Task Decomposition in Reinforcement Learning

Jonas Karlsson *
Computer Science Dept.
University of Rochester
Rochester, NY 14627-0226
jonas@cs.rochester.edu

January 28, 1994

1 Introduction

Decomposing a complex task into a set of simpler subtasks is a common technique used by problem solving agents. In reinforcement learning decomposition can be used by having separate modules learn to achieve individual subgoals independently and in parallel. Since each module can ignore parts of the state space that are not relevant to its task, learning time is drastically reduced. However, since a modular decomposition implies the loss of a global perspective of the task, the agent's performance depends on the strategy used to combine information provided by the different modules in order to select what action to execute. One class of such strategies use the local utilities provided by the individual modules to estimate the global utilities of actions. In previous work we have shown that an agent with pre-defined decomposition using such approximation strategies can achieve fast learning times with little loss in performance. However, since reinforcement learning is intended for domains where the human programmer has little knowledge about the structure of the task, it would be useful if the agent could discover a good task decomposition by itself. The only domain knowledge initially available to the agent is the reward function. The same knowledge used to encode the reward function can be used by the agent to define an initial decomposition of its task. This decomposition can then be refined, as the agent discovers the structure of the world and its task.

2 Reinforcement Learning

Reinforcement learning is a technique used to program agents when accurate information about the robot's domain and the effects of its actions is not easily available. This lack of information may be due to the stochasticity of the domain or that the cost of obtaining the necessary information is prohibitive. Instead the agent is simply given a "reward" that is a numerical indication of how well it is doing. Often, the reward function is assumed to be a part of the agents environment. However, in most applications the reward function is a part of the agent, that translates a state into a reward function [Whitehead and Ballard, 1989][Kaelbling, 1989]. In section 5 we discuss how this view of the reward function, and the information it encodes, as part of the agent's *a priori* knowledge, allows the agent to identify several features about its task.

The reward function is thus the agent designer's method of giving the agent information about what its task is and how it might best be achieved. Reinforcement learning techniques thus strive to maximize some measure of this reward, since the larger the reward gathered during the agent's lifetime, the better it has achieved its task.

Though there are many different reinforcement learning techniques, we will focus on Q-learning. We will only briefly describe Q-learning, a more detailed treatment can be found in [Watkins, 1989].

In Q-learning it is assumed that the agent-environment interaction can be modeled as a Markov decision process. In a Markov decision process (MDP), the robot and the environment are modeled by two synchronized finite state automatons interacting in a discrete time cyclical process. At each point in time, the following series of events occur:

1. The agent senses the current state of the environment.
2. Based on the current state, the agent chooses an action to execute and communicates it to the environment.
3. Based on the action issued by the agent and its current state, the environment makes a transition to a new state.
4. The agent's reward function evaluates the current state and returns a reward to the agent.

Q-learning tries to maximize its expected future reward, by keeping an estimate, Q, of the expected value of taking an action from any given state. This value changes depending on the reward according to the following formula:

$$Q(x,a) = (1-\alpha)Q(x,a) + \alpha[r + \gamma U(y)], \qquad (1)$$

*This work was supported by Air Force - Rome Air Development Center research contract F30602-91-C-0010

where x is an element of the set of states S, a is an element of the set of actions A, , α is a learning rate, γ is the temporal discount factor, y is the state resulting from taking action a in state x, and

$$U(y) = \max_{b \in A}[Q(y, b)] \tag{2}$$

The agent's world state can be composed of readings from input sensors, as well as values of internal state variables (i.e. memory). Since the agent uses a digital representation of these values each of the variables describing a world state can be rewritten as a set of bits. We can thus think states as a a set of propositions,

$$S = \{p_1, p_2, \ldots, p_n\}, \tag{3}$$

and any given state as a set of truth values for each of the propositions. Classes of states can now be described as a set of propositions, negated or not. For example,

$$C = \{p_2, \neg p_4\} \tag{4}$$

would denote all states in which p_2 was true, and p_4 was false. If p_2 corresponded to a `wall_to_left` sensor, and p_4 indicated the value of a `wall_in_front` sensor, C might be the set of states in which a wall-following robot was accomplishing its task. As we will see below, such classes of states will be useful when defining an agent's reward function. Furthermore, if we describe a state as a set of propositions, in particular a goal state, then we can regard each proposition as a sub-goal.

