in some state spaces of the domain but not in others), then they do not limit the search space (and in fact can negatively effect planning). For instance, consider the task in `BRIDGEWORLD`, in which the agent must place a block in the trench that separates the agent from the goal. The subgoal $\langle blockInTrench, reachGoal \rangle$ might be a perfectly useful subgoal in `BRIDGEWORLD`, but an adversary could easily come up with thousands of worlds in which such a subgoal would completely derail the agent's planner. Thus, many subgoals do not scale beyond a particular instance of a state space. In order for subgoals to be useful, they must be necessary claims about the domain, otherwise, one can always come up with a counter world (by definition of necessary). Compare this scenario to the problem of baking bread in Minecraft: possessing grain is always required to make bread, and it is impossible to construct a world where this precondition is not true.

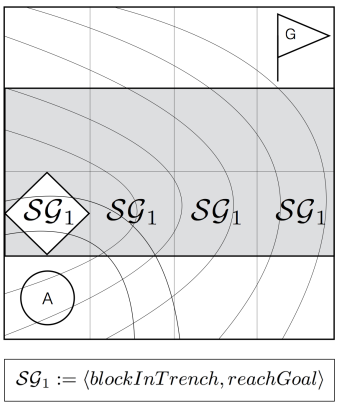


Figure 4: Subgoal planner in `BRIDGEWORLD`

Subgoal planners benefit in many ways by being made affordance-aware. One of the main problems of subgoal planning is that subgoal planners re-explore large portions of the state space, as illustrated by Fig. 4. The affordance-aware version of subgoal planners will tend to avoid this problem by focusing on actions that are likely to direct the agent to the goal. Since subgoals are handcrafted to be preconditions for arriving at the goal, a large portion of the state space should will be pruned away (namely, the portion that is only accessible through actions that do not take you toward the goal). Furthermore, the low-level planners still suffer from all of the standard issues of planning we have discussed above; particularly in the Minecraft domain, planners cannot scale to accommodate that

	VI	RTDP	SG	A-VI	A-RTDP	A-SG
4BRIDGE	39015	2368	3674	9660	259	214
6BRIDGE	162945	4894	4403	23000	916	473
8BRIDGE	496179	16873	?	41860	1975	944
DOORB	312579	11782	11322	92460	3285	1482
LAVAB	177174	3242	4490	41860	717	359
TUNNEL	196810	30676	6443	51430	581	220
BREAD	26502	8501	8564	1554	1124	589

Table 1: Tests on a variety of tasks that require block placement and destruction actions. (results incomplete as experiments are ongoing)

	VI	RTDP	SG	A-VI	A-RTDP	A-SG
10WORLD	1600	1369	1166	1600	1408	1051
15WORLD	5850	5920	4168	5850	4042	6016
20WORLD	14400	15645	18400.	14400	16233	15345
JUMP	2940	2857	2323	2940	2435	2146
DOOR	6315	2952	3122	6315	3075	2104
MAZE	4266	2857	1961	4266	2665	1418
LAVA	800	698	570	800	638	494
HARD	16588	7359	3295	16588	6611	3413

Table 2: Tests on a variety of tasks without block placement and destruction actions

state space sizes that are possible in Minecraft (and thus, the real world). Thus, we may make Subgoal planners affordance-aware by equipping the subsequent low-level planners with affordances in order to plan efficiently from subgoal to subgoal.

4 EXPERIMENTS

We conducted a series of experiments in the Minecraft domain that tested standard planners from each planning paradigm: Value Iteration, RTDP, and Subgoal planning (with RTDP as the low-level planner). These planners were compared with *affordance-aware* versions of each algorithm tasked with the same set of problems. Our experiments consisted of a variety of tasks, ranging from basic path planning, to baking bread, to opening doors and jumping over trenches. We also tested each planner on worlds of varying size and difficulty to demonstrate the scalability and flexibility of the affordance formalism. The evaluation metric for each trial is the number of state backups that were executed in each iteration of each planning algorithm.

