Discovering Subgoals in Complex Domains

Marie desJardins, Tenji Tembo, Nicholay Topin, Michael Bishoff, Shawn Squire, James MacGlashan*, Rose Carignan, and Nicholas Haltmeyer
Department of Computer Science and Electrical Engineering
University of Maryland, Baltimore County
Baltimore MD 21250
{mariedj,tenji.tembo,ntopin1,bishoff1,ssquire1}@umbc.edu, jmacglashan@cs.brown.edu, {rcarign1,hanicho1}@umbc.edu

Abstract

We present ongoing research to develop novel option discovery methods for complex domains that are represented as Object-Oriented Markov Decision Processes (OO-MDPs) (Diuk, Cohen, and Littman, 2008). We describe Portable Multi-policy Option Discovery for Automated Learning (P-MODAL), an initial framework that extends Pickett and Barto's (2002) PolicyBlocks approach to OO-MDPs. We also discuss future work that will use additional representations and techniques to handle scalability and learning challenges.

We present our ongoing and planned work on Portable Multi-policy Option Discovery for Automated Learning (P-MODAL), a framework for finding options that can be used to identify useful subgoals in Object-Oriented Markov Decision Process (OO-MDP) domains (Diuk, Cohen, and Littman, 2008).

An *option* (Sutton, Precup, and Singh, 1999) is an extended action that enables an agent to achieve a subgoal. Options are useful for structuring complex planning spaces and can accelerate both planning and learning. Much of the work on options requires that the initiation and termination conditions be manually identified, but there has also been a limited amount of research on *option discovery*—i.e., automatically identifying potentially useful options in a given domain.

Our work focuses on the creation of novel option discovery methods that can be applied in OO-MDP domains, enabling greater scalability and applicability than the existing work on option discovery.

PolicyBlocks (Pickett and Barto, 2002) can be used to find options by identifying and merging similar policies (i.e., solutions to related tasks (goals) that select the same actions in the same set of states). Agent spaces (Konidaris and Barto, 2007) can be used to learn policies for hand-specified options using a local feature space, defined relative to the agent. These policies can be transferred to new environments because of the agent-centered relational representation of the features. Stolle and Precup (2002) find interesting states—which are used as subgoals—using several different heuristics, and automatically construct options for reaching those states.

Our interest is in discovering options automatically in rich domains that can be represented using OO-MDPs, which describe state spaces as a set of objects and associated attributes. The ongoing work described here represents a first step in that direction, extending PolicyBlocks to be applicable to OO-MDP domains by introducing novel techniques for abstracting state spaces from policies as they are merged, and methods for sampling from the space of possible domains for the OO-MDPs being merged. In future work, we plan to further extend these techniques by exploring heuristic search over an agent-space predicate language to augment the space of possible options that can be discovered, and by incorporating landmark methods such as those developed by Stolle and Precup.

The remainder of this paper provides some background on options, PolicyBlocks, and OO-MDPs; describes our approach for extending PolicyBlocks to OO-MDP domains through state space abstraction and mapping; and summarizes our empirical methodology for testing our new methods. (These tests are currently underway, and we will present our results at the symposium.)

Background

Options. An *option* is a temporally extended macro-action, which is a conditional sequence of primitive actions (Sutton, Precup, and Singh, 1999). Options are represented as a triple consisting of a policy, a set of initiation states, and a set of probabilistic termination conditions. For an option to be initiated, an *initiation condition* must be met. Once initiated, the option's policy executes according to the policy, a probabilistic mapping from states to actions. This continues until a *termination condition* is satisfied. Options are used in MDP learning and planning to aid in searching the decision space, and possibly represent subgoals.

PolicyBlocks. PolicyBlocks is an algorithm for creating options by identifying common subpolicies in a set of initial policies. These initial policies may have been hand-crafted by a designer or learned from observation. The PolicyBlocks algorithm first generates a set of *option candidates* by merg-

*James MacGlashan: Department of Computer Science, Brown University, Providence RI

ing subsets of the policies in the initial set. (In principle, all possible subset merges could be created; in practice, Pickett and Barto claim that creating all possible pair and triplet sets is sufficient to obtain good policies.) A *merge* is defined as the subset of states that are mapped to the same action in all of the source policies. Once an option candidate set has been created, each candidate is assigned a score, defined as the product of the number of states it contains and the number of policies in the initial set that contain those states. (Note that the latter policies need not all have been source policies for the merge. States that appear in non-merged initial policies with the same action mapping are also counted in the score.) The highest-scoring option is placed in an *option set*, and all of its states are removed from the initial policies that contain the same action for that state.

Object-Oriented MDPs. Diuk, Cohen, and Littman (2008) introduced Object Oriented Markov Decision Processes (OO-MDPs) to facilitate learning in environments with large state spaces. Extremely large state spaces result in an exponential increase in the total number of states as a function of the number of state variables. OO-MDPs represent these environments by describing a set of objects and their attributes, simplifying the representation of transition models and policies by permitting abstraction over states that behave identically. The state of the environment is defined to be the union of the states of all of the objects in the domain. If the sets of objects in two different domains are the same (or overlapping), then knowledge transfer methods can potentially be used to transfer knowledge from one domain to another (MacGlashan, 2013).