3 Modular decomposition in Reinforcement Learning

In [Tenenberg *et al.*, 1993] and [Whitehead *et al.*, 1993] we show how a reinforcement learning agent can decrease its learning time by using a modular decomposition, where subtasks are learned by independent, concurrently active modules. This differs from other work in reinforcement learning, where tasks are decomposed into a set of *sequential* sub-tasks which are then learned [Sing, 1992][Mahadevan and Connell, 1991]. In our work, the decomposition does not impose an ordering on the sub-tasks, and each module learns its task in parallel with the other modules. Each module uses Q-learning to learn its subgoal, and gets reward *only* dependent on how well it is accomplishing the sub-task. Each module therefore needs a separate reward function, independent of all the other reward functions. In addition, each module only learns in the part of state space needed for its task, thereby making it easier to find solutions.

The agent contains an *arbiter* which considers the information provided by the individual modules in order to select which action to execute next. The information available to the arbiter are the modules' Q-values (or expected utilities), and it must therefore try to combine these values in a way to approximate the "real", global, expected utilities. We have experimented with two such approximation strategies: nearest neighbor and greatest mass. Nearest neighbor simply estimates the global utility by selecting the largest of the local expected utilities.

$$Q_{nn}(x, a) = \max_{1 \le i \le n} Q_i(x, a)$$

Greatest mass adds the Q-values given to each action by the modules, and selects the action with the highest sum.

$$Q_{gm}(x, a) = \max_{a \in A} \sum_{i=1}^{n} Q_i(x, a)$$

In experiments described in more detail below, we have shown that these approximation strategies lead to good performance in some simple domains. Furthermore, the modular decomposition drastically reduces the size of the agent's state space (since each module only pays attention to the part of the space relevant to its sub-tasks). Since learning time grows with the size of the state-space [Whitehead, 1991], agents using the modular approach have a much faster learning time than those that do not.

4 Task decomposition in two domains

We have experimented with the modular decomposition approach in two relatively simple domains. In both cases, the decomposition, the individual reward functions, and the approximation strategy were predefined by the experimenter.

4.1 Grid World

In [Whitehead *et al.*, 1993] we describe a domain consisting of a 20x20 grid, where the task is to reach a set of n "goal" cells in the grid. The agent can move up, down, left, and right, and has the following sensors: `<X, Y, A>`, where `X` and `Y` describe the agent's location in the grid, and `A` is a vector of n bits, where bit i is set if goal i has been reached. The overall goal of the agent can then be described by the set of states,

$$G = \{a_1, a_2, \ldots, a_n\} \tag{5}$$

It would be sufficient to define the reward function to give a positive reward for states in G, and 0 reward for all other states:

$$R(x) = \begin{cases} 1 & \text{if } x \in G \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

However, we could give more information to the agent by also giving rewards as the agent reaches its subgoals. Then, the reward function might be defined as:

$$R(x) = \frac{1}{n} \sum_{i=1}^{n} [a_i] \tag{7}$$

(where $[a_i]$ is 1 if a_i is true, and 0 otherwise). In both cases it is clear that the agent's goal is to reach a state where all the a_i's are true. In the second case however, it is also clear that not only can the a_i's be considered subgoals, but they are also *independent* in the sense that the reward received for being in a state where a given a_i is true, is independent of the truth value of the other a_i's. This can be interpreted as the utility of achieving one subgoal is independent of achieving the other subgoals. Therefore, when the reward function defines such a relationship among subgoals, we will say that the subgoals are "utility independent".

Using the second version of the reward function, we decomposed the task into n modules with state-spaces defined by the sensors: `<X, Y,` a_i`>`. Each module had a reward function $R_i(x) = \frac{1}{n}[a_i]$. Figure 1 compares the learning time for agents using the two algorithms with an agent that does no decomposition, as the number of locations the agent must visit in the grid increases. The learning time is the number of steps until the agent achieves a performance level within 90% of that of an agent using a perfect nearest neighbor strategy. As a new location is added to the task, the agent's state description is increased by one bit (to indicate whether the new location has been visited), and therefore the size of the agent's state space is doubled. Since learning time is grows with the size of the state space [Whitehead, 1991], it is not surprising that the agent not using decomposition (called the monolithic agent) would have a roughly exponential increase in learning time (the learning time actually increases slower than an exponential rate, since as the number of goals increase, the *goal density* also increases, effectively making it simpler for the agent to solve the task). The learning time remains almost constant for the agents that decompose the task into separate modules since as a new location is added to the task, the size of the state space of all modules remains constant, and a new module is created to learn how to reach the new location. For the grid world domain therefore, the simple approximation strategies do not lead to any significant performance loss.