Our entire set of experiments were repeated twice, the first iteration being deterministic, and the second iteration being non-deterministic.

4.1 RESULTS

Table 1 indicates the results of running the standard planners and their affordance aware counterparts on a set of tasks that require the use of block placement and/or destruction. For this set of experiments, the actions were all deterministic. Worlds XBRIDGE had a trench going through the middle which the agent needed to build a bridge over (by placing a block) in order to reach the goal. We also tested on worlds that included lava, doors the agent had to open, a bread baking world, and a world where a wall stood between the agent and the goal and the agent needed to cut down the wall to reach the goal. For testing, we used RTDP as the low level planner for the Subgoal Planner as well as for the affordance aware Subgoal Planner.

The affordance aware planners did significantly better than their ordinary counterparts in the majority of these experiments. They proved especially effective when paired with subgoals, as can be seen by the results of the affordance aware subgoal planner. These results were expected, as affordances are particularly useful if subgoal knowledge is known. The reason this is so effective is that, for each subgoal, different sets of affordances are chosen based on the lifted task descriptions of each affordance, allowing different types of pruning to occur depending on what the agent is trying to accomplish at each stage. Additionally, affordance aware VI and affordance aware RTDP outperforms their regular planners; this is expected, too, as the other planners do little to prune away the majority of the useless action applications in these block building, block destruction, and bread baking types of tasks.

Table 2 indicates the results of running the standard planners and their affordance aware counterparts on tasks that do *not* require the use of powerful actions like block placement and block destruction. In these cases, the affordance awareness did not improve any of the planners from their baseline versions, which are precisely the results we would expect to see. Affordances are only beneficial in those cases where powerful actions, which allow the agent to combinatorially alter the state space, may be pruned. In each of these worlds, the agent only needed basic "weak" actions, such as opening doors, jumping, and movement. Thus, they are not good candidate tasks for affordance aware planning.

This highlights the fact that affordance aware planers are not always the right way to fix planning scalability issues - while effective in many cases, they do fall short in some regards. For one, they are specific to task types, as the lifted goal description g of each affordance must be defined with a task type (e.g. a predicate that defines the goal) in mind. Additionally, affordances (if defined optimally) only prune away actions that are useless in achieving the goal, but fail to prune away actions whose importance is not easily inferred. For instance, in BRIDGEWORLD, when the agent is up against a trench, it will still explore the space in

which it places blocks behind it, and to its side, despite the fact that these are not useful applications of the place action. In future work, we would like to change the precondition p to a set of features that represent the salient portions of the agent's current state and output a distribution over the agent's action set. This would allow an affordance aware planner to *prioritize* and rank actions in each state, which would lead to pruning much larger portions of the state-space.

The final factor to consider when augmenting a planner with affordances is whether or not the desired domain provides the agent with actions that can dramatically affect the state space. If the action set is relatively small and the actions don't significantly impact the shape of the state-space with each application (as is the case for block placement), then affordances are not likely to help. For instance, consider the classic Reinforcement Learning problem of balancing an inverted pendulum - in this task, the agent must choose between moving the base of an inverted pendulum $left$ and $right$ and attempt to balance the pendulum in equilibrium. For tasks such as this, equipping a planner with affordances would have no impact. However, in cases where actions can affect the state-space dramatically (i.e. result in combinatoric explosions of the state space), affordances can help many planning systems.

One of the most compelling results is the scope of task that affordance-aware planners are capable of solving. With an affordance-aware Subgoal planner (i.e. using an affordance aware RTDP as the low level planner), a Minecraft agent was able to solve an obstacle course in which it must open a door to find a block of grain, pick up the grain, avoid lava, build a bridge over a trench, and finally place the grain in the oven and bake it. In this course, the agent was capable of completing several different types of tasks (subgoals) with a single action space and affordance knowledge base. Another interesting example was that, given only a single block and the ability to jump in a world with a trench of 2 blocks wide, the planner placed a block in the trench, narrowing the gap enough to jump across and reach the goal.