Approach

We are developing the P-MODAL framework to apply a PolicyBlocks option discovery approach to OO-MDP domains. *Union Merge*, the top-level function for the option creation process, merges subsets of OO-MDP policies to create candidate options that are then scored. Union Merge uses several subprocesses, as explained in the remainder of this section.

In this discussion, we use the term *domain* or *task* to refer to an OO-MDP that is characterized by a set of objects, a transition model, start state(s), goal state(s) and an associated reward function. An *application* or *domain space* refers to a set of related domains that share common objects and a probability distribution of domains within this application. Our option discovery is designed to be applied within a given application, where policies have been provided or learned for some of the domains in the distribution (the *source domains*) and the goal is to learn policies for new domains (the *target domains*) as efficiently as possible.

Union Merge Similar to merging in PolicyBlocks, P-MODAL uses a "union merge" cycle to combine multiple source policies. The source policies are assumed to be optimal *partial policies* for a set of source domains within an application. A partial policy, as defined in PolicyBlocks, is a policy that is not defined in all states; the source policies are assumed to be defined only for the states that are reachable from some initial state in the applicable source domain. The power set of the source policy combinations is generated by identifying each combination of source policies resulting in 2^n combinations, where n is the number of source policies. The *Merge* operator is then applied to each combination of source policies, resulting in a set of *option candidates*. The option candidates are subsequently scored using a *Score* operator, and the highest-scoring option candidate is added to the option set. The *Subtract* operator is then applied to each of the source policies, removing the states associated with the new option, which prevents the option set from including multiple instances of the same state-action pair. The process then repeats (using the modified source policies) until a stopping condition is reached: either the desired number of options (a parameter of the algorithm) have been created, or all states in the source policies have been included in some option.

Merge The Merge operator accepts any number of source policies and produces an option candidate. Similar to PolicyBlocks, a merge is defined simply as the collection of state-action pairs that are shared between the source policies. To obtain a non-empty merge from source policies that were learned in different domains, however, a single abstract domain (i.e., a state space defined by some shared set of objects) must be identified, and both of the source policies must be mapped to that target state space. This process is accomplished by the *Abstract* operator. A merge of three or more policies is done by repeated pairwise merging (e.g., merge(p1, p2, p3) = merge(merge(p1, p2), p3)) after all of the source policies have been abstracted to the same target state space.

Abstract When merging a set of policies, a common state space is first identified by finding the Greatest Common Generalization (GCG) of the source policies' state spaces. The GCG is the intersection of the domain objects—that is, it is the largest subset of objects that appear in all of the source policies.

Once the GCG has been computed, an abstract policy is produced from a source policy by eliminating each object not in the GCG from the state descriptors of each state. When this elimination is done, multiple source states will collapse into a single state in the target domain. The Q-value for the target state is computed by averaging the Q-values of these source states. If multiple objects of the same type t_i appear in the target domain, then there may be multiple possible abstracted policies–specifically, there are k_i choose m_i possible abstracted policies, given k_i objects of a given type in the source domain and m_i objects of that type in the target domain. If there are T object types in the target domain, then the number of possible mappings is the product of these

Note that for large source policy sets, using the power set is infeasible. We leave the problem of searching for a "good" set of source domains to future work.

terms for each object type:

$$\mathcal{M} = \prod_{1}^{T} \binom{k_i}{m_i} \tag{1}$$

Each of the $\mathcal{M}$ possible abstractions for a given merge is generated and evaluated by *grounding* the abstract policy and comparing this grounded policy to the source policy. An abstract policy is grounded by identifying each state from the source policy that would map into each target state in the abstraction. When a single grounded state could be an origin of multiple abstract states, one of the abstract states is chosen at random and it is assumed that the grounded state is the origin of only this single abstract state. The abstract policy with the greatest number of state-action pairs that match is used as the abstraction for the given source policy.

Score Since an option candidate in P-MODAL is not necessarily represented in the same domain (state space) as the source policy or target domain, absolute policy size is not a good heuristic, so we instead use a policy ratio. Specifically, P-MODAL's scoring method grounds the option candidate to the domain of each source policy used in the merge. The score associated with each source policy is the percentage of the source policy's state-action pairs that match (recommend the same action as) the target policy. These source policy scores are summed to create an option candidate score. This process results in similar behaviour to PolicyBlocks scoring: abstract policies created from a large number of source policies will have inherently higher scores, as will abstract policies that represent more of the state space for source policies.

Subtract Mapping several source states to a single set of abstracted states 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 each state-action pair 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 P-MODAL, the source policies are typically in a different domain than 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.

Applying Options The highest-scoring option candidate is used as an option by grounding its abstract states into a target domain as the state space of that target domain is explored. Rather than grounding the entire option in the target state space, as a state is encountered, it is grounded in the same way that is done for subtraction. All states for which the partial policy is defined are treated as initiation states, and option execution continues until a state is reached for which the option is not defined; this state is treated as a termination state.

