Combining dynamic abstractions in very large MDPs

Kurt Steinkraus, Leslie Pack Kaelbling
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
200 Technology Square, Cambridge, MA 01982
{kurtas, lpk}@ai.mit.edu

Abstract

One of the reasons that it is difficult to plan and act in real-world domains is that they are very large. Existing research generally deals with the large domain size by exploiting a single type of domain structure. In this paper, we create a framework that encapsulates existing and new abstraction and approximation methods into modules, and combines arbitrary modules into a system that allows for dynamic representation changes. We show that, using this framework, it is possible to solve larger and more interesting domains than were previously possible, and while there are no optimality guarantees, suitable module choices gain tractability at little cost to optimality.

Introduction

One goal of AI is to be able to plan and act in the real world. Unfortunately, most real-world domains are difficult for a computer to plan and act in for a variety of reasons, including the lack of a domain model, the continuous nature of state or controls, partial observability, stochasticity, and the sheer size of the domains. This paper tackles the last issue, that of domain size, and it does so for real-world types of problems, those that have multiple types and scales of exploitable structure scattered irregularly through the domain.

Optimality is an unreasonable requirement for very large domains and is in fact rarely required. What is desired instead is deriving a reasonable solution, acting "well enough." in a reasonable amount of time. These criteria are suggested by human performance, which is rarely optimal but gets the job done and scales up to very large problems. The performance curve for humans is flatter as problem size increases due to spending a comparatively long time analyzing a problem's structure, decomposing and simplifying it, and to spending comparatively little time doing repetitive, unstructured calculations.

There are many many existing methods for tackling very large domains, and each is useful in its own way, taking advantage of different types of domain structure. A problem with most of these methods, though, is that they exploit only one type of structure. Thus these methods do not scale up well: they either don't simplify the domain enough to make it tractable, or they oversimplify the domain and lose important structure. A better approach is to combine multiple abstraction methods, matching differing types of domain structure with abstraction methods that take advantage of it.

Another common problem with existing methods is that the representation is static. It is desirable to take advantage of the way that certain domain parts become more or less important in different situations, by planning over the whole domain in approximate terms and then filling in more detailed plans as corresponding domain parts are visited. That way, no wasted detailed planning is done for domain parts that are never visited.

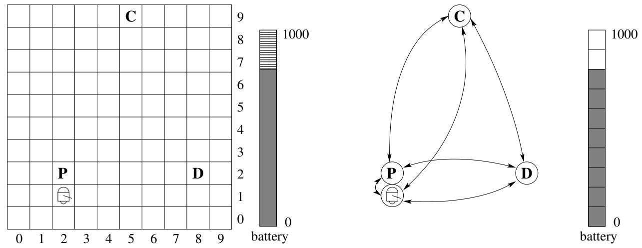
Rather than trying to modify a planner to exploit all the different ways of making an approximate plan, a better approach is to dynamically change the representation. Being able to change the detail level of the representation dynamically means that the plan's level of detail can be dynamically adjusted simply by keeping it in step with the representation.

Framework

We use factored Markov decision processes (MDPs) as our domain representation. These are MDPs whose state and action spaces are expressed as the cross product of state and action variables, in the same way that dynamic Bayesian networks are a factored representation of hidden Markov models. Factored MDPs allow modeling stochasticity in action outcomes, and they expose a certain type of structure (conditional independence) in the domain dynamics. We assume that the domain is completely observable and that a model of the domain is given, though it is our goal to relax these assumptions in later work.

To help understand the framework, consider a trivial example of a robot that lives in a 10×10 gridworld (see figure 1 (a)). The robot can move north, south, east, and west, but each time it moves, it may fail to move in the correct direction (or at all). Its goal is to move boxes from one location to another: a new package appears at the pickup location whenever the current one is dropped off. The robot has a battery level running from 1000 to 0; the battery gradually runs down and must be charged at a charger every so often. The robot gets reward for successfully transferring boxes, and the reward is discounted over time.

In this example, there are two different kinds of domain structure to exploit. First, the robot only ever will want to



(a) (b)

Figure 1: (a) The package and charger gridworld example. (b) An abstract version of the gridworld example.

