Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2004-065
AIM-2004-023

October 21, 2004

Combining dynamic abstractions in large MDPs

Kurt Steinkraus and Leslie Pack Kaelbling

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.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 using a static representation and 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 the dynamic changes of representation allow our framework 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.

Acknowledgements

The authors gratefully acknowledge the support of the Defense Advanced Research Project Agency under contract #6890633.

1 Introduction

Recent planning algorithms for deterministic and stochastic systems have improved considerably, allowing the solution of moderately large problems. When exact solution is required for relatively circumscribed applications, these approaches are clearly appropriate. In this paper, we wish to consider a somewhat different case, analogous to the action selection problem faced by a human, or by a robot operating in a highly complex and open-ended domain, such as disaster relief or general battlefield operations. In such domains, the world model is so big that, ideally, a planning algorithm would run in time that is effectively constant, independent even of the number of state variables in the model.

Our approach will be to use dynamic abstractions, so that the agent only ever needs to solve very small planning problem instances. This approach will, of necessity, give up on achieving optimal performance, instead emphasizing the ability to continue to behave without complete failure in situations of extreme complexity. As the agent moves through the environment, it represents the domain at multiple levels of abstraction. What makes our approach different from a variety of other abstraction methods is the dynamism of the abstraction: the agent's current view of its environment depends on the current state. In the current work, the adaptation methods are designed into the system; in future work, the agent should learn which abstractions are appropriately used in which circumstances.

This paper presents a framework for building agents using a dynamic combination of abstractions, describes a particular concrete set of abstraction methods, shows how they can be combined into a dynamically adapting hierarchy, and applies that hierarchy to the problem of controlling an agent in a moderately complex game domain.

2 Framework

We assume that the agent has a domain model expressed as a factored Markov decision process and a high-level goal articulated as a reward function over the variables in the factored model. In addition, it has a hierarchy of abstraction modules that dynamically create a hierarchy of abstracted versions of the base-level domain model.

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's goal is to move packages from one location to another. The robot's movement is stochastic, and it has a battery, whose charge gradually runs down from 1000 to 0 and needs to be charged at a charger. The robot gets reward (discounted over time) for successfully transferring packages. In this example, there are two different kinds of domain structure to exploit. First, the robot will only ever want to go back and forth between three locations: the pickup, drop-off, and charger points. Second, the battery sensor is more fine-grained than needed, so similar battery levels can be clustered together, say, in groups of 100 (see figure 1(b)).

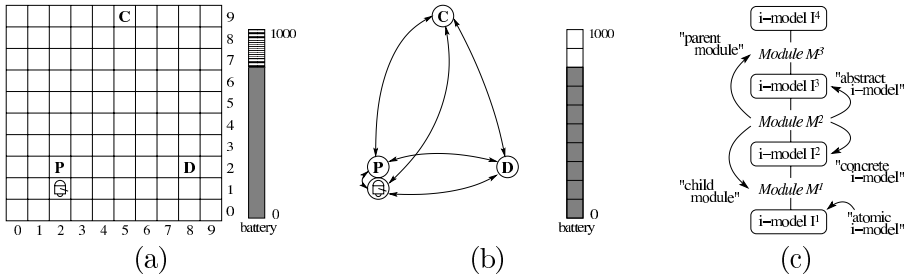

Figure 1: (a) The package and charger gridworld example. (b) An abstract version of the gridworld example. (c) Module hierarchy nomenclature, with labels given relative to module M^2.

In order to allow multiple abstraction methods to be used together, each focusing on the domain structure it is able to simplify, our framework packages each abstraction method into a *module*, and individual modules are combined into a *module hierarchy*, which is then used to plan and act in a domain. The modules in the module hierarchy induce successive *i-models* (see figure 1(c)). The top-level i-model is a trivial MDP with one state and one action (the action means "act in the domain"), while the bottom-level i-model, I^1, is identical to the input model.

2.1 Module hierarchy

Each i-model I^j is an intermediate representations for the domain and is created by module M^{j-1} looking at I^{j-1} and applying some abstraction. We let i-models be *factored semi-MDPs*, since they need to be able to represent the input model, a factored MDP, as well as temporal abstraction information (because the options module described below, for example, generates such information). Due to the difficulties in coming up with consistent, intuitive semantics for factored semi-MDPs when allowing multiple concurrent action variables, for now we assume that domains contain only a single action variable, with multiple possible 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 s_i ranges over a set of values $\{s_{im}\}$.
- a is the action variable; $a = \{a_n\}$.
- τ is a time distribution, where $\tau : \prod_{s \in S} s \times a \times \mathbb{N} \rightarrow \mathbb{R}$ 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 $t_k : \prod_{s \in S_k} s \times a \times \mathbb{N} \times s_k \rightarrow \mathbb{R}$ 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 the action's duration.

- r is a reward function, where $r : \prod_{s \in S} s \times a \rightarrow \mathbb{R}$.

