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

November 5, 2003

1 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.

Our research tackles the last issue, that of domain size, and it does so for real-world types of problems, those that are complex but not random. Some domains are so random that there are no good approximations and planning has to proceed along the lines of modified policy iteration. Alternately, some domains like the traveling salesman problem are so structured that there are highly specific optimal algorithms tuned for that domain type. Real-world domains lie in-between these two extremes, having irregularly scattered different types and scales of exploitable structure.

Optimality is an unreasonable requirement for very large domains, and in addition, optimality is rarely required. Instead, what is desired is a guarantee of coming up with some solution, and being able to act "well enough." This criterion is suggested by human performance, which is rarely optimal but gets the job done and scales up to very large problems without giving up. This less steep performance curve for humans as problem size increases results from spending a comparatively long time analyzing a problem's structure, decomposing and simplifying it, and from spending comparatively little time doing repetitive, unstructured calculations.

There are many many previous methods for tackling very large domains, and each is useful in its own way, taking advantage of different types of domain structure. Unfortunately, these methods generally have some common weaknesses.

One weakness is that they only take one simplification step, or take advantage of only one type of structure. Thus, these methods do not scale up well; they either don't simplify the domain enough, or they oversimplify the domain and lose important structure. A better approach than trying to find a single best abstraction method is to use multiple ones, matching differing types of domain structure with abstraction methods that take advantage of it.

Another common problem with many previous methods is that they create (explicit or implicit) policies for the entire domain. This is unnecessary since only a small fraction of state space is ever visited, and creating the whole policy is intractable on very large domains. The envelope method [2, 1] is an exception and suggests that a good alternative is the creation of a plan that can be extended and repaired as needed.

A third weakness of many previous methods is that their 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 take advantage of 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. Also, this dynamic representation would allow for taking advantage of the changing availability of computational resources, or perhaps the availability of an improved domain model.

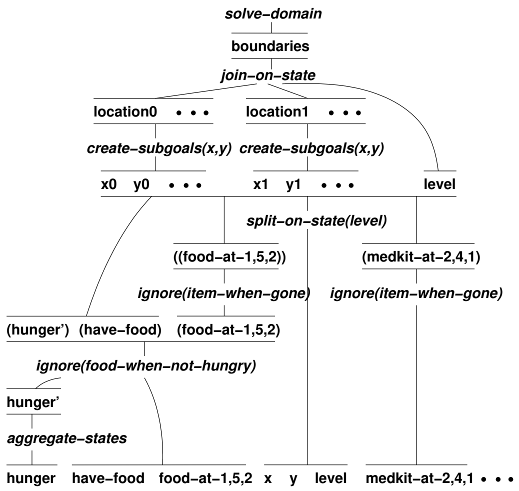

Figure 1: A sample module hierarchy. Modules are in italics and operate on different portions of the state space.

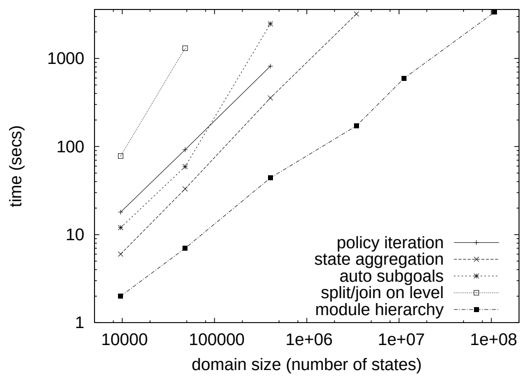

Figure 2: Experimental results.

2 Approach

We use factored (factorial) 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.

In order to use multiple abstraction methods together, each is packaged into a *module*, and individual modules are combined into a *module hierarchy*, which is then used to plan and act in a domain. Figure 1 gives a sample module hierarchy.