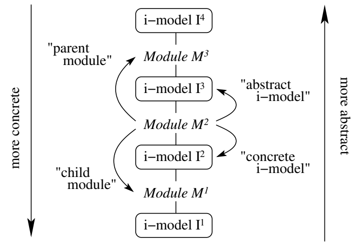


Figure 2: Module hierarchy nomenclature, with labels given relative to module M^2.

move between the pickup, drop-off, and charger locations. It seems reasonable to approximate the robot as having abstract actions that move it between these three locations. Second, the battery sensor is more fine-grained than needed; having a battery level of 871 isn't that different than a level of 872. It seems like a reasonable approximation to cluster together states that have similar battery levels, e.g., 1000-901, 900-801, …, 100-1, and 0 (see figure 1 (b)).

In order to allow these multiple abstraction methods to be used together, our framework packages each into a *module*, and individual modules are combined into a *module hierarchy*, which is then used to plan and act in a domain. The module hierarchy is a linear alternation between intermediate models, or *i-models*, and modules (see figure 2). The top-level i-model is a trivial MDP with one state and one action (the action's meaning is "act in the domain"), while the bottom-level i-model is identical to the input factored MDP. For each module M^j, i-model I^{j+1} is its *abstract i-model*, containing *abstract states*, *abstract actions*, etc., and i-model I^j is its *concrete i-model*. Also, module M^{j+1} is its *parent module* and module M^{j-1} is its *child module*. The input factored MDP model, I^0, is called the *atomic model*, with corresponding *atomic states*, *atomic actions*, etc.

Module hierarchy

The module hierarchy is arranged so that the i-models get increasingly abstract up the hierarchy. Each module M^j looks at I^j, applies some abstraction, and creates I^{j+1}. This layered approach allows abstraction modules to work together, each focusing on the domain structure it is able to simplify.

The i-models are intermediate representations for the domain, at each step of abstraction. An i-model needs to be able to contain information equivalent to a factored MDP (because the input model is a factored MDP and is I^0), and it needs to be able to contain temporal abstraction information as well (because the options module, described below, generates such information). We would therefore like to have a model type that is generalization of both semi-MDPs and factored MDPs. We might call this a *factored semi-MDP*.

It turns out that it is difficult to come up with consistent, intuitive semantics for factored semi-MDPs when allowing multiple unsynchronized action variables. Other researchers have begun to tackle the issue of multiple concurrent actions (Rohanimanesh & Mahadevan 2001), but this issue is a very difficult one and will not be explored further here. Instead, the domains are assumed to contain only a single action variable with multiple possible action values.

We define an i-model I as a tuple (S, a, τ, T, r):

- $S = \{s_i\}$ is a set of state variables; each state variable s_i ranges over a set of values $\{s_{im}\}$.
- a is the action variable; $a = \{a_n\}$.
- τ is a time distribution, where $\tau : \prod_{s \in S} s \times a \times \mathbb{N} \to \mathbb{R}$ gives the probability distribution over lengths of time that each action will take in each state.
- $T = \{t_k\}$ is a set of transition distributions, where $t_k : \prod_{s \in S_k} s \times a \times \mathbb{N} \times s_k \to \mathbb{R}$ gives the probability distribution over post-states, given the action and relevant portions of the pre-state space, $S_k \subseteq S$, and conditioned on how long the action takes.
- r is a reward function, where $r : \prod_{s \in S} s \times a \to \mathbb{R}$.

The first abstraction discussed in the example above, having abstract actions that move the robot between the pertinent locations, is similar to the options framework (Precup, Sutton, & Singh 1998) and to nearly deterministic abstractions (Lane & Kaelbling 2002); the options here are subpolicies to go from one location to another. The resulting abstract model is a semi-MDP (containing temporally abstract actions) with 3 locations instead of 100 x-y combinations.

The inputs to a subgoal-options module M^j are:

- G, a set of goal states; each goal state $\sigma \in G$ specifies values over some subset $S_{\text{goal}} \subseteq S^j$ of I^j's state variables; and
- $a_{\text{goal}} \subseteq a^j$, a set of action values that the options can use to reach a subgoal; these actions are replaced by the options in the abstract model.

In the example, $S_{\text{goal}} = \{x, y\}$, $G = \{(2, 2), (8, 2), (5, 9)\}$, and $a_{\text{goal}} = \{north, south, east, west\}$.