As with the option search, for large domains, this process could be done through heuristic search rather than exhaustive search.

Experimental Methodology

We have implemented four applications to use for experimental evaluation of our methods: Four Rooms, Taxi, Blocks World, and Sokoban. The Four Rooms application is a simple grid world that includes one agent, one goal state, and multiple walls that define the outline of the four rooms. The agent begins in one room in its start state, and is able to move either north, south, east, or west. The agent may not move through the walls, or outside of the bounds of the four rooms on its path to finding the goal state.

The Blocks World application consists of a flat surface and multiple blocks. Blocks may be placed on top of each other or on the surface, but only one block can be moved at a time. For this reason, blocks that are under other blocks cannot be moved until the upper blocks have been relocated. Blocks can have various attributes, permitting abstraction and state grouping.

In the Taxi application, one taxi must find a specified number of passengers and deliver them to the goal state. Initially, the taxi begins in the goal state and has no passengers. The taxi is able to traverse the domain by traveling north, south, east, or west. Once the taxi finds a passenger, it must deliver the passenger back to the goal state and repeat the same process with the rest of the passengers.

Sokoban is modeled after a Japanese puzzle game in which the player's goal is to achieve a specified configuration of items in the minimum number of moves. This application consists of four types of objects: the agent, the rooms, the blocks, and the doorways, which function as connections between the rooms in the domain. An example of a Sokoban goal is for the agent to push a specific block (defined by a color, type, or relative location) into a designated room.

We are developing a set of experiments that will create options from source domains with varying degrees of overlap. We will measure the degree to which our option discovery methods accelerate learning and improve planning performance in more complex instances of the applications.

Results

We are currently in the process of gathering data. Our initial results are promising and show that P-MODAL has a significant improvement over standard Q-learning and PolicyBlocks.

Related Work

Prior work in autonomously building generalized options requires utilizing cuts and clustering in the state space to discover policies that lead through critical paths. Local graph partitioning (Şimşek, Wolfe, and Barto, 2005), clustering (Mannor et al., 2004), bottleneck state discovery (McGovern and Barto, 2001; Menache, Mannor, and Shimkin, 2002), and betweenness centrality (Simsek and Barto, 2007) all provide a graph-theoretic approach to autonomously generating options. However, since all of these methods require direct analysis of the state space, they produce policies that are not transferrable to domains that have differing state spaces or reward functions.

Several authors have attempted to create options with subgoal discovery (Şimşek and Barto, 2004) or landmarks (Butz, Swarup, and Goldberg, 2004). These methods may allow for transfer if two domains share subgoals. In domains where there is no clear subgoal or landmark, however, it is preferable to find commonalities with merging instead.

Other work considers reframing the domain into a new state space that is simpler to transfer (Konidaris and Barto, 2007; Konidaris, Scheidwasser, and Barto, 2012). These methods perform autonomous option generation by learning in an agent-space representation of the domain. 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 are not directly useful for discovering new options.

Future Work and Conclusions

Our work currently focuses on extending the PolicyBlocks method to OO-MDP domains, but we are also exploring more powerful relational representations (including agent-spaces) and search methods for identifying useful options in complex environments. Two major challenges are scalability and applying these methods in a learning context (i.e., in domains for which the agent does not know the MDP).

To scale the PolicyBlocks-based techniques to larger domains and state spaces will require solving several additional challenges. In such domains, a state table representation of the learned policy is impractical, so we will have to extend the iteration and abstraction methods to be applicable with value function approximation (VFA) methods for policy representation. Iterating over all states will not be feasible, so a sampling approach will need to be used when merging the policies. This sampling process may be guided by domain analysis, focusing on likely states and potentially using heuristic methods to identify local applicability regions for the new options.

When the agent does not know the MDP initially and must learn it through experience, we may not even have a complete ontology of objects and attributes. In this case, we may need to determine whether and how an option can be applied to a completely new state (i.e., one containing objects or attribute values that have not previously been encountered). More powerful abstraction methods may be needed to handle these cases.

Acknowledgements

This work was supported by NSF's Division of Information and Intelligent Systems on awards #1065228 and #1340150 (REU supplement). We would like to thank the other co-PIs on the project, Michael Littman and Smaranda Muresan, for many fruitful discussions.

References

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

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

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

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

MacGlashan, J. 2013. *Multi-Source Option-Based Policy Transfer*. Ph.D. Dissertation, University of Maryland, Baltimore County.

Mannor, S.; Menache, I.; Hoze, A.; and Klein, U. 2004. Dynamic abstraction in reinforcement learning via clustering. In *Proceedings of the Twenty-First International Conference on Machine Learning*.

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

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

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

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

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

Şimşek, Ö.; Wolfe, A. P.; and Barto, A. G. 2005. Identifying useful subgoals in reinforcement learning by local graph partitioning. In *Proceedings of the 22nd International Conference on Machine Learning*, 816–823.

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

Sutton, R. S.; Precup, D.; and Singh, S. 1999. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence* 112(1):181–211.