Each module operates on some part of the domain and abstracts it appropriately. Modules have a well-defined interface and behavior specifications, so that they are interchangeable and a general planning and execution system can be built to handle any abstraction. The modules create a succession of increasingly abstract models, ideally culminating in a trivial model at the topmost abstract level. Modules may be combined in serial or parallel, using a factored semi-MDP as an intermediate representation (where the model is *factored* to maintain the conditional independence structure, and it is *semi-* to allow temporally abstract actions).

State representation, planning, execution, execution monitoring, and replanning are all done hierarchically, distributed throughout each module. For instance, the only explicit planning is the trivial planning done at the top abstract level, but planning is also done when each module figures out the appropriate abstract dynamics to create. Execution similarly happens only explicitly in the top abstract domain, but an abstract action taken in that domain needs to be concretized appropriately by the various modules in the hierarchy.

We have tested the module hierarchy in 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.

We implemented the module hierarchy in figure 1, using 18 modules of 5 different types. Its running time and solution quality were compared both to policy iteration and to several abstraction methods used in its modules, operating individually. The running time results are given in figure 2. The reward gained by all methods was equal (modulo the randomness in the domain) on the instances where they succeeded in producing a solution, but the module hierarchy's curve is lower than the others as the problem size increases, and it is the only method that scales up to problems with very large numbers of states.

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 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.

Although the module hierarchy framework makes no guarantees about optimality (choosing tractability over optimality), the results on the nethack domain show that, if the modules are chosen carefully and fitted together appropriately, it may not necessary to sacrifice much reward value in order to gain tractability. In fact, so little reward value is lost precisely because those modules, containing the individually good abstraction techniques, can be chosen and arranged to take advantage of the specific structure of different parts of the domain.

3 Future work

The execution times above show that the module hierarchy can handle larger domains than any single abstraction method; even so, there is considerable room for improvement. Here are some of the interesting questions to investigate:

- Does a hierarchy of modules work as well with other model types like POMDPs, relational MDPs, and first-order MDPs? For instance, methods tackling POMDPs usually solve the POMDP or convert to belief space rather than creating an abstract POMDP, and so there may be few or no useful POMDP modules to create.

- What structure is useful in describing the domain? Currently, most of the module hierarchy's time and memory are spent calling functions in other modules and caching results. If the transition distributions and reward functions were structured, e.g., as decision diagrams, then the abstract dynamics would be created much more quickly while still retaining their compact structure.

- What sort of execution monitoring should there be? As an example, the actual execution of an abstract action could be compared to its expected dynamics; if they differ substantially, due to low-probability events, then it might be a good time to replan.

- What can be said about the overall optimality of a module hierarchy? Obviously, if each individual module is optimal, then the system's execution will be optimal, but there may be some notion like recursive or hierarchical optimality.

- How can the applicability or "fit" of a module be described? This would be some sort of measure of whether the information that a module abstracts out is important or irrelevant. It is unclear if this can be done on a module-by-module basis or needs to happen with respect to a module's children in the hierarchy, or even with respect to the whole hierarchy.

- How can modules cut corners without affecting the solution quality? For instance, a module may only need to calculate approximate abstract dynamics until the pertinent part of the state space is reached, or perhaps it can reuse some (or all) of the previously created plan when it replans.

- What benefits are there to representing the same part of the domain in multiple ways, and how should those multiple representations interact? Multiple representations might be useful, for instance, when planning to achieve multiple tasks.

- How can the module hierarchy be created automatically? This involves several parts: choosing abstraction modules appropriate to the domain, connecting the modules together, and determining appropriate parameters needed by the modules.

Our eventual goal is to create a black-box system, where the modules themselves determine where and when they apply, calculate or learn whatever parameters they need to, and automatically construct and rearrange the module hierarchy.

References

[1] J. Baum and A. Nicholson. Dynamic non-uniform abstractions for approximate planning in large structured stochastic domains. In *Proceedings of the Fifth Pacific Rim International Conference on Artificial Intelligence*, pages 587–598. Springer, 1998.

[2] T. Dean, L. Kaelbling, J. Kirman, and A. Nicholson. Planning with deadlines in stochastic domains. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 574–579. AAAI Press, 1993.