2.1.1 Subgoal-options module

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 [1] and to nearly deterministic abstractions [2]; the options here are sub-policies to go from one location to another. The resulting abstract model is a semi-MDP with 3 locations instead of 100 x-y combinations.

The inputs to a subgoal-options module M^j are (a) G, a set of goal states, where each goal state $\sigma \in G$ specifies values over some subset $S_{\text{goal}} \subseteq S^j$ of I^j's state variables; and (b) $a_{\text{goal}} \subseteq a^j$, a set of action values that the options can use to reach a subgoal, where 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 (*pett*, *pest*, and *per*, respectively) are calculated for moving from goal state to goal state.

Let $u : \prod_{s \in S^{j+1}} s \rightarrow \prod_{s \in S^j} s$ 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 $I^j \rightarrow I^{j+1}$ as follows:

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

2.1.2 State aggregation module

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 (a) $s_j \in S^j$, the state variable being transformed; (b) s_{j+1}, the replacement state variable; and (c) $f : s_j \rightarrow s_{j+1}$, the aggregation function. In the example, $s_j =$ *battery-level*, $s_{j+1} =$ *coarse-battery-level*, and $f(x) = \lceil x/100 \rceil$.

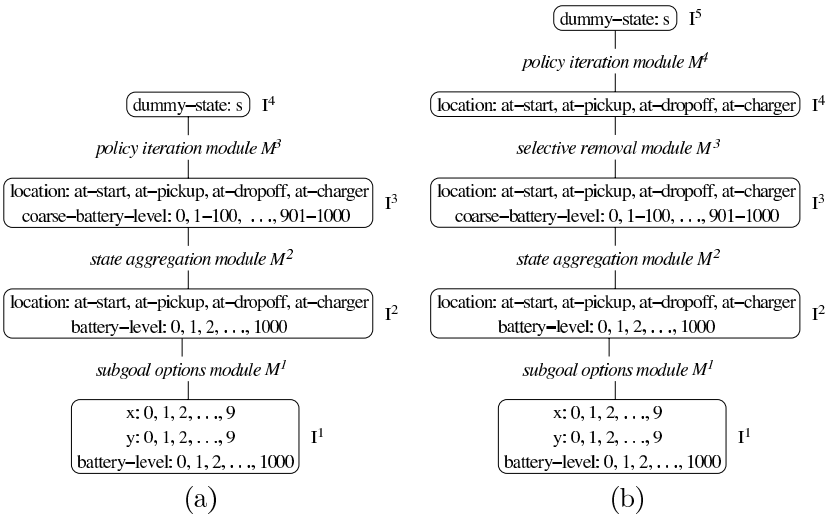

Figure 2: (a) A module hierarchy for the gridworld example. (b) The same module hierarchy with a selective removal module added.

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:

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

2.2 Planning and execution

Given a module hierarchy that some domain expert has created, the goal is to create a plan or policy to follow and then act on that plan or policy while monitoring its execution.

2.2.1 Planning

A module hierarchy for the example domain is given in figure 2(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 [3], so that the single abstract action is a temporally abstract action meaning "execute the optimal policy that was calculated using policy iteration."

In order from the bottom to the top, each module M^j creates I^{j+1} from I^j. Planning is implicit in the module hierarchy as part of the process of creating the i-models. Some obvious planning is done by module M^3 as it uses policy iteration to solve the abstract model 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 pieces that are created by and stored in the individual modules. Notice that each piece of planning is done in a much smaller domain than the whole 200,000 state domain: the subgoal-options module creates options for a 1100 state domain, and the policy iteration module creates a policy in a 66 state domain.

2.2.2 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, the module M^{j-1} beneath makes observations and chooses concrete actions in I^{j-1} to execute until the abstract action is done executing. Each time it selections an action, it is then translated further and further down the module hierarchy, and eventually it ends up in I^1 as an atomic action that can be executed directly in original domain model.

The execution loop consists of two steps: informing all modules of the current state, and then asking 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(a)) with the given module hierarchy, when the single top action is executed, 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 M^2. M_2 passes the action on to M^1, which determines that the current concrete action corresponding to *goto-pickup* is *north*. This is in I^1 and atomic, so the robot takes this action. Suppose this action fails to move the robot; M^1 determines that *north* is again what should be done. This time, the action succeeds, and the M^1 determines that *goto-pickup* has terminated. So, it returns control to M^2, telling it that *goto-pickup* has terminated, and M^2 returns control to M^3. Since *location* is now *at-pickup*, the optimal policy indicates that *pickup* should be executed. Execution continues in a similar manner.

2.2.3 State representation

As part of execution, a module M^j needs to determine the current state in I^j when choosing a new action. The current state is determined by having each module successively translate the observed (atomic) state up from I^1. Not all atomic states are always 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. This is not a problem, though, because the only time that the current state needs to be representable in some i-model is when a new action is being selected; at these times, an action has just finished, so the current state will be representable assuming that the abstract i-models are well-defined.