4.2 Simple Driving

Simple Driving (or SID) is a more complex domain, where the agent has to drive on a road, while negotiating obstacles, street lights, and road signs [Karlsson, 1993]. However, we will discuss only a subset of the domain that includes street lights, small, and large obstacles, as well as the agent. As in grid world, the agent moves in discrete steps from cell to cell, but its movements are limited to: standing still, moving forward one cell, turning left and right, and making left and right "jumps" where the agent moves on step diagonally, and then forward one cell. Figure 2 shows an example situation encountered by an agent in SID. The 4x4 sub-grid shown also defines the agent's sensor range: it can sense the presence of objects in the sub-grid extending two steps in front, and one step back of the agent, as well as one step to the left and right of the road. Table 1 list some of the very large set of specialized sensors available to the agent. The sensors with indices describe the presence of objects at the co-ordinates of the sub-grid surrounding the agent. Thus, `road(-1, 0)` is 0, is there is no road (but a sidewalk) to the agent's immediate left, and `obst(1, 1)` detects the kind of obstacle present in the cell diagonally to the right of the agent. The `road` sensor returns 0 if there is a sidewalk at the specified location, and 1 otherwise. The `obst` sensor returns 0 if there is no obstacle at the specified location, 1 if there is a small obstacle, and 2 if there is a large obstacle. The `lightVal` sensor returns 0 if there is no street light ahead of the agent, 1 if there is a light and is green, 2 for a yellow light, and 3 for a red light. The `lightDist` sensor returns how many steps ahead the light is.

sensor	value	sensor	value
`road(-1, 0)`	0–1	`road(0, 0)`	0–1
`road(1, 0)`	0–1	`road(-1, 1)`	0–1
`road(0, 1)`	0–1	`road(1, 1)`	0–1
`road(-1, 2)`	0–1	`road(0, 2)`	0–1
`road(1, 2)`	0–1	`obst(-1, 0)`	0–2
`obst(0, 0)`	0–2	`obst(1, 0)`	0–2
`obst(-1, 1)`	0–2	`obst(0, 1)`	0–2
`obst(1, 1)`	0–2	`obst(-1, 2)`	0–2
`obst(0, 2)`	0–2	`obst(1, 2)`	0–2
`lightVal`	0–3	`lightDist`	0–2

Table 1: Some of the sensors available to the agent in SID

The agent's task is loosely specified as successfully driving on the right side of the road, avoiding obstacles, and not running any red lights. We might thus define the reward function as follows:

$$R(x) = R_{\texttt{road}}(x) + R_{\texttt{obstacle}} + R_{\texttt{streetlight}}(x) \quad (8)$$

So that the total reward is a combination of of three separate reward functions, each contributing a measure of how well the agent is doing with respect to different aspects of the tasks. We'll define the separate reward functions as follows:

$$R_{\texttt{road}} = \begin{cases} c_r & \text{if } \texttt{road(0,0)} = 1 \\ 0 & \text{otherwise} \end{cases} \quad (9)$$

$$R_{\texttt{obstacle}} = \begin{cases} 0 & \text{if } \texttt{obst(0,0)} = 0 \\ -\frac{1}{2}c_o & \text{if } \texttt{obst(0,0)} = 1 \\ -c_o & \text{if } \texttt{obst(0,0)} = 2 \end{cases} \quad (10)$$

$$R_{\texttt{streetlight}} = \begin{cases} 0 & \text{if } \texttt{lightVal} = 0 \\ -c_s & \text{if } \texttt{lightVal} = 1 \\ & \text{and } \texttt{lightDist} = 0 \\ \frac{1}{2}c_s & \text{if } \texttt{lightVal} = 2 \\ & \text{and } \texttt{lightDist} = 0 \\ c_s & \text{if } \texttt{lightVal} = 2 \\ & \text{and } \texttt{lightDist} = 0 \end{cases} \quad (11)$$

Though each of these individual reward functions might work well in isolation, when they are combined, the magnitude of the individual constants c_r, c_o, and c_s becomes

