Hierarchical Skill Learning for High-Level Planning

Keywords: planning, reinforcement learning, abstraction, approximation

James MacGlashan JMAC1@CS.UMBC.EDU
University of Maryland Baltimore County, 1000 Hilltop Circle, Baltimore, MD 21250 USA

Marie desJardins MARIEDJ@CS.UMBC.EDU
University of Maryland Baltimore County, 1000 Hilltop Circle, Baltimore, MD 21250 USA

Abstract

We present *skill bootstrapping*, a proposed new research direction for agent learning and planning that allows an agent to start with low-level primitive actions, and develop skills that can be used for higher-level planning. Skills are developed over the course of solving many different problems in a domain, using reinforcement learning techniques to complement the benefits and disadvantages of heuristic-search planning. We describe the overall architecture of the proposed approach, discuss how it relates to other work, and give motivating examples for why this approach would be successful.

1. Introduction

Many of the existing techniques for controlling goal-directed agent behavior fall into two primary approaches: heuristic-search planning (HSP) and reinforcement learning (RL). Each has its advantages and disadvantages. For instance, heuristic-search planning does not traditionally learn from previous experience, and can only be applied in domains for which a complete domain model exists. On the other hand, reinforcement learning often performs poorly in new situations until it has gained enough experience to learn an effective policy, and it is difficult to scale RL up to large, complex domains.

Both RL and HSP also face difficulties in domains that require long action sequences. Heuristic-search state-space planning is intractable in such domains, because of the very large search spaces, and reinforcement learning may require exponentially many execution trace to converge.

In order to address both the unique and the shared problems of HSP and RL, we propose a new research direction called *skill bootstrapping* (SB). The goal of SB is to provide an integrated learning and planning architecture that can improve its performance over time in complex domains. An SB agent starts with a basic set of primitive actions (and their preconditions and effects) as its model of the world. Over the course of solving numerous problems by applying HSP to the primitive actions, SB identifies recurring subgoals, for which it uses RL to create *skills* that can be applied within the HSP process to solve these subgoals more efficiently. The skills behave as partial policies that can be used reactively, without lengthy deliberative reasoning.

Once a new skill is learned, it becomes available for use by the planner along with the other primitive actions, allowing for more compact plans. Additionally, just as future plans can use learned skills, future skills may be built upon lower-level skills. Over the course of the agent's experience, this will eventually result in a *hierarchy* of skills that support high-level reasoning.

The proposed SB approach is still in its very early stages, so all of the design issues have not been solved. We present here our preliminary ideas about how to create SB; we use the present tense throughout the paper, but wish to emphasize that this is proposed research, for which no implementation yet exists. We will detail the architecture of the proposed SB framework, discuss possible test domains and how the SB architecture would handle them, describe how SB relates to other current research, and discuss the next critical research steps that will be taken.

Appearing in *Proceedings of the ICML/UAI/COLT Workshop on Abstraction in Reinforcement Learning*, Montreal, Canada, 2009.

