Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence (IJCAI 2015)

Portable Option Discovery for Automated Learning Transfer in Object-Oriented Markov Decision Processes

Nicholay Topin, Nicholas Haltmeyer, Shawn Squire, John Winder,
Marie desJardins, James MacGlashan*
Department of Computer Science and Electrical Engineering
University of Maryland, Baltimore County
{ntopin1, hanicho1, ssquire1, jwinder1, mariedj}@umbc.edu, jmacglashan@cs.brown.edu

Abstract

We introduce a novel framework for option discovery and learning transfer in complex domains that are represented as object-oriented Markov decision processes (OO-MDPs) [Diuk *et al.*, 2008]. Our framework, Portable Option Discovery (POD), extends existing option discovery methods, and enables transfer across related but different domains by providing an unsupervised method for finding a mapping between object-oriented domains with different state spaces. The framework also includes heuristic approaches for increasing the efficiency of the mapping process. We present the results of applying POD to Pickett and Barto's [2002] PolicyBlocks and MacGlashan's [2013] Option-Based Policy Transfer in two application domains. We show that our approach can discover options effectively, transfer options among different domains, and improve learning performance with low computational overhead.

1 Introduction

Reinforcement learning (RL) typically studies learning algorithms applied to single, well-defined environments. RL agents seek to find an optimal policy for their environment, where a policy, $\pi(s)$, is a mapping of states to actions that describes how an agent should behave, given its perceived state. Ideally, however, we would like to create agents that can learn how to accomplish more general tasks in a range of environments. One approach to enable this kind of *learning transfer* is to provide the agent with a set of high-level actions that guide the agent to useful parts of the state space. The options framework [Sutton *et al.*, 1999] provides a way to formulate high-level actions in a Markov decision process (MDP) that can be easily incorporated into a variety of different RL algorithms. An *option* describes a sequence of actions that an agent can take to achieve a subgoal in an MDP domain. For instance, three options in a building navigation domain could describe the actions needed to move to a door, unlock it, and open it, respectively. In the original options work, each option was provided to the agent by a domain expert. Later work has focused on agents autonomously discovering useful options from a set of previously learned policies, then using these options to accelerate learning to accomplish future goals (see Section 6). However, much of the existing work either assumes that all policies use identical state spaces, or requires manual mapping across different state spaces. These limitations restrict the applicability of the learned options.

We introduce a framework called *Portable Option Discovery* (POD) that provides the first techniques for fully automated option discovery and learning transfer across related domains with different state spaces, modeled as object-oriented Markov decision processes (OO-MDPs) [Diuk *et al.*, 2008]. We use the term *domain* to refer to an environment described by an OO-MDP, which is composed of a set of objects with assigned attributes, a transition model, start state(s), goal state(s), and an associated reward function. OO-MDPs provide a structured representation that enables state space abstraction by focusing on the objects and attributes that are relevant for a specific task. POD performs learning transfer by identifying options that achieve subgoals in source domains, scoring candidate mappings for abstracting the source options, and grounding the abstracted options in new, target domains. POD is a general framework that can be applied to a range of existing option transfer algorithms.

We implement POD on two existing methods: PolicyBlocks [Pickett and Barto, 2002] and Transfer Options (TOPs) [MacGlashan, 2013]. PolicyBlocks is an automated method for identifying options in non-structured state spaces by finding common behaviors among different policies. By contrast, TOPs enable options to work in object-oriented state spaces, but requires an expert to specify a small set of useful source policies from which TOPs will be created. POD addresses the limitations of both of these methods by providing novel techniques for abstracting state spaces and scoring different possible mappings, enhancing PolicyBlocks to operate over object-oriented state spaces, and replacing the hand-crafted expert mappings required for TOPs with efficient, automatically generated ones. We refer to the new algorithms, respectively, as Portable PolicyBlocks (PPB) and Portable Transfer Options (PTOPs). We evaluate these methods experimentally in two application domains (Taxi World and Block Dude), showing that the POD-enhanced methods consistently improve performance.

*Department of Computer Science, Brown University.

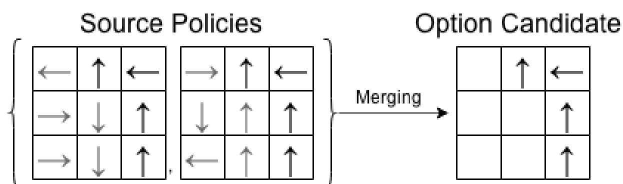


Figure 1: Conceptual diagram of PolicyBlocks' merge.

The main contributions of our work are (1) a process for abstracting an object-oriented domain and then grounding these abstractions to a target domain, (2) the scoring process for selecting the best mapping for these abstractions, (3) the application of these techniques to create the PPB and PTOPs learning transfer methods, (4) experimental results showing that the learning transfer methods improve performance significantly over a non-transfer baseline, and (5) an open-source implementation of POD.

2 Background

MDPs and Policies A Markov decision process (MDP) is a stochastic model that describes the effects of an agent's actions on its environment. An MDP can be represented as a four-tuple, $\{\mathcal{S}, \mathcal{A}, \mathcal{P}(\cdot,\cdot), \mathcal{R}(\cdot,\cdot)\}$, consisting respectively of a set of states, a set of actions, a transition probability function, and a reward function. An agent makes decisions in an MDP environment by using a *policy* that provides a mapping from states to actions. In general, an agent's policy can be deterministic or stochastic. In our work, we make the simplifying assumption that a policy is a deterministic mapping from states to individual best actions; however, the methods could readily be applied in the context of probabilistic policies, as we will discuss. Learning an optimal policy is often achieved through *value iteration* methods to compute an action-value function $Q : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$. The Q-value $Q^{\pi}(s, a)$ represents the expected return (cumulative reward) of taking action a in state s under policy π.