2.2.4 Replanning and changing the representation dynamically

The module hierarchy so far is a static entity: the decomposition and the modules are chosen, then the framework executes. There are several similar previous approaches to hierarchical planning under uncertainty (e.g., MAXQ [4] and action hierarchies [5]), but most use a single abstraction method and exploit only one kind of domain structure.

The most similar previous work is the hierarchies of abstract machines (HAMs) framework [6], which allows for combining multiple abstraction types expressed as non-deterministic finite state machines. While the module hierarchy is slightly better in that intermediate models are expressed as factored semi-MDPs and therefore easily expose their structure, the largest advantage of the module hierarchy 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. Suppose we insert a new module right below the policy iteration module and have it selectively remove or not remove the *coarse-battery-level* state from the abstract i-model that it creates (see figure 2(b)), say, removing *coarse-battery-level* when its value is above *1-100*. This gives the policy iteration module a much smaller model to find a policy for.

When the battery level gets low enough, the selective removal module notices it and changes the abstract i-model to include *coarse-battery-level*, causing 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, just like the initial execution step.

When each module M^j updates I^{j+1} to take account of changes, it could recalculate the i-model from scratch, but in many cases, it can reuse most of the solution 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 sometimes moves to one of 10 other places. Instead of representing them all, the charger location can be approximated as fixed in one place. 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, since it just copies information about *location* from I^2 to I^3. Reabstraction can therefore occur fairly frequently in this framework without being a burden. For large domains, this ability to keep the current representation small will likely mean the difference between tractability and intractability.

The ability to adapt the representation dynamically can be used in other ways as well. 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.

3 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 simplified version of nethack that we used, the goal is to escape from a dungeon, where the dungeon has several levels and the escape staircase 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. 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 he starves, his health decreases, but there is food available to eat lying around the dungeon. The player's health normally stays constant, except when starving or when attacked by a monster. The player can heal himself by using one of the medkits lying around the dungeon.

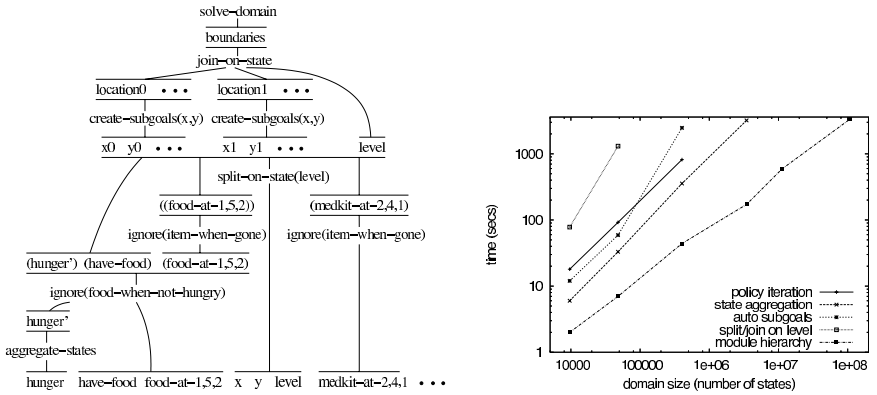

Figure 3: (a) The module hierarchy used to solve the simplified nethack domain. (b) Experimental results.

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.

3.1 Implemented module hierarchy

The module hierarchy that was created to solve the nethack domain is shown in figure 3(a). 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, who tailored the structure and parameters so as to solve the simplified nethack domain as well as possible. 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 other module hierarchies, 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 [7]. 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.

3.2 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. Each method was run on a sequence of progressively more complicated instances of the nethack domain until it failed to return a result after one hour. In successive domains, 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 3(b). 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.

The most important point of comparison between the different methods is the size of the domains that each works in after having applied pertinent abstractions. The previous methods end up attempting to work in models with hundreds of thousands of states by the fourth or fifth test domain. In contrast, the largest model that the module hierarchy needs to solve has just 450 states and 15 actions. Granted, several 450-state domains are solved during the course of execution, but this is certainly preferable to intractability.

4 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, a reward loss bound for the whole hierarchy can be found by summing reward loss bounds for the individual modules.

Although the module hierarchy makes no guarantees about optimality, the nethack domain results show that it may not be necessary to sacrifice much optimality in order to gain tractability. Tractability is gained because modules can be chosen to exploit the specific structure of different parts of the domain, and because those modules have the ability to reabstract dynamically, changing the representation to focus domain solving on the (small) currently relevant portions of state space.

The execution times show that the module hierarchy can handle larger domains than any single static abstraction method; even so, there is room for considerable improvement. For instance, most execution time and memory is

spent creating and caching abstract dynamics. We have almost completed an implementation that structures the dynamics as algebraic decision diagrams, allowing abstract dynamics to be created much faster 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, and extension to other models like POMDPs and relational MDPs.

References

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

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

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

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

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

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

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