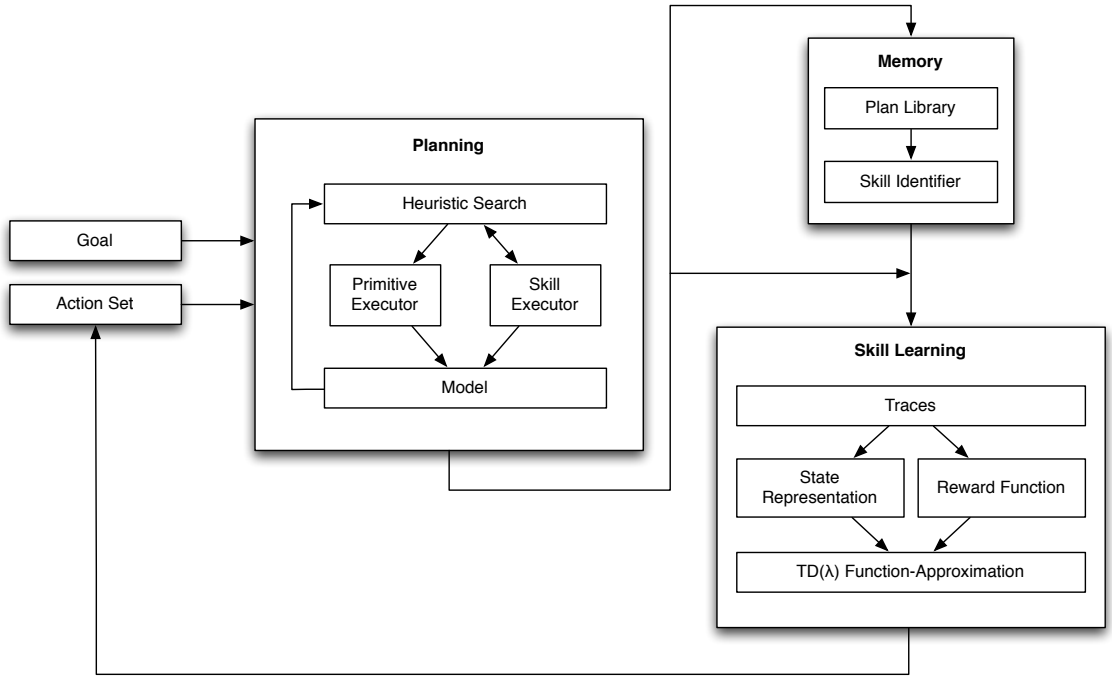

Figure 1. The SB architecture has three primary components: *Planning*, *Memory*, and *Skill Learning*. The planning component takes a set of actions and a goal, and uses heuristic search to find a plan that achieves the goal. Successful plans are stored in a *plan library* which is monitored by a *skill identifier* to find sets of plans that share common types of goals. These sets of plans are used to learn a parametrized skill that accomplishes that goal type. Learned skills augment the action set, and can then be used by the heuristic search planner. This process is applied repeatedly, resulting in a hierarchy of skills.

2. SB Approach

The SB architecture consists of three primary components: *planning*, *memory*, and *skill learning* (Figure 1).

The heuristic-search planning component takes as input the current goal to be achieved and a list of available actions. In a newly created SB agent, the planner simply uses a heuristic to guide a state-space search for the goal. The resulting plan is sent to the memory component, where the plan is stored and indexed in a *plan library*. A *skill identifier* monitors the plan library to find common *types* of goals for which large numbers of plans have been indexed. We say common *type* because goal states need not be identical, but must only share similar properties. For instance, picking up block A is not the same goal as picking up block B, but it is the same *type* of goal. These goal similarities can be determined using the organization of the plan library, similar to techniques used by case-based planning algorithms such as CHEF (Hammond, 1989).

Once a common goal type with sufficient relevant plans has been detected, a new skill is created to achieve that goal. *Skill learning* uses the traces stored in the plan library to create a policy for the skill. In other words, each plan retrieved from the plan library is rerun and the Q-values of the state-action pairs in the plan are learned by the RL process (see Section 2.2). (The reward value received at each state is set to 0 for all non-goal states and 1 for the goal state.)

Since learned skills become part of plans that are stored in the plan library, over time, skills will be developed that utilize other skills. This property is desirable because the resulting hierarchies of skills permit efficient planning and execution without a lengthy deliberative reasoning process.

The execution process, and possible replanning that may occur if a skill fails, are discussed next, in Section 2.1.

2.1. Plan execution and replanning

After producing a plan, the sequence of actions is executed by the execution modules. Primitive actions are

simply applied. Skills, however, require some execution monitoring, since their outcomes are not known with complete certainty. Specifically, when a skill is executed from a given state (real, or in the search's model), an ϵ-greedy policy is used. The ϵ-greedy policy is followed until the subgoal associated with the skill has been achieved, at which point control is returned to the top-level planning and execution process. RL updating is applied to this new planning trace, so that performance can be improved even after a skill is first learned.

If at any point a skill becomes "stuck" and cannot reach its goal state, then the skill is terminated and the planner replans the current subgoal from the current state. Such execution failures may be detected either through cycles in the execution, unusually long execution, or perhaps by associating certainty values with each skill that estimate how often each state has been explored in the past.

2.2. Skill learning

The purpose of a skill is to learn how to accomplish a *type* of goal, independent of the specific problem formulation. Additionally, it would often be useful to invoke a skill multiple times within the same problem, but parametrized to different contexts. To respect these properties, a converted representation of the world state must be provided as part of the input to the learning algorithm. This converted representation would be used both when a skill is initially learned and when it is invoked by future plans.

Consider the previous example of forming a skill for picking up a block. In this case, we may use plans that resulted in picking up Block A and plans that resulted in picking up Block B. In order for the skill to be parameterized—that is, applicable to any kind of block—the world state representation for this particular skill must explicitly indicate which block in the world is the target block. Furthermore, each such object may have relevant attributes (such as its position and size) that should be used when applying the skill. Therefore, a state representation is created for each skill using an appropriate vector format that includes any such parameters.