Options An *option* for an MDP is a conditional sequence of primitive actions defined as a three-tuple, $\{\pi, \mathcal{I}, \beta\}$, consisting of a policy ($\pi : \mathcal{S} \rightarrow \mathcal{A}$), a set of initiation states ($\mathcal{I} \subseteq \mathcal{S}$), and termination conditions ($\beta : \mathcal{S} \rightarrow [0, 1]$) [Sutton *et al.*, 1999]. The initiation set $\mathcal{I}$ is the subset of states in which the option can begin to be executed. If an option's initiation condition is satisfied and the agent selects it, the option's policy is followed until a termination condition is met. The termination condition β is a probability function over states that defines the likelihood that the agent ends the execution of an option when it is in that state. Options are typically used as high-level actions that achieve subgoals in MDPs, terminating deterministically when a goal state is reached. The states in an option's initiation set do not need to be contiguous (e.g., a door-opening option might only apply locally in the regions immediately adjacent to the doors in a building).

PolicyBlocks The PolicyBlocks option discovery method [Pickett and Barto, 2002] finds options from a set of previously solved source MDPs for a set of target MDPs that share the same tabular (unstructured) state space. PolicyBlocks creates options by identifying common subpolicies in the set of source policies for the source MDPs. The source policies may have been hand-crafted by a designer or learned by the agent with a reinforcement learning algorithm like Q-learning [Watkins and Dayan, 1992]. Algorithm 1 shows the pseudocode for the main operation of PolicyBlocks.

Algorithm 1 PolicyBlocks Power Merge

```
P ← set of source policies
O ← empty option set
c ← null
while |P| > 0 do
    for all M ← subsets(P) do
        M′ ← merge(M)
        if score(M′) > score(c) then
            c ← M′
    subtract(P, c)
    O ← O + c
return O
```

The PolicyBlocks algorithm first generates a set of *option candidates* by merging subsets of the initial policy set P. In principle, all possible subset merges could be created; in practice, Pickett and Barto claim that merging all possible pair and triplet subsets is sufficient to obtain good option candidates. A *merge* is defined as the collection of state-action pairs that exist in each of a set of source policies. The result of each merge is a *partial policy*, a policy that maps a subset of the full state space to actions and maps the remaining states to a special "empty" action. A visualization of the merging procedure given two source policies is shown in Figure 1. An inherent limitation of PolicyBlocks (and motivation for POD) is that the merge operator cannot be applied to source policies from domains with different state spaces.

Each option candidate is assigned a *score*, defined as the product of the number of states defined in the candidate and the number of policies in the original source policy set that contain the same state-action pairs. All policies are considered when calculating the score, even those that were not source policies for the merge. The highest-scoring option is placed in an *option set* O, and the states associated with that option are *subtracted*, or removed, from each source policy that maps that state to the same action as the new option.

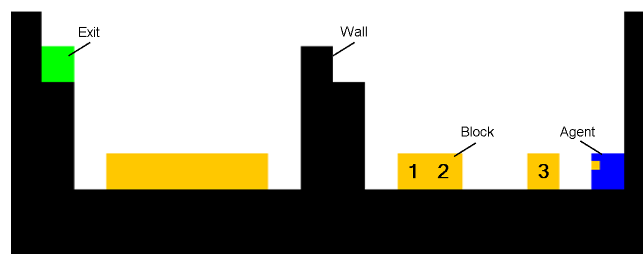


Figure 2: A Block Dude domain with labeled objects.

Object-Oriented MDPs Diuk *et al.* [2008] introduced object-oriented Markov decision processes (OO-MDPs) to facilitate learning in environments with large state spaces. OO-MDPs represent environments by a set of n objects, $\mathcal{O} = \{o_1, \ldots, o_n\}$, where every object belongs to an *object class* and has a value assignment to a set of attributes that are associated with the object class. We will refer to n as the OO-MDP's *domain size*. The *status* of an object at any given time is the current set of values for each attribute of its class. Likewise, state s of an environment is defined as the union of the attribute values of all objects in the domain, $s = \cup_{i=1}^{\mathcal{O}} \text{status}(o_i)$.

Figure 2 shows the Block Dude OO-MDP domain, with object classes *wall*, *block*, *agent*, and *exit*. Each class has *x* and *y* position attributes; the agent class has an orientation attribute (left or right); and the block class has a "being carried" attribute. The agent can move left, move right, pick up an adjacent block in the direction it is facing (if the cell above the block is empty), drop a block in the direction it is facing (if that cell is empty), or climb up a single block or wall (if the cell over and up one in the direction it is facing is empty). Any agent or moved block always falls to the lowest unoccupied cell. The agent must stack blocks in a configuration that grants access to the exit, its goal. Figure 2 shows three numbered block objects. Note that if the position attributes of these objects were swapped so that each object was now in the location previously occupied by another of the same class, the state would be functionally identical. This property facilitates an agent's ability to generalize across a variety of arrangements, given a set of interchangeable objects.