This module creates a set O of options, one for each goal $g \in G$. Each option o_g gives a policy that terminates when the restriction of the current state σ to S_{goal} is g. For each option, the probabilistic expected time transition, state transition, and reward functions are calculated (*pett*, *pest*, and *per*, respectively) for moving from goal state to goal state.

Let $u : \prod_{s \in S^{j+1}} s \to \prod_{s \in S^j} s$ be a mapping that unpacks the goal part of a state in I^{j+1} into its constituent state variables, giving a state in I^j. The subgoal-options module maps $I^j \to I^{j+1}$ as follows:

- $S^{j+1} = S^j \cup \{G\} \setminus S_{\text{goal}}$

- $a^{j+1} = a^j \cup O \setminus a_{\text{goal}}$
- $\tau^{j+1}(\sigma^{j+1}, \alpha, n) = \begin{cases} pett(u(\sigma^{j+1}), \alpha, n) & \text{if } \alpha \in O \\ \tau^j(u(\sigma^{j+1}), \alpha, n) & \text{if } \alpha \notin O \end{cases}$
- $t_k^{j+1}(\sigma^{j+1}, \alpha, n, \sigma'^{j+1})$
  $= \begin{cases} pest_k(u(\sigma^{j+1}), \alpha, n, u(\sigma'^{j+1})) & \text{if } \alpha \in O \\ t_k^j(u(\sigma^{j+1}), \alpha, n, u(\sigma'^{j+1})) & \text{if } \alpha \notin O \end{cases}$
- $r^{j+1}(\sigma^{j+1}, \alpha) = \begin{cases} per(u(\sigma^{j+1}), \alpha) & \text{if } \alpha \in O \\ r^j(u(\sigma^{j+1}), \alpha) & \text{if } \alpha \notin O \end{cases}$

The second abstraction discussed in the example above, clustering together states that have similar battery levels, is a simple state aggregation. Using the mapping between original and abstract states, transition dynamics for the new states can be calculated as the average of the dynamics for the corresponding original states.

The inputs to a state aggregation module M^j are:

- $s_j \in S^j$, the state variable being transformed;
- s_{j+1}, the replacement state variable; and
- $f : s_j \rightarrow s_{j+1}$, the aggregation function.

In the example, $s_j =$ *battery-level*, $s_{j+1} =$ *coarse-battery-level*, and $f(x) = \lceil x/100 \rceil$.

For any state $\sigma^j \in \prod_{s \in S^j} s$, let $f(\sigma^j)$ be the same state but with the value v of s_j in σ^j replaced by $f(v)$. Also, let $c(\sigma^{j+1}) = |\{\sigma^j : f(\sigma^j) = \sigma^{j+1}\}|$. The state aggregation module maps $I^j \rightarrow I^{j+1}$ as follows:

- $S^{j+1} = S^j \cup \{s_{j+1}\} \setminus \{s_j\}$
- $a^{j+1} = a^j$
- $\tau^{j+1}(\sigma^{j+1}, \alpha, n) = \frac{1}{c(\sigma^{j+1})} \sum_{\sigma^j \in f^{-1}(\sigma^{j+1})} \tau^j(\sigma^j, \alpha, n)$
- $t_k^{j+1}(\sigma^{j+1}, \alpha, n, \sigma'^{j+1})$
  $= \frac{1}{c(\sigma^{j+1})} \sum_{\sigma^j \in f^{-1}(\sigma^{j+1})} \sum_{\sigma'^j \in f^{-1}(\sigma'^{j+1})} t_k^j(\sigma^j, \alpha, n, \sigma'^j)$
- $r^{j+1}(\sigma^{j+1}, \alpha) = \frac{1}{c(\sigma^{j+1})} \sum_{\sigma^j \in f^{-1}(\sigma^{j+1})} r^j(\sigma^j, \alpha)$

The module hierarchy containing these two abstraction modules is given in figure 3 (a). In the i-models, the state variables are given in order to show how the input model is successively abstracted. The topmost module is a special module that creates an abstract i-model with only one state and one action. This "abstraction" involves solving the model using policy iteration (Puterman & Shin 1978), so that the single abstract action is a temporally abstract action meaning "execute the optimal policy that was calculated using policy iteration."

Planning and execution

Given a module hierarchy that some domain expert has created, the goal is to act well in the modeled domain. This will involve creating a plan or policy to follow, and then acting on that plan or policy while monitoring its execution.