5 Related Work

In the past, numerous different forms of background knowledge have be used to accelerate planning algorithms. In section 3.3, subgoal planning was discussed and in our experimental results, was compared against affordance-aware planning. In this section, we discuss the differences between affordance-aware planning and other forms of background knowledge that have been used to accelerate planning. Specifically, we discuss heuristics, temporally extended actions, and related action pruning work.

5.1 Heuristics

Heuristics in MDPs are used to convey information about the value of a given state or state-action pair with respect to the task being solved and typically take the form of either *value function initialization*, or *reward shaping*. For planning algorithms that estimate state-value functions, heuristics are often provided by initializing the value function to values that are good approximations of the true value function. For example, initializing the value function to an admissible close approximation of the optimal value function has been shown to be effective for LAO* and RTDP, because it more greatly biases the states explored by the rollout policy to those important to the optimal policy [13]. Planning algorithms that estimate Q-values instead of the state value function may similarly initialize the Q-values to an approximation of the optimal Q-values. For instance, PROST [15] creates a *determinized* version of a stochastic domain (that is, treating each action as if its most likely outcome always occurred), plans a solutions in the determinized domain, and then initializes Q-values to the value of each action in the determinized domain.

Reward shaping is an alternative approach to providing heuristics in which the planning algorithm uses a modified version of the reward function that returns larger rewards for state-action pairs that are expected to be useful. Reward shaping differs from value function initialization in that it may not preserve convergence to an optimal policy unless certain properties of the shaped reward are satisfied [21] that also have the effect of making reward shaping equivalent to value function initialization for a large class of planning/learning algorithms [29].

A critical difference between heuristics and affordances is that heuristics are highly dependent on the task being solved; therefore, different tasks require different heuristics to be provided, whereas affordances are state independent and transferable between different state-spaces. However, if a heuristic can be provided, the combination of heuristics and affordances may even more greatly accelerate planning algorithms than either approach alone.

5.2 Temporally Extended Actions

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*and *op-*

tions [27]. 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 hand crafted to open doors. An option o is defined as follows:

$o = \langle \pi_0, I_0, \beta_0 \rangle$, where:

$$\pi_0 : (s, a) \rightarrow [0, 1]$$

$$I_0 : s \rightarrow \{0, 1\}$$

$$\beta_0 : s \rightarrow [0, 1]$$

Here, π_0 represents the *option policy*, I_0 represents a precondition, under which the option policy may initiate, and β_0 represent the post condition, which determines which states terminate the execution of the option policy.

Although the classic options framework is not generalizable to different state spaces, creating *portable* options is a topic of active research [1, 9, 16, 17, 18, 23].

Although temporally extended actions are typically used because they represent action sequences (or sub policies) that are often useful to solving the current task, they can sometimes have the paradoxical effect of increasing the planning time because they increase the number of actions that must be explored. For example, deterministic planning algorithms that successfully make use of macro-actions often avoid the potential increase in planning time by developing algorithms that restrict the set of macro-actions to a small set that is expected to improve planning time [5, 20] or by limiting the use of macro-actions to certain conditions in the planning algorithms like when the planner reaches heuristic plateaus (areas of the state space in which all child states have the same heuristic value) [8]. Similarly, it has been shown that the inclusion of even a small subset of unhelpful options can negatively impact planning/learning time [14].

Given the potential for unhelpful temporally extended actions to negatively impact planning time, we believe combing affordances with temporally extended actions may be especially valuable, because it will restrict the set of temporally extended actions to those which may actually be useful to a task. In the future, we plan to more directly explore the benefit from combining these approaches.

5.3 Action Pruning

Perhaps the most similar work to ours is Sherstov and Stone's action transfer work [25]. In their work, they considered MDPs with a very large action set and 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. Since the actions of the optimal policy of a source task may not include all the actions of the optimal policy in the target task, source task action bias was reduced by randomly perturbing the value function of the source task to produce new synthetic tasks. The action set transferred to the target task was then taken as the union of the actions in the optimal policies for the source task and all the synthetic tasks generated from it.