The object-oriented representation of an environment simplifies the representation of policies and transition models by permitting abstraction over states that are identical for the purposes of an agent's action choice. If state spaces from two different RL domains share a common set of objects, then knowledge can potentially be transferred between these domains.

Transfer Options Transfer Options (TOPs) [MacGlashan, 2013] are designed to be transferred from one or more source domains to a target domain with a different state space. Whereas traditional options are designed to represent local subgoals and leverage often-visited states, TOPs are used to transfer contiguous state-action pairs from a source policy. Rather than identifying noteworthy portions of previous policies, TOPs apply an entire source policy to a new domain through a user-supplied mapping. The TOP creation method takes a state space mapping (from the source domain to the target domain), an initiation set, and the termination condition as input. If multiple mappings exist, each mapping will produce a unique TOP. The primary limitation of TOPs is that the approach does not provide guidance about which policies are most appropriate for transfer, or which state space mappings are most likely to be useful.

3 Approach

The POD framework encompasses several procedures that allow existing option transfer methods to perform automated

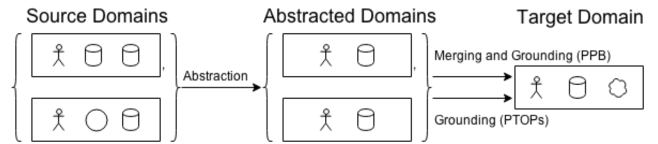


Figure 3: Conceptual diagram of abstraction and grounding.

option discovery in richer, more complex domains. Applied to PolicyBlocks, POD enables learning transfer across OO-MDPs; applied to TOPs, POD provides a means of identifying the most useful source policies and mappings from a large set of source policies and possible mappings.

We define an *application* as a set of OO-MDP object classes, their associated attributes, and a set of actions. A domain, as defined previously, is a specific OO-MDP environment that is an instance of an application. Domains that belong to the same application differ in the number of objects for each object class in the domain's state space. A key challenge in learning transfer across such domains is to identify the best mapping between corresponding objects. POD accomplishes this mapping through an *abstraction* process that uses novel *scoring* and *mapping search* techniques. Integrating POD into PolicyBlocks also requires modifying the *subtraction* and *merging* procedures of PolicyBlocks.

POD learns abstract options from a set of source domains in which policies have been provided or learned. These abstract options can be applied to many different target domains of the same application to help learn new policies in these domains as efficiently as possible. Figure 3 shows a high-level diagram of this process, where each domain is abstracted to the same level.

Abstraction To transfer knowledge between domains, we abstract policies from previously learned (source) domains so they can be used in new (target) domains that share some commonalities. Consider policy π in source domain D, which provides a mapping from the state space S to the action set A. To abstract π, we create an abstract domain D' in abstract state space S'. The abstract state space is defined as a subset of the OO-MDP objects present in the source domain; therefore, $S' \subseteq S$. In order to provide a mapping between S and S' when there are multiple objects of the same type in S, we must associate each object in S with an object of the same type in S'. This correspondence will induce a mapping function $f : S \rightarrow S'$.

Note that some objects in S will not correspond to any object in S' – they are effectively irrelevant to the abstracted policy. The process of determining the relevant objects in S and their roles in the abstract domain is at the heart of POD's abstraction process and the ensuing learning transfer.

For any source domain that contains multiple objects of any class c_i, there will be multiple possible mappings to the target domain. Specifically, given k_i objects of class c_i in the source domain and k'_i objects of that class in the abstract domain, there are $k_i!\binom{k_i}{k'_i}$ possible mappings. If there are $|C'|$ object classes in the abstract domain, then the number of possible

mappings is the product of these terms for each object class:

$$\mathcal{M} = \prod_{i=1}^{|C'|} k_i \binom{k_i}{k'_i}. \tag{1}$$

Clearly, for complex domains, exhaustive consideration of all $\mathcal{M}$ mappings is infeasible. Therefore, we select the best mapping by first defining a scoring metric (see "Scoring") and then applying a heuristic search method to discover a high-scoring mapping (see "Mapping Search").

Since f is necessarily many-to-one, with this mapping, there will exist a one-to-many inverse mapping from the abstracted domain to the source domain, $f^{-1} : S' \rightarrow \mathcal{P}(S)$ (where $\mathcal{P}$ denotes the power set). Once the mapping is determined, an abstract policy for S' can be created by associating each abstract state $s' \in S'$ with an action from the source domain $a \in A$, using the function $f^{-1}(s')$ to identify the set of source states that map to the abstract state s'. Because the source policy may recommend different actions for different source states in $f^{-1}(s')$, a consensus action must be computed. In our experimental setup, the learned policies are all derived from Q-values, so we define this consensus function by taking the action with the highest average Q-value among the source states $f^{-1}(s')$:

$$\pi'(s') = \arg\max_{a \in A} \frac{1}{|f^{-1}(s')|} \sum_{s \in f^{-1}(s')} Q(a, s) \tag{2}$$

Note that if the source policy indicates that the same action is optimal for all of the source states (i.e., when $\forall s_1, s_2 \in f^{-1}(s'), \pi(s_1) = \pi(s_2)$), then Equation 2 will select that action as the optimal action for the abstract state. Our method can easily be applied to other policy representations (including stochastic policies) by defining a different consensus function (e.g., using a voting method to select the most frequently recommended action).