Planning Planning happens implicitly in the module hierarchy as part of the process of creating the i-models. In order from the bottom to the top, each module M^j creates I^{j+1} from I^j.

The completed i-models for the example domain are shown in figure 3 (a). Planning is explicitly done by module M^3 as it uses policy iteration to solve the abstract model

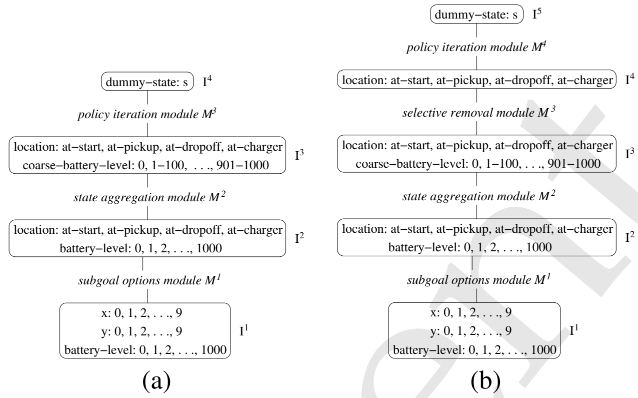


Figure 3: (a) A module hierarchy for the gridworld example. (b) The same module hierarchy with a selective removal module added.

I^3. But implicit planning also occurs in the subgoal-options module M^1, because creating each option requires creating a plan or policy to get from one location to another.

In most module hierarchies, the majority of the planning will happen this latter way, i.e., as part of the process of creating an abstract i-model, rather than explicitly. The overall plan is therefore composed of plan pieces that are created by and stored in the individual modules.

Execution After the i-models (and the relevant plan pieces) have been created, the module hierarchy starts executing. The initial execution happens by executing the single action in the top i-model.

When an action is executed in any I^j, it gets translated by M^{j-1} into one or more corresponding actions in I^{j-1}. This series of actions may consist of a single step, or perhaps it comes from executing a plan or policy. These actions then get translated further and further down the module hierarchy, and eventually they end up in I^1 , as atomic actions. These atomic actions are from the original domain model, so they can be executed as is.

This description makes it sound like the system expands the policy into a sequence of actions before executing any of them, but in fact the concretization of actions is more like a subroutine call than a static translation. When an abstract action in I^j is executing, the module M^{j-1} that created that action continues to select concrete actions in I^{j-1} to execute until the abstract action is done executing. At the same time as the system executes, it receives observations about the current state. These observations are passed to the modules in the hierarchy, so that the distributed policy can be closed-loop.

The execution loop consists of two steps:

1. Inform all modules of the current state.
2. Ask the topmost module for the next atomic action to execute.

When receiving a request for the next atomic action to execute, each module M^j follows these steps:

1. From M^{j-1}, get the next atomic action to execute.

2. If M^{j-1} returns a `terminated` action,
   (a) Choose the next action in I^j to execute, according to the current action that is executing in I^{j+1}.
   (b) If there is no such action, then the action in I^{j+1} is finished, so return a `terminated` action to M^{j+1}.
   (c) Tell M^{j-1} to execute the chosen action in I^j.
   (d) From M^{j-1}, get the next atomic action to execute.
3. Return the atomic action specified by module M^{j-1}.

In the example domain (starting as shown in figure 1) with the given module hierarchy, when the single top action is executed, module M^3 gains control and executes the optimal policy that it has found (i.e., the top-level action never terminates). The *goto-pickup* action is executed, and control passes to module M^2. This module passes on the *goto-pickup* action to module M^1, which determines that the current concrete action corresponding to *goto-pickup* is *north*. This is in I^1 and is therefore atomic, so the robot takes this action and observes what happens. Suppose this action fails to move the robot; module M^1 determines that *north* is again the concrete action corresponding to *goto-pickup*. This time, the action succeeds, and the module M^1 determines that the *goto-pickup* abstract action has terminated. So, it returns control to module M^2, telling it that *goto-pickup* has terminated, and module M^2 returns control to module M^3. Since the state variable *location* is now *at-pickup*, the optimal policy indicates that the *pickup* action should be executed. This is passed down the module hierarchy without being changed until it reaches the bottom; it succeeds, terminates, and control passes back up to the policy iteration module. Execution continues in a similar manner.