Even with a skill-specific representation of the world state, the skill would still need to be invariant across all possible goals. In the pickup-block example, there may be a designated position in the state vector that represents the target block's position and size, so the learning must capture a policy that respects differences in position and size. Traditional look-up table RL techniques would quickly become insufficient in this context. Saving an entry for every possible state that may be seen in a domain would be intractable, in terms of both memory usage and learning time. Instead, skill learning is performed with a TD(λ) function approximation approach (Sutton, 1988) that allows for compact storage of skills. Function approximation also enables the learning process to generalize to multiple states.

There are a number of different techniques for function approximation with TD(λ). We propose using function approximation based on an artificial neural net (ANN), in much the same way that TD-Gammon does (Tesauro, 1992). TD-Gammon is a well known implementation of an ANN TD(λ) algorithm, which learned to play backgammon very successfully, using only a raw input representation of the board. An ANN was chosen in TD-Gammon, because ANNs can learn non-linear functions of the input vectors, which proved necessary for learning how to effectively play backgammon. Because RL problems do not have a fixed training set and can continually learn, it was also found that increasing the number of hidden nodes in the ANN did not lead to overfitting of the data, as it often does in supervised learning domains. Instead, performance monotonically increased with the number of hidden nodes used in the ANN.

Because of these properties, an ANN makes a good choice for use with skill learning as well. Since skill learning will have to be applied to an unknown number of problems with various levels of complexity, the non-linearity of an ANN will allow the skill learning to be robust to even complex problems. Further, because performance monotonically increases with the number of hidden nodes, performance can easily be scaled up as necessary, while minimizing the risk of overfitting.

There are several key differences between TD-Gammon's implementation of ANN-based TD(λ) and our proposed approach for skill learning. The most obvious is that TD-Gammon played against another player (a duplicated version of itself) when learning. For skill learning, there is no other player; instead, learning is performed through successful plan traces (either from previously saved plans, or in actual or simulated execution). Additionally, since a model of the game was provided in TD-Gammon, the ANN estimated the V(s) state values, which represent the expected utility of each state. During execution, TD-Gammon would use the model to determine the resulting states from applying each valid action, and choose the state with the highest V(s) value.

An alternative approach is to estimate the Q(s, a) state-action pair values, similar to Sarsa(λ) function

approximation (Rummery & Niranjan, 1994), instead of the V(s) values. Since multiple state-action pairs can lead to the same state, learning only the V(s) values, and querying the model for the resulting states, requires fewer values to be learned. However, the fact that SB produces a skill hierarchy motivates our decision to estimate the Q(s, a) values, rather than V(s) values. Using V(s) values, the planner would handle skill execution as TD-Gammon did, by examining the resulting state of each possible action, and choosing the action with the highest V(s) value. However, because skills can be nested in the hierarchy, this process would potentially require significant additional state space expansion. For example, if skill sk_1 is considering the outcome of each of its possible actions, and one of those actions is another skill, then that skill actually has to be executed itself to determine the end state, even if that skill is not ultimately selected. Not only does this mean the full execution of each potentially used skill, but at each time step of execution for each of the child skills, they must also consider all possible actions. If the child skills also have nested skills, they too must be expanded.

Computing the result of a high-level skill therefore requires a large search space expansion, which is exactly what skills are intended to avoid. By instead learning the Q(s, a) values, the agent can quickly select the best action or skill to apply, without having to perform this state space expansion.

3. Example Domains

In general, the SB architecture is best suited for domains in which an agent may have very low-level action primitives, and can be trained first with simple problems, then over time presented with progressively more complex problems. In such domains, it may be difficult to design a complete, effective set of planning operators. The SB approach, however, permits an agent to construct its own set of HTN-like operators over time.

We present two examples of such domains, and discuss how the SB architecture might be applied within them: the taxi domain (Dietterich, 2000) and a chimps-and-bananas domain inspired by Köhler's chimpanzee experiments (1925).

3.1. Taxi domain

Dietterich's taxi domain (2000) is a grid world with various stations where people can be located, or wish to go to, and other locations where fuel can be obtained. The ultimate goal is to pick people up and take them to where they need to go, without running out of fuel. The taxi has actions for moving to the north, south, east, or west grid cell, picking up or dropping off a person at the taxi's location, and filling up at a refueling station.

An SB agent might first be given simple problems to solve, such as driving to a particular location. The agent will create plans that use the primitive movement actions to move to different locations. Once the agent has indexed a number of plans that involve reaching a location, the agent will develop a parameterized skill, DriveTo(loc), that allows the agent to travel to a location, loc.