Scoring If there are many possible source options and mappings, providing the agent with an option for each possible option/mapping combination may counteract the benefits of knowledge transfer, due to an increase in the branching factor of the action space. Therefore, we introduce a scoring system for ranking the amount of knowledge transfer that each mapping could provide, and we select only the best mapping for use. The score of a mapping is based on the proportion of the state-action pairs that are preserved between the source policy and the abstract policy.

Given an input source domain D and output abstract domain D', consider π, a partial policy for the input domain, f, a mapping from the input state space to the output state space, and π', the abstract policy (Equation 2). We can then use f to reconstruct a policy π'' in the input domain that takes the action associated with each corresponding state in the output domain:

$$\pi''(s) = \pi'(f(s)).$$

(Essentially, π'' is the grounded version of the abstract policy in the original source domain.) Due to the information loss from abstraction, there will be differences between π and π''. Minimizing this information loss means maximizing the number of actions that π and π'' recommend in common. Therefore, we wish to maximize the ratio of the number of common actions in π and π'' to the total number of state-action pairs in π:

$$Score = \frac{|\pi \cap \pi''|}{|\pi|}. \tag{3}$$

Algorithm 2 PPB Power Merge

```
n ← number of options
P ← set of source policies
O ← empty option set
c, c' ← null
while |P| > 0 and |O| < n do
    for all M ← subsets(P) do
        g ← gcg(M)
        M' ← merge(abstract(g, M))
        M'' ← ground(M')
        if score(M'') > score(c') then
            c ← M'
            c' ← M''
    subtract(P, c)
    O ← O + c
return O
```

Mapping Search Greedy search is used to avoid scoring all $\mathcal{M}$ mappings by identifying a subset of relevant possible mappings. To find a useful mapping from a source domain to a target domain, first the source domain is abstracted relative to the *greatest common generalization (GCG)*, defined as the maximal subset of objects in each class that appear in both the source domain(s) and the target domain. In other words, given a set of objects in the source domain, $\mathcal{O}_s$, and a set of objects in the target domain, $\mathcal{O}_t$, the GCG is defined as $\mathcal{O}_s \cap \mathcal{O}_t$, regardless of the values of the attributes associated with the objects. The GCG is computed only once per policy to be transferred.

Once the GCG has been computed, an abstract domain representation is produced from the source domain by selecting one of the $\mathcal{M}$ possible mappings (Equation 1). We use a stepwise greedy search method, which performs this abstraction in b steps, where b is the number of objects in the source domain that are not in the GCG. We denote the source domain as D'_b and the final abstracted domain as D'_0. At each step, the least relevant object is eliminated. That is, for step m, D'_{m-1} is obtained by using the highest-scoring mapping in the set of all mappings from D'_m to a domain with one fewer object.

Applying the Transferred Policy An abstract policy can be used in a target state space by *grounding* it to that domain. This grounding step requires the same type of mapping as was necessary for the source-to-abstract step, and is done using the same mapping scoring method (Equation 3). Again, the score is based on the number of matching state-action pairs between the abstract policy and the grounded (target) policy.

3.1 Portable PolicyBlocks

Portable PolicyBlocks (PPB) creates its option candidates from OO-MDP policies, allowing it to work in richer and more complex domains than PolicyBlocks. PPB uses a modified Power Merge process (Algorithm 2) that creates sets of initial source policies, identifies the GCG for each set, abstracts each policy using the relevant GCG, merges each set, grounds each result, scores each result, and subtracts the highest-scoring option candidate. Thus, PPB follows PolicyBlocks with the addition of the GCG identification, abstraction, and grounding procedures.

We assume that source policies are optimal partial policies for a set of source domains within an application. Source policies are assumed to be defined for all of the states that are reachable from all *initial states* in the applicable source domain. An initial state of a domain is a state in which the agent would begin to accomplish a specific task; therefore, a policy for a specific task or set of tasks includes at least one such state. PPB Power Merge generates a set of option candidates by applying the modified *merge* operator to all pairs and triples of the source policies.[1]

Merging The merge operator accepts any number of source policies and produces an option candidate. As in PolicyBlocks, a merge is simply the collection of state-action pairs shared among source policies. Obtaining a non-empty merge from source policies learned on domains with different sets of objects requires identifying a single abstract domain (i.e., a state space defined by some shared set of objects). Source policies must be mapped to that abstract state space; this process is accomplished by the *abstract* operator. In order to abstract everything to the same GCG, we implement greedy merge, abstracting one step at a time and merging at the earliest possible step. For example, given four source domain policies π_1, π_2, π_3, and π_4 with number of objects k, such that $k_{\pi_1} < k_{\pi_2} < k_{\pi_3} < k_{\pi_4}$, merging proceeds in this order: π_4 is abstracted relative to π_3, π'_4 and π_3 merge to form $\pi_{4,3}$, $\pi_{4,3}$ is abstracted relative to π_2, $\pi'_{4,3}$ and π_2 merge to form $\pi_{4,3,2}$, and so on.

Subtraction Mapping several source states to a single abstracted state also changes the way that the *subtract* operator of the original PolicyBlocks algorithm is applied to the source policies. In PolicyBlocks, subtraction is done by iterating over the state-action pairs in the option candidate, removing all instances of matching state-action pairs from all initial policies, and replacing them with an "undefined" action for those states. Using POD, the source policies are typically in a different domain from the option candidate, but grounding the abstracted option to the domain of each of the source policies (as described previously) allows the use of subtraction as described in PolicyBlocks.