State representation As part of execution, a module M^j needs to determine the next action in I^j corresponding to the current abstract action in I^{j+1}, which means it needs to determine the current state in I^j. For each i-model, the current state is determined by successively translating the observed (atomic) state up from I^1. For instance, the initial atomic state is *x=2*, *y=1*, *battery-level=1000*. In I^2, the initial state is *location=at-start*, *battery-level=1000*, as translated by the options module. In I^3, the state is *location=at-start*, *coarse-battery-level=901-1000*, as translated by the state aggregation module. Finally, in I^4, the state is the dummy state.

Not all atomic states are representable at all i-models; for instance, only four of the hundred combinations of *x* and *y* correspond to values of the *location* state variable. At any given time, then, the current state may not be representable in all i-models, but this is not a problem. The only time that the current state needs to be representable in some i-model is when a new action is being selected, and as long as the abstract i-model that each module creates is valid, the current state will be representable when needed.

Note that when the i-models are initially created, the initial state needs to be representable in all i-models, because actions will be selected in every i-model when execution starts. This is why the *location* state variable has an *at-start* value, even though the *at-start* state will never be subsequently visited by any optimal policy.

Replanning The module hierarchy so far is a static entity: the decomposition and the modules are chosen and connected up, then the framework executes. A real advantage of the module hierarchy, though, comes in changing the representation dynamically, and updating the plan accordingly.

In the example, the robot's battery level isn't important until it gets low. A new module can be inserted right below the policy iteration module; this module can selectively remove or not remove the *coarse-battery-level* state from the abstract i-model that it creates (see figure 3 (b)), say, removing *coarse-battery-level* when its value is above *1-100*. This is advantageous because it gives the policy iteration module a much smaller model to find a policy for; in larger example domains, the size difference could make the difference between tractability and intractability.

When the battery level gets low enough, the selective removal module notices the low battery level and changes its abstract i-model to include *coarse-battery-level*. This causes a cascade of updates up the hierarchy, as each module propagates the change by updating its abstract i-model. In the example, the policy iteration module must update the optimal policy it has found to take account of the new *coarse-battery-level* state variable. All currently executing abstract actions are also terminated, since they may not be optimal (or may not even exist) any more; in the execution step right after replanning, the top i-model's single action is executed and gets expanded down the hierarchy, just like the initial execution step.

When each module M^j updates I^{j+1} to take account of changes, it could do whatever calculations it needs from scratch, but in a lot of cases, it can reuse most if not all of the data from the previous version of I^{j+1}. This causes the change of representation to happen much faster than the creation of the initial representation.

For instance, suppose the charger occasionally moves to one of 10 other locations. Instead of representing all possible charger locations in the *location* state variable, the charger location can be approximated as fixed in one place, and that "fixed" place can be changed as needed. When the charger location changes, the options module only has to create a new *goto-charger* option; it can reuse the options to reach other locations. Even better, changes to *location* cause the state aggregation module no new work at all, because it just copies the information about *location* from I^2 to I^3.

Reabstraction and replanning can therefore occur fairly frequently in this framework without being a burden. And for large domains, this reabstraction is certainly better than trying to represent and solve all cases simultaneously.

The ability for the representation to adapt to changes dynamically can be used in other ways than just allowing modules to initiate changes. For instance, if more processing power is suddenly available, it may be advantageous to reduce the amount of approximating, in the hopes of getting a better policy. Or, if a better atomic model of the domain's dynamics becomes available (say, because it is being learned online), then that better model can replace the old model without needing to plan from scratch.

Experiments

In order to test the module hierarchy, we used a simplified version of the computer game `nethack` (http://www.nethack.org). Nethack is a good domain for testing different approaches to solving real-world problems because it contains several different types of structure, some simple and some complex. The varying structure and the interaction between the different parts is representative of even larger, real-world domains, such a disaster relief robot, a Mars rover, or a general purpose battlefield robot.

In the particular simplified version of nethack that we used, the goal is to escape from a dungeon, where the dungeon has several levels and the outside world (escape) is at the top. The levels consist of rooms connected by hallways, and the levels are connected by staircases to the ones directly above and below. The player can move north, south, east, west, up, and down but not diagonally.