A critical difference between our affordance-based action set pruning and this action transfer work is that affordances prune away actions on a state by state basis. Therefore, affordance aware planners are significantly more flexible as they move throughout a state space, as certain actions are useful for a given planning task in general, but not in specific subspaces of the statespace. Further, with the lifted goal descriptions, affordances may be hooked into Subgoal planning neatly for a huge benefit in planning tasks where complete subgoal knowledge is known (or may be inferred).

6 CONCLUSION

We proposed a novel approach to representing knowledge in terms of *affordances* [11] that allows an agent to efficiently prune its action space based on domain knowledge. This led to the proposal of affordance-aware planners, which improve on classic planners by providing a significant reduction in the number of state/action pairs the agent needs to evaluate in order to act optimally. We demonstrated the efficacy as well as the portability of the affordance model by comparing standard paradigm planners to their affordance-aware equivalents in a series of planning tasks in the Minecraft domain.

In the future, we hope to introduce a more robust inference system around pruning actions, such that the agent not only prunes away *useless* actions, but also prioritizes between *great* actions, and *mediocre* ones. This also lends itself nicely for applying machine learning techniques to learn affordances directly, which would overcome the fact that a human designer must currently hand craft affordance knowledge bases. Further, we foresee extensions in natural language processing and information extraction, in which affordances may be inferred via text or from dialogue with a human partner. This promises extensions in which a robotic agent receives aid from a human partner through natural language dialogue; the agent may

ask for help when it is stuck and receive affordance or subgoal *hints* from a human companion. Lastly, we consider applications to POMDPs, a classical extension of the MDP in which the agent must make decisions with respect to a distribution over a *belief* state, as opposed to knowing precisely what state it is in at all times.

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] A.G. Barto, R.S. Sutton, and C.W. Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *Systems, Man and Cybernetics, IEEE Transactions on*, SMC-13(5): 834 –846, sept.-oct. 1983.

[3] 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.

[4] Richard Bellman. Dynamic programming, 1957.

[5] 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.

[6] S.R.K. Branavan, Nate Kushman, Tao Lei, and Regina Barzilay. Learning high-level planning from text. In *Proceedings of the Conference of the Association for Computational Linguistics*, ACL '12, 2012.

[7] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *Computational Intelligence and AI in Games, IEEE Transactions on*, 4(1):1–43, 2012.

[8] Andrew Coles and Amanda Smith. Marvin: A heuristic search planner with online macro-action learning. *Journal of Artificial Intelligence Research*, 28:119–156, 2007.

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

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

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

[12] Matthew Grounds and Daniel Kudenko. Combining reinforcement learning with symbolic planning. In *Proceedings of the 5th, 6th and 7th European conference on Adaptive and learning agents and multi-agent systems: adaptation and multi-agent learning*, ALAS '05, 2005.

[13] 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.

[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] T. Keller and P. Eyerich. Prost: Probabilistic planning based on uct. In *International Conference on Automated Planning and Scheduling*, 2012.

[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] M.G. Lagoudakis and R. Parr. Least-squares policy iteration. *The Journal of Machine Learning Research*, 4:1107–1149, 2003.

[20] 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.

[21] 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.

[22] Jan Peters and Stefan Schaal. Natural actor-critic. *Neurocomputing*, 71(7):1180–1190, 2008.

[23] 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.

[24] G.A. Rummery and M. Niranjan. On-line q-learning using connectionist systems. Technical Report 166, University of Cambridge, Department of Engineering, 1994.

[25] 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.

[26] David Silver and Joel Veness. Monte-carlo planning in large pomdps. In *NIPS*, volume 23, pages 2164–2172, 2010.

[27] 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.

[28] R.S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3(1): 9–44, 1988.

[29] Eric Wiewiora. Potential-based shaping and q-value initialization are equivalent. *Journal of Artificial Intelligence Research*, 19:205–208, 2003.