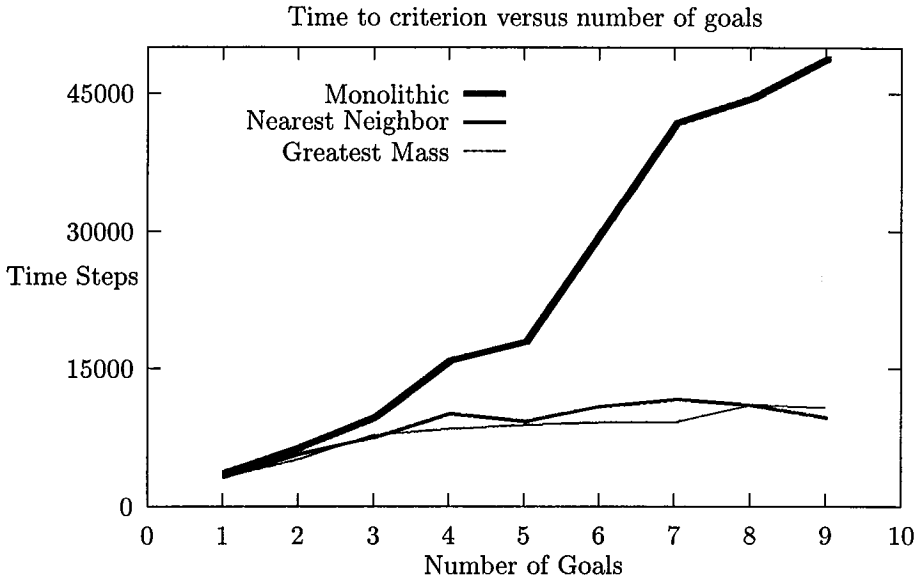

Figure 1: The time needed to reach the "reasonable-performance" criterion as a function of the number of goals in a multi-goal task for two modular systems and a monolithic system. The learning time required by the modular systems is largely independent of the number of goals.

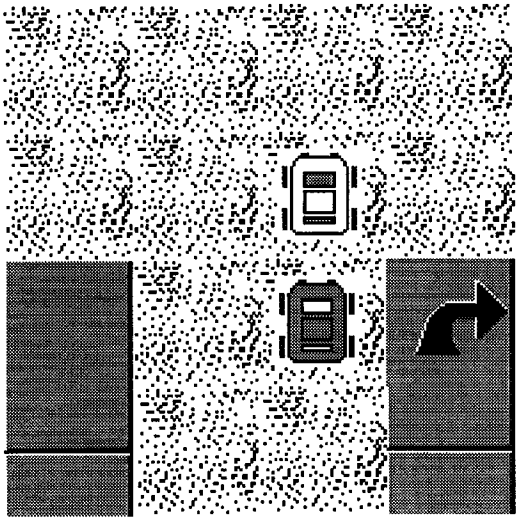

Figure 2: An example scenario in SID. The agent is the dark car

a crucial factor in determining the agent's behavior. The relative magnitudes of the constants indicate how the agent should behave when faced with a choice of reward sources. If the punishment for running into an obstacle has a much smaller magnitude than the reward for going through a green street light, it might be worthwhile for the agent to ignore the fact that there are obstacles in its way, and simply make sure that it always goes through the light when it is green.

In experiments where $c_s > c_o > c_r$, we discovered that depending on the type of sub-tasks needed to be accomplished, the performance of different approximation strategies varied greatly [Karlsson, 1993]. For example, figure 3 shows the cumulative reward for agents using the two approximation strategies in a domain where only roads and obstacles appear. The nearest neighbor strategy keeps leading the agent into obstacles, since by only taking account of the maximum local Q-value, the negative utilities learned by the obstacle avoidance module are always ignored.

Similarly, in a world where only streetlights appear on the road, the greatest mass strategy performs slightly worse than nearest neighbor. this may be due to the fact that the greatest mass approximation strategy attempts to discover a "compromise" path to solve all sub-tasks concurrently, but in the case where all sub-tasks are sources of positive reward, it is usually best to commit completely to one task while ignoring the others, solving each in order.