[1]We verified experimentally that increasing sets of merged source policies beyond triples does not improve performance, as was suggested by Pickett and Barto [2002].

Applying Options An option candidate is used by grounding its abstract states to a target domain as the state space of that target domain is explored. When a previously unexplored state in the target domain is found, the set of states in the abstract domain that could map to that target domain state are identified. Then, the mapping for these specific states is applied in the same way as is done for subtraction. This method provides the desired grounding procedure, and avoids performing computations for unreachable states. All states for which the partial policy is defined are treated as initiation states and any state for which the option is undefined is treated as a termination state.

3.2 Portable Transfer Options

Normal TOPs require expert knowledge in the form of a mapping from target states to source states, and source actions to target actions. We implement PTOPs by using POD's mapping and scoring procedures to perform learning transfer across analogous OO-MDPs. When presented with multiple source policies, generic TOPs have no efficient method to determine which options would be best to improve learning in a target domain. Therefore, all option candidates generated using PTOPs are transferred, using the highest-scoring mapping. Applying these option candidates to a target domain is performed in the same way as PPB.

3.3 Complexity

The computational complexity of POD varies depending on which option discovery method is used as its basis, but is dominated by the mapping procedure, which encompasses finding the GCG, abstracting policies, scoring each of the mappings considered, and grounding to the target domain. For this analysis, we use n_{min} to denote the size of the GCG, n_{max} to denote the largest domain size in a set of source and target policies being abstracted, and $\Delta = n_{max} - n_{min}$ to denote the number of objects that need to be eliminated from the largest domain when performing abstraction.

In PTOPs, a mapping is found independently for each source policy, $\pi \in P$, relative to the target domain. Computing the GCG is $O(n_{min})$, since we only need to consider that many objects to construct the GCG. The greedy mapping search for the abstraction step has Δ iterations, and must consider as many as n_{max} objects on each iteration. To evaluate each of these objects requires scoring the resulting mapping, which entails iterating over all $|S_\pi|$ states. Since the GCG term is dominated by mapping search, this process yields the following worst case complexity for PTOPs:

$$O\left(\sum_{\pi \in P} \Delta\, n_{max}\, |S_\pi|\right).$$

In PPB (Algorithm 2), for each π to be added to an option candidate, we find the GCG, abstract, and merge. As with PTOPs, abstraction consists of dropping up to Δ objects, and considering up to n_{max} objects each time (but this time, for each policy in sequence during the merging process). Scoring the mapping at each iteration again entails iterating over the states in the source policy. Therefore, for a single merge step, the complexity is:

$$O(\Delta\, n_{max}\, |S_\pi|).$$

Once the source policies are merged, the option candidate must be scored. Because this candidate score is based on the states in *all* of the initial policies that contain the same state-action pairs as the candidate (not just the merged policies), the option candidate must be grounded to each of the initial policies, with the same complexity as a single merge step.

The highest-scoring option candidate is then subtracted from the initial policies, but this only requires one iteration over the initial policies, and its complexity is therefore dominated by that of the scoring step.

An inherent drawback of PolicyBlocks (and thus PPB) is that it is intractable to use the full power set of source policies to create option candidates. We use the heuristic suggested by the PolicyBlocks description and consider only all pairs and triples. There are $O(|P|^3)$ such combinations. Since each option candidate is created sequentially using the process outlined above, these combinations are re-examined k times, where k is the number of options to be created. Therfore, the overall complexity of PPB is:

$$O\left(k\,|P|^3\sum_{\pi\in P}(\Delta\,n_{max}\,|S_\pi|)\right)$$

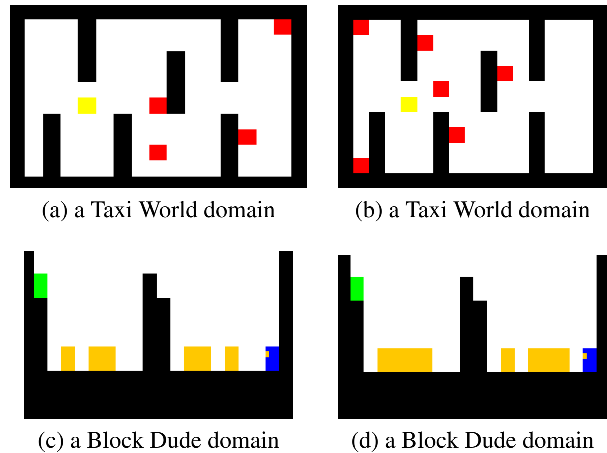
4 Experimental Methodology

4.1 Experimental Settings

We have implemented POD in Java and integrated it into an existing open-source library.[2] We applied POD to two application domains, using several different domain sizes to demonstrate its generalizability. In the experiments reported here, we measure performance using the average cumulative reward over a series of episodes in which the agent is learning the domain. The rate of change of this value reflects the learning speed of the agent.