The game is not just a path planning problem, because the player has hunger and health. The player gets progressively more hungry as time goes on; if the player starves, his health slowly goes down, but there is food available to eat lying around the dungeon. The player's health normally stays constant, except that it goes down when starving or when attacked by a monster. The player can heal himself by using one of the medkits that are lying around the dungeon. The player can also pick up food and medkits and carry them around, for later use.

We represented the domain as an infinite-horizon discounted factored MDP with 11 primitive actions and a varying number of states (depending on the exact layout of the dungeon). Some of the actions, such as movement and attacking a monster, had probabilistic outcomes (e.g., the monster dies with a certain probability). The reward was set to be positive for escaping from the dungeon, negative for dying, and zero elsewhere.

Implemented module hierarchy

The module hierarchy that was created to solve the nethack domain is shown in figure 4. Even though the module hierarchy is a linear alternation of i-models and modules, this diagram shows which part of the domain each module changes, in order to better and more compactly illuminate the structure of the changes that each abstraction module makes.

This module hierarchy uses 18 modules instantiated from six module types. These modules were arranged in the module hierarchy and parameters were supplied by a domain expert, and then the system was used to solve sample nethack domains. It is important to note that, although the above modules were created in order to solve this simplified nethack domain, they are completely general and can be used in any other module hierarchy, given appropriate parameter choices.

Of the six module types used, four are the modules used in the package and charger gridworld example above. The remaining two modules allow subparts of the state space to be solved separately and recombined, in a way similar to the macro-action framework (Hauskrecht *et al.* 1998). The first module divides up the state space into subparts that can be abstracted and worked on separately. The second module finds the boundary states, where it is possible to go from one subpart to another. This second module creates an abstract i-model whose states are those boundary states and whose actions are subpolicies to get from one boundary state to another within a subpart. In the nethack domain, these modules are used to divide the dungeon up by floor, so that an escape path can be found separately through each floor.

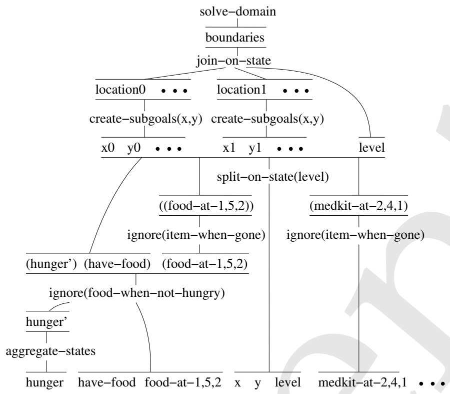


Figure 4: The module hierarchy used to solve the simplified nethack domain.

Experimental results

The running time and solution quality of the module hierarchy were compared both to policy iteration and to several abstraction methods used in its modules, operating individually. Unfortunately, it was not possible simply to do a direct comparison on the very large domain instances, since only the module hierarchy could solve those; the other methods took an extremely long time to run or ran out of memory. Instead, each method was run on a sequence of progressively more complicated instances of the nethack domain, until the method failed to return a result after one hour.

Experiments were run on several nethack domains, where the number of items in the domain, the number of levels, and the *x*-*y* size of each level were gradually increased, so that the state space size ranged from 9,600 to 108,900,000.

The running time results are given in figure 5. As expected, the module hierarchy is the only method that scales up to problems with very large numbers of states. Averaged across several trial runs, the reward gained by all methods was equal (modulo the randomness in the domain) on the instances where they succeeded in producing a solution.

An additional point of comparison between the different methods is the size of the domains that each tries to solve after having applied the pertinent abstractions. The previous methods like state aggregation end up attempting to solve models with hundreds of thousands of states by experiments 4 and 5. In contrast, the largest model that the model hierarchy needs to solve, either during the course of dynamics representation or when doing top-level planning, has just 450 states and 15 actions.

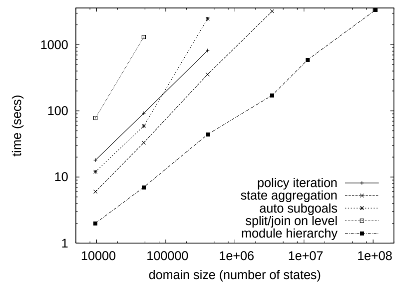


Figure 5: Experimental results.

Related work

There are a number of previous approaches to hierarchical planning under uncertainty, but most of these use a single abstraction method and take advantage of only one kind of domain structure. For instance, MAXQ (Dietterich 1998) and action hierarchies (Pineau, Roy, & Thrun 2001) decompose the domain solely based on subgoals to be achieved.