Once this basic navigation skill has been learned, planning for taking a person to or from a location becomes trivial. For example, if person p needs to be picked up from location Red and be taken to location $Green$, the top-level plan would consist of the actions DriveTo(Red), Pickup(p), DriveTo($Green$), and Dropoff(p). After a number of these problems have been solved, a Transport(p, loc) skill might be developed for transporting a person. The Transport skill would be parameterized by the person (with properties such as their current position) and the location where they need to travel. All of the problems with this kind of goal would thus result in single-action plans.

3.2. Köhler's Chimp-Banana problem

A more complex domain is a recreation of Köhler's chimp-banana experiments (1925). In Köhler's famous study of chimpanzees' cognitive ability, one of his experiments involved hanging a banana from the ceiling, and placing boxes in the room such that a chimp would have to stack the boxes in order to get to the banana. This is an interesting problem because it requires planning, understanding of the world, and a set of physical skills that chimps would have developed in their lives.

In this domain, we can imagine creating an agent that has primitive abilities to move its legs, arms, and hands. The agent could start by being given a task to pick up a banana that is sitting within its grasp. The agent would then have to form plans to move its arm out, and grab the banana with its hand. Various problems could be given with the banana in slightly different locations, but still within grasp. This would eventually lead to skill development for grasping bananas. The agent could also be presented with problems of moving to different locations so that it develops walking skills. Similar skills for climbing or picking up boxes could also be developed. Eventually, in order to retrieve a banana from the ceiling, the agent would have enough skills to form plans that consisted of walking, climbing, picking up boxes, and grabbing bananas.

This would result in much higher-level reasoning and much more compact planning, than if the agent had to create such a complex plan using only the primitive actions (i.e., individual body movements).

4. Related Work

The concept of policy control that builds on lower-level action primitives to achieve a goal is not new to agent control. In planning fields, this notion is usually referred to as *macro-operators* or *macro-actions*. Macro-actions are generally constructed as a fixed sequence of primitive actions. Botea et al. (2005) present an algorithm called Macro-FF that examines a planning domain for potential sequences of actions to create macro-actions, and then filters that list based on heuristics and experience in training problems. Newton et al. (2005) use genetic algorithms on training problems to determine useful sequences of actions for use as macro-actions in planning. Marvin (Coles & Smith, 2007) is a learning algorithm that uses macro-actions to escape heuristic plateaus. Coles et al. (2007) extended Marvin to allow macro-actions learned in previous problems of the same domain to be applied to solve future problems. This large collection of macro-actions is stored in a macro-action library that is managed and pruned.

Two key commonalities of these approaches is that macro-actions are a fixed sequence of actions, and that the list tends to need to be pruned to avoid large collections of macro-actions. The SB approach proposed here differs in that skills are *not* fixed sequences of actions. Instead, they are policy control mechanisms that vary the action sequence depending on the particular state of the world. Because skills can vary their action sequence depending on the situation, a single skill could effectively represent a *collection* of macro-actions as one succinct unit.

Using hierarchies of actions in reinforcement learning has also been an area of active research. The MaxQ algorithm (Dietterich, 2000) used a designed hierarchy of subtasks to efficiently solve more complex problems. These subtasks are often referred to as *temporally extended actions*. More recent work has focused on automatically identifying the action hierarchy. Jonsson and Barto (2005) presented the VISA algorithm, which uses a Dynamic Bayesian Network (DBN) to assist in construction of the action hierarchy. The HI-MAT algorithm (Mehta et al., 2008) is similar to the VISA algorithm, but couples a DBN with a successful trajectory of a source reinforcement learning problem to determine the hierarchy.

With these algorithms, the action hierarchies are fixed structures that have a defined root structure and are specific to a single problem. With the SB architecture, skills form hierarchies, but are not fixed in structure and can grow over time. Additionally, skills learned in the SB architecture are not explicitly structured. That is to say, skills that are referenced by parents skills, do not have to be invoked by the parent. Rather, any skill can be independently invoked if it is pertinent to the problem at hand. This also means that skills can be shared among different problems, and their structure does not have to be relearned.

Other work on forming action abstraction comes from Simsek and Barto (2007). They use the same terminology of *skills* to represent policies that achieve some subgoals in a task. To identify skills, they examine the graph structure of reinforcement learning problems to identify states that are likely important in the problem. The SB architecture differs in that instead of explicit states being used to identify places for skills, *types* of goals are identified that may result in different states, but states that share similar properties and may be parameterized. SB also differs in that skills can be hierarchical, building on the abilities of lower-level skills.

5. Research Questions and Discussion

The SB architecture proposes a new direction for planning and reinforcement learning that leads to a number of important research questions.