Two key contributions of our work are (1) the process for abstracting an object-oriented domain that can be grounded to a target domain, and (2) the scoring process for selecting the best mapping for these abstractions. Therefore, our experimental design uses a set of methods that let us separate the effect of abstraction and transfer from the heuristic for selecting the best mapping. PPB and PTOPs are the "POD-enabled" versions of PolicyBlocks and TOPs, as presented earlier. PPB and PTOPs are not directly comparable to their counterparts, since PolicyBlocks will produce empty options when merging policies consisting of different objects and both PolicyBlocks and TOPs require a mapping to apply options in a target of different objects. To accommodate this inability, we also include randomized versions of both methods that use the domain abstractions and transfer, but use random mappings in the abstraction and grounding steps, instead of the highest-scoring mapping. These randomized versions are referred to as RPB and RTOPs. The difference between PPB/RPB and PTOPs/RTOPs gives information about how effective the scoring metric is.

As an upper bound on performance, we include the performance of a "Perfect" policy option, which transfers the option



(a) a Taxi World domain (b) a Taxi World domain

(c) a Block Dude domain (d) a Block Dude domain

Figure 4: Example domains. In Taxi, the agent is yellow; passengers are red; and floors are black. In Block Dude, the agent is blue; blocks are yellow; and the exit is green.

whose policy represents the optimal solution for the target domain (i.e., it is the policy found by Q-learning after convergence in the target domain.) This method, which is basically an "oracle," provides the best possible learning transfer performance that could be provided by an ideal expert. (Note that the agent will still need to learn that it should use the perfect option, since this option is provided alongside the primitive actions for the domain.)

For the transfer methods PPB and RPB, the top-scoring single option candidate is transferred (i.e., $k = 1$). However, in RTOPs/PTOPs, all source policies are transferred, as it would be inefficient to determine the best performing option candidate. Experimentally, the difference between transferring all source policies as options performs only slightly worse than transferring the single best candidate (as determined post-transfer) for the RTOPs/PTOPs methods.

All experiments were performed using intra-option ϵ-greedy Q-learning ($\epsilon = 0.025$) as the base learning algorithm. Intra-option ϵ-greedy Q-learning is a variant of ϵ-greedy Q-learning that, when updating the Q-value for a state following the execution of a primitive action, also updates the Q-value of every other option defined in that state for which the current action of the options are the primitive action taken. The parameter ϵ in intra-option ϵ-greedy Q-learning is the probability of the agent taking a random action. All options used in the experiments have a termination probability of 0.025 in states for which the option is defined. That is, while the agent is following the option's trajectory, there is a 0.025 chance of early termination. Results presented are the average of at least 20 trials, with 20 randomly generated source domains, and one randomly generated target domain for each trial. All experiments were run with pessimistic Q-value initialization, with a uniform cost reward function (-1 for each step taken).[3] All performance differences noted in the text are statistically significant, using a paired Wilcoxon

[2]The source code for our implementation is available at https://github.com/maple-robot-subgoaling/BURLAP.

[3]We calibrated Q-values by running experiments ranging from fully optimistic to fully pessimistic. A nearly pessimistic initializa-

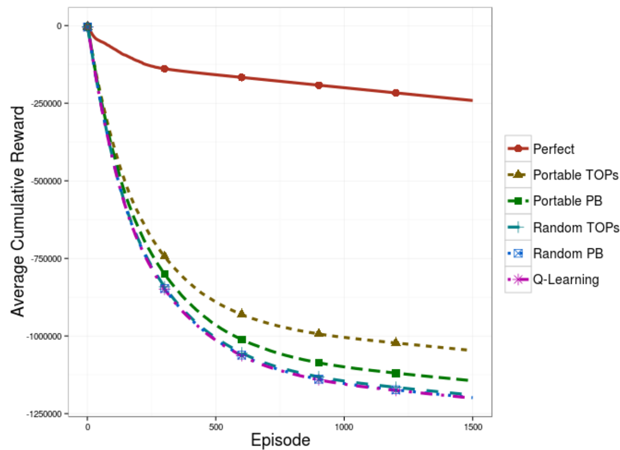


Figure 5: Taxi World Cumulative Reward (6 to 4 passengers)

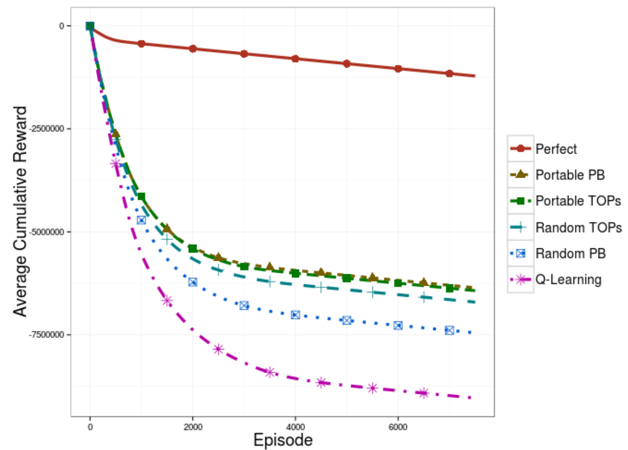


Figure 6: Taxi World Cumulative Reward (4 to 6 passengers)

signed-rank test ($p < 0.05$) after convergence (episode 5000).

4.2 Application Domains

Taxi World The Taxi World application is a grid world consisting of the object classes passenger and agent (taxi driver), with static walls. All objects have x and y position attributes. Passenger and agent classes possess similar attributes, indicating whether a passenger is currently being transported by the agent. The agent's goal is to transport every passenger onto a specified location. The agent can move north, south, east, or west. If the agent is occupying a cell containing a passenger, it may pick up the passenger. If the agent is carrying a passenger, it may drop off the passenger at the current cell. Two examples of Taxi World can be seen in Figure 4.