The hierarchies of abstract machines (HAMs) framework (Parr & Russell 1998) is closest to module hierarchies. In HAMs, each abstract machine is a non-deterministic finite state machine (NDFSM) and may also have choice points and/or make subroutine calls to other abstract machines. The idea is that the HAM imposes constraints on the generated policy, allowing a smaller portion of policy space to be searched. The HAM, when combined with the given domain MDP model, induces a new MDP such that an optimal policy for the induced MDP is the optimal policy consistent with the HAM for the given MDP. The HAM framework is very general, since the abstract machines can do the work of any abstractions expressible as NDFSMs.

Module hierarchies improve on the HAM framework in a few ways. One big advantage is that the module hierarchy's ability to dynamically change the representation without needing to recreate the plan from scratch. Another improvement is the module hierarchy's i-models, which are factored semi-MDPs. This exposes a lot more exploitable structure than using flat MDPs, for instance, certain abstractions like state aggregation now need only touch a small part of the model, and also the i-models allow structured representations of the transition and reward distributions.

Conclusions and future work

The module hierarchy trades optimality and high speed on small domains for tractability on large domains. Actual bounds on the reward loss of a module hierarchy are available if bounds are available for each module used in it. For instance, if each module M^j gives a bound on the possible reward loss when taking any policy in I^{j+1} and running the corresponding version in I^j, then these reward losses can be summed to find a bound on the reward loss for the module hierarchy as a whole.

Although the module hierarchy makes no guarantees about optimality, the nethack domain results show that, if modules are chosen and fitted together carefully, it may not be necessary to sacrifice much optimality in order to gain tractability. In fact, it is precisely because modules can be chosen to take advantage of the specific structure of different parts of the domain that larger instances of the nethack are tractable and yet the same level of reward is achieved.

The execution times show that the module hierarchy can handle larger domains than any single abstraction method; even so, there is room for considerable improvement. For instance, most of the module hierarchy's time and memory is spent creating and caching abstract dynamics. If the transition distributions and reward functions were structured, e.g., as algebraic decision diagrams, then the abstract dynamics could be created much more quickly while still retaining their compact structure.

Other ways to improve on this first module hierarchy system include monitoring abstract action execution and interrupting when low probability occurrences can be exploited, measurement of the applicability and "fit" of modules to certain situations, and extension to other models like POMDPs and relational MDPs. An eventual goal is to create a black-box system, where the modules themselves to determine where and when they apply, automatically constructing and rearranging the module hierarchy.

References

Dietterich, T. 1998. The MAXQ method for hierarchical reinforcement learning. In *Proceedings of the Fifteenth International Conference on Machine Learning*, 118–126. Morgan Kaufmann.

Hauskrecht, M.; Meuleau, N.; Kaelbling, L.; Dean, T.; and Boutilier, C. 1998. Hierarchical solution of Markov decision processes using macro-actions. In *Proceedings of the Fourteenth Annual Conference on Uncertainty in Artificial Intelligence*, 220–229. Morgan Kaufmann.

Lane, T., and Kaelbling, L. 2002. Nearly deterministic abstractions of Markov decision processes. In *Proceedings of the Eighteenth National Conference of Artificial Intelligence*, 260–266. AAAI Press.

Parr, R., and Russell, S. 1998. Reinforcement learning with hierarchies of machines. In *Advances in Neural Information Processing Systems*, volume 10. The MIT Press.

Pineau, J.; Roy, N.; and Thrun, S. 2001. A hierarchical approach to POMDP planning and execution. In *ICML Workshop on Hierarchy and Memory in Reinforcement Learning*.

Precup, D.; Sutton, R.; and Singh, S. 1998. Theoretical results on reinforcement learning with temporally abstract options. In *Proceedings of the Tenth European Conference on Machine Learning*, 382–393. Springer-Verlag.

Puterman, M., and Shin, M. 1978. Modified policy iteration algorithms for discounted Markov decision problems. *Management Science* 24(11):1127–1137.

Rohanimanesh, K., and Mahadevan, S. 2001. Decision-theoretic planning with concurrent temporally extended actions. In *Proceedings of the Seventeenth Conference on Uncertainty in Artificial Intelligence*, 472–479. Morgan Kaufmann.