The first important question is what kind of heuristic-search planning algorithm should be used, and how to handle heuristics. In practice, traditional methods for computing heuristics and planning may be utilized, but when skills are introduced, determining how to handle heuristics may become a problem, since heuristics are often computed as a relaxed version of the problem. In this case, it might be best to use a approximate postcondition for each skill (namely, the specific goal that the skill is intended to solve), even though there may also be other postconditions (side effects) that result from executing the skill. When expanding a skill in the actual search space, the side effects can be determined from the underlying primitive-action model.

A second question is how to handle any uncertainty in the model. The model the agent uses may represent the expected outcome of primitive actions, but during actual execution, the outcome could be different, and may prevent the current plan from being executable. In these cases, one approach to take is to replan to

the expected outcome of a primitive action. However, if there is a high variability for a primitive action's outcome, this may result in a lot of replanning. Since RL is effective in uncertain environments, it instead may be more effective to generate a skill that achieves the expected outcome.

A third question is how to organize the plan library so that similar plans may be easily detected. Further, it may often be useful to extract subgoals from plan traces in order to develop skills; however, it may then be unclear which prior actions in the plan were actually relevant for achieving that subgoal. An expensive solution would be to replan from the start state explicitly to the desired subgoal, but this would have to be done for every such plan. A more practical approach might be to evaluate all the other plans with that subgoal to determine what the necessary preconditions were and only take the relevant parts of the plan traces—but doing so might not be trivial.

A fourth question is how the skill learner can represent skills that involve a variable number of objects. For instance, in a blocks world that includes problems with different numbers of blocks, how can a mapping of the world state be created that is compatible with these differences?

Finally, another important question is how to structure a training regime such that the agent can develop a set of useful skills. One possible solution is to provide an expert instructor who can hand design a set of skills that may potentially be relevant for the agent, and who would give the agent progressively more difficult tasks. Another less user-intensive possibility is to allow the agent to explore the world, and detect common types of states that occur in exploration. The agent could then create its own set of goals that represent these types of states. By starting from many different random states, the agent could then develop plans for these goals that would lead to skill development in the usual way. The exploration process could then continue with the new skills augmenting the exploration.

Although there are significant challenges to overcome, we believe that the SB architecture represents a novel new direction that can combine the benefits of both heuristic-search planning and reinforcement learning. The SB model provides an effective and adaptable approach for designing agents that can operate in complex, dynamic environments.

References

Botea, A., Enzenberger, M., Muller, M., & Schaeffer, J. (2005). Macro-FF: Improving AI planning with automatically learned macro-operators. *Journal of Artificial Intelligence Research*, *24*, 581–621.

Coles, A., Fox, M., & Smith, A. (2007). Online identification of useful macro-actions for planning. *Proceedings of the International Conference on Automated Planning and Scheduling*.

Coles, A., & Smith, A. (2007). MARVIN: A heuristic search planner with online macro-action learning. *Journal of Artificial Intelligence Research*, *28*, 119–156.

Dietterich, T. G. (2000). Hierarchical reinforcement learning with the MAXQ value function decomposition. *Journal of Artificial Intelligence Research*, *13*, 227–303.

Hammond, K. (1989). *Case-based planning: Viewing planning as a memory task*. Academic Press Professional, Inc. San Diego, CA, USA.

Jonsson, A., & Barto, A. (2005). A causal approach to hierarchical decomposition of factored MDPs. *ICML '05: Proceedings of the 22nd International Conference on Machine Learning* (pp. 401–408). New York, NY, USA: ACM.

Köhler, W. (1925). *The mentality of apes*. London and New York: K. Paul, Trench, Trubner & Co., Ltd.

Mehta, N., Ray, S., Tadepalli, P., & Dietterich, T. (2008). Automatic discovery and transfer of MAXQ hierarchies. *Proceedings of the 25th International Conference on Machine Learning* (pp. 648–655).

Newton, M., Levine, J., & Fox, M. (2005). Genetically evolved macro-actions in AI planning problems. *Proceedings of the 24th UK Planning and Scheduling SIG*, 163–172.

Rummery, G., & Niranjan, M. (1994). *On-line q-learning using connectionist systems* (Technical Report). University of Cambridge, Department of Engineering.

Simsek, O., & Barto, A. (2007). *Betweenness centrality as a basis for forming skills* (Technical Report). University of Massachusetts, Department of Computer Science.

Sutton, R. (1988). Learning to predict by the methods of temporal differences. *Machine learning*, *3*, 9–44.

Tesauro, G. (1992). Practical issues in temporal difference learning. *Machine learning*, *8*, 257–277.