Block Dude The Block Dude application, based on the game of the same name, is a grid world composed of the following object classes: agent, block, static floors, and exit (goal). The domain is viewed in a classic side-scrolling perspective; examples are visualized in Figure 4. A detailed description was given in Section 2. In our experiments, all trials have a step cap of 1000 per episode before forced termination, preventing the agent from getting permanently stuck.

5 Results

Taxi World Learning transfer in the Taxi World application can be beneficial because the agent is able to reuse knowledge about static elements in the application domain, specifically the location of the walls and goal. Here we show results for learning transfer from a larger domain to a smaller domain (source domain with 6 passengers and target domain with 4 passengers, Figure 5) and from a smaller domain to a larger domain (4 passengers to 6 passengers, Figure 6; 2 passengers to 7 passengers, Figure 7). In the large-to-small transfer, PTOPs perform better than any of the other methods, and in fact, PPB, RPB, RTOPs, and Q-learning are statistically indistinguishable. This result tells us that transferring options directly (as PTOPs do) is more useful than the PolicyBlocks method of discovering new options, and that the mapping heuristic used in PTOPs provides a learning benefit over choosing random mappings. In the small-to-large transfer of 4 passengers to 6 passengers, however, PTOPs, PPB, and RTOPs are all statistically indistinguishable, and all outperform RPB, which in turn outperforms Q-learning. In the small-to-large transfer of 2 passengers to 7 passengers, PTOPs performs best, PPB second best, with RTOPs and RPB indistinguishable from each other, but outperforming Q-learning. Here, the random mappings seem to perform well with respect to Q-learning, even as good as the heuristic mapping for PTOPs in the 4 passenger to 6 passenger transfer, which is interesting and reflects potential symmetry in the domain (i.e., some mappings may be equally good).

Block Dude Learning transfer in Block Dude is useful because the agent is able to reuse knowledge of overcoming obstacles in the application domain, regardless of where the blocks may be initially placed. PTOPs outperform PPB, which in turn outperforms both of the randomized methods. In fact, the randomized methods are statistically indistinguishable from Q-learning. This poor performance of random methods occurs because, in the Block Dude domain, many of the possible mappings do not facilitate meaningful reuse of the options. PTOPs perform better than PPB in this domain, because there are some instances where noise in the source policies will cause loss of information during the merge process in PPB, whereas this information is retained in PTOPs.

The POD-enhanced learning transfer methods (PPB and PTOPs) clearly outperform non-transfer learning, demonstrating the effectiveness of our methods for abstraction and mapping across object-oriented source and target domains. Although PPB sometimes reaches the learning performance of PTOPs, it never outperformed PTOPs in our experiments. PPB does, however, have the advantage that it can identify common subtasks across multiple source domains, which may be useful for larger or more heterogeneous domains. Therefore, PPB does warrant further exploration.

The heuristic mapping search provides some benefit in cer-

tion was ideal in each case, including the Q-learning baseline.

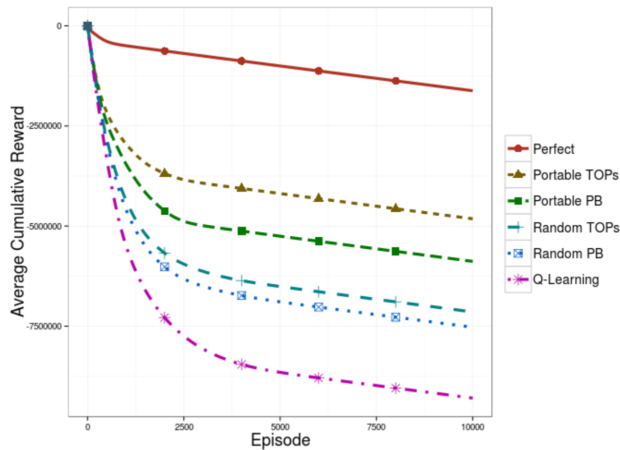


Figure 7: Taxi World Cumulative Reward (2 to 7 passengers)

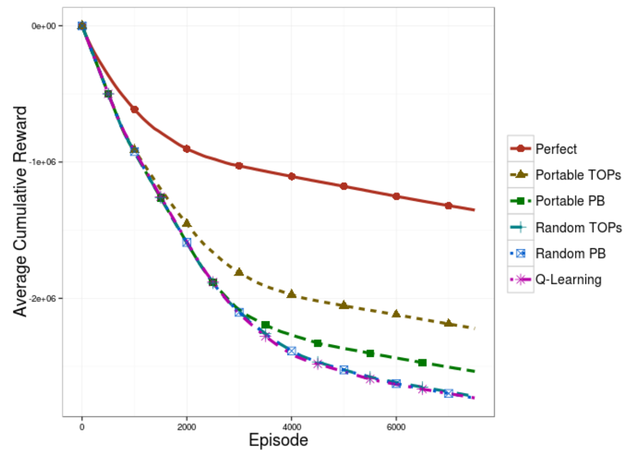


Figure 8: Block Dude Cumulative Reward

tain cases, but not others. Although it never hurts performance (and therefore there is no reason *not* to use the heuristic scoring method), it seems likely that the current scoring method does not always find the best mapping. Therefore, more exploration of alternative scoring metrics is warranted.

6 Related Work

Prior work in option discovery typically utilizes cuts and clustering in the state space to discover policies that lead through critical paths. Local graph partitioning [Şimşek *et al.*, 2005], clustering [Mannor *et al.*, 2004], bottleneck state discovery [McGovern and Barto, 2001; Menache *et al.*, 2002], and betweenness centrality [Simsek and Barto, 2007] all provide graph-theoretic approaches to generating options autonomously. However, each of these methods requires direct analysis of the state space, unlike POD, meaning they produce policies that are not transferable to domains of differing state spaces or reward functions.

Several authors have attempted to create options with subgoal discovery [Şimşek and Barto, 2004] or landmarks [Butz *et al.*, 2004]. These methods may allow for transfer if two domains share subgoals. However, in domains where there is no clear subgoal or landmark, it is preferable to find commonalities with merging instead. Stolle and Precup [2002] find interesting states (which can be used as subgoals) in standard domains with several different heuristics, and automatically construct options for reaching those states. Brunskill and Li [2014] provide the first theoretical analysis of sample complexity for option discovery in MDPs, and propose a PAC-inspired algorithm capable of working in analogous domains with a performance approaching that of handcrafted options.

Other work considers reframing the domain into a new state space that is simpler to transfer [Konidaris and Barto, 2007; Konidaris *et al.*, 2012]. These methods perform autonomous option generation by learning in an *agent-space* representation of the domain, where agent spaces use deictic features that are defined relative to the agent's position or state (allowing the agent to refer to "the nearest wall" or "the key I am holding"). Agent-space learning allows the agent to develop skills that can be easily transferred, but require special design considerations and more additional learning time than traditional option learning. Therefore, while agent spaces may be a beneficial addition to learning, they do not solve the problem of discovering new options.

7 Future Work

As mentioned earlier, further exploration of heuristic scoring methods, and studying these methods in larger, more heterogeneous domains are promising avenues of future work. Another major improvement to POD would be to expand beyond tabular learning. Techniques such as Q-learning and SARSA require a table of state-action pairs mapped to rewards, which is a significant limitation for domains that are extremely large or continuous. In many of these cases, value function approximation (VFA) can be used, allowing the agent to estimate a state's value from similar states. Extending POD to support domains that require VFA would also entail modifying the abstraction and scoring steps to use a sampling-based approach, rather than iterating over the entire state space. These extensions would provide increased scalability for complex applications.

8 Conclusion

We have introduced the POD framework to perform automated option discovery in OO-MDP domains. We extended two existing algorithms to create the Portable PolicyBlocks (PPB) and Portable Transfer Options (PTOPs) methods. The source code for our implementation of POD is available online as part of an open source library. We have demonstrated that POD's heuristic mapping selection permits these methods to be applied automatically in object-oriented domains, significantly outperforming standard Q-learning, and outperforming random mapping selection in most cases. These methods represent the first fully automated techniques for learning transfer in object-oriented domains and offer promising avenues for future improvement.

References

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.

Martin V. Butz, Samarth Swarup, and David E. Goldberg. Effective online detection of task-independent landmarks. *Online Proceedings for the ICML'04 Workshop on Predictive Representations of World Knowledge*, 2004.

Carlos Diuk, Andre Cohen, and Michael L. Littman. An object-oriented representation for efficient reinforcement learning. In *Proceedings of the 25th International Conference on Machine Learning*, pages 240–247. ACM, 2008.

George Konidaris and Andrew G. Barto. Building portable options: Skill transfer in reinforcement learning. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence*, pages 895–900, 2007.

George Konidaris, I. Scheidwasser, and A.G. Barto. Transfer in reinforcement learning via shared features. In *Journal of Machine Learning Research*, pages 1333–1371, 2012.

James MacGlashan. *Multi-Source Option-Based Policy Transfer*. PhD thesis, University of Maryland, Baltimore County, 2013.

Shie Mannor, Ishai Menache, Amit Hoze, and Uri Klein. Dynamic abstraction in reinforcement learning via clustering. In *Proceedings of the Twenty-First International Conference on Machine Learning*, 2004.

Amy McGovern and Andrew G. Barto. Automatic discovery of subgoals in reinforcement learning using diverse density. In *Proceedings of the 18th International Conference on Machine Learning*, pages 361–368, 2001.

Ishai Menache, Shie Mannor, and Nahum Shimkin. Q-cut: Dynamic discovery of sub-goals in reinforcement learning. In *Proceedings of the 13th European Conference on Machine Learning*, pages 295–306. Springer, 2002.

Marc Pickett and Andrew G. Barto. PolicyBlocks: An algorithm for creating useful macro-actions in reinforcement learning. In *Proceedings of the 19th International Conference on Machine Learning*, pages 506–513, 2002.

Özgür Şimşek and Andrew G. Barto. Using relative novelty to identify useful temporal abstractions in reinforcement learning. In *Proceedings of the 21st International Conference on Machine Learning*, pages 751–758, 2004.

Ozgür Simsek and A. Barto. Betweenness centrality as a basis for forming skills. Technical Report TR-2007-26, University of Massachusetts Department of Computer Science, 2007.

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.

Martin Stolle and Doina Precup. Learning options in reinforcement learning. In *Proceedings of the 5th International Symposium on Abstraction, Reformulation and Approximation*, pages 212–223, London, UK, UK, 2002. Springer-Verlag.

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.

C.J.C.H. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8(3):279–292, 1992.