The experiments in the SID domain not only demonstrate that approximation strategies fail differently depending on the types of sub-tasks, but also show the difficulty of designing a correct reward function. By assigning rewards to states, the reward function ranks all the possible states according to how preferable they are. In the grid world case, this ranking was simple: states where more locations have been visited are preferred to those where few locations have been visited. In SID it is not so clear what the correct ranking should be. The reward function designer must consider all possible combinations states (for example, being on the road, going through a small obstacle vs. being off the road but avoiding a large obstacle vs. going through a yellow light while going through a small obstacle etc.). a simple linear combination of individual rewards may often produce unexpected results. for example, in our experiments, the utility of going through a green light was large enough that the agent attempted to pass through green light, disregarding the fact whether any obstacles were in the way. Thus, while the agent is acting correctly relative to the provided rewards, the resulting behavior was not the desired one. It seems that while the individual sensors (presence of obstacles and so on) used in the reward functions are the relevant ones, the correct reward function should be something more complex than a simple linear combination.

5 Identifying an initial task decomposition

In order to decompose a task into the modular architecture, the agent must define four elements: the state description of the sub-goals, the reward functions for the individual sub-goals, what parts of state-space are relevant to each sub-goal, and how the local reward functions should be combined to approximate the global reward function. In the ideal case, the agent would learn a decomposition where each module is learning a part of the task that is utility independent, *structurally independent* and *syntactically independent* from the other sub-tasks. Two goals are structurally independent if and only if there exists a path through the agent's state-space, starting at a state satisfying the first goal, ending at the second goal, where the first goal remains valid on all states on the path, and a similar path exists starting at the second goal and ending at the first, that never invalidates the second goal. Syntactical independent sub-tasks are tasks whose relevant parts of state-space do not intersect at any point. The presence or absence of these independence properties in the task vary the difficulty of finding a good decomposition, and vary the likelihood that the approximation algorithm will be close to the global reward function.

Since any prior information given to the agent may have little relevance to the actual structure of the wold and its task, one would like the agent to be able to discover a decomposition completely on its own. However in many cases a pre-determined decomposition may be sufficient most of the time. Below, we describe how the agent can analyze the reward function in order to define an initial decomposition. We assume, that the reward function is completely accessible to the agent, even to the point that the actual definition of the function is available in some simple form that can be analyzed by the agent.

5.1 Determining sub-goals

Since the reward function identifies goal states, described by bit values or propositions, we can regard each of the propositions, or sub-sets of the goal-propositions as sub-goals.

If there are utility independent sub-goals, this should be apparent from the reward function. The human designer will certainly be able to determine if the function is defined as some combination of independent reward functions. It would also be possible to require that the formulation of the reward function be given explicitly to the agent in a format allowing it to detect whether it is composed of a set of independent reward functions. Otherwise, the agent could use the heuristic method of sampling the rewards at some subset of the state-space and determine statistically what sub-sets of the goal descriptions are utility independent.

The agent cannot determine from the reward function alone whether there are structurally or syntactically inde-

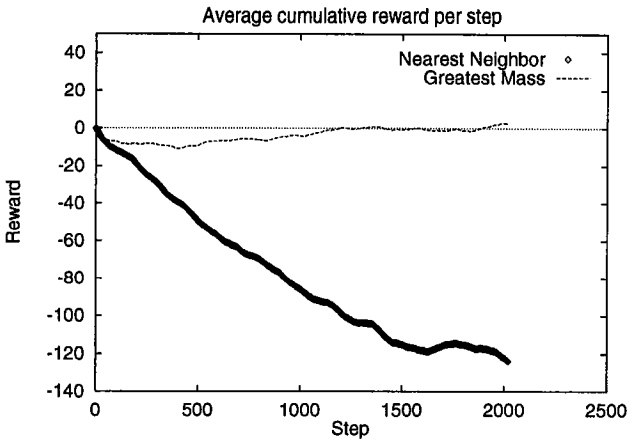

Figure 3: Cumulative reward per step for nearest neighbor and greatest mass strategies. Small and large obstacles appear in the world. Nearest neighbor performs worse, since by only considering the highest utility action, the obstacle avoidance module is ignored.

pendent ways to decompose the tasks. These properties can only be determined by actually exploring the state-space. Furthermore, if the agent decides that some partition of the goal description define utility independent sub-goals, it is possible that some of the sub-goals can be further decomposed into structurally independent tasks.

The agent could then use the heuristic methods of assuming some partition of the goal state is structurally independent. For example, the agent could attempt to treat each individual proposition as an independent sub-goal, or bits generated by the same sensor as one sub-goal. That is, in SID, all the bits defined by the `obst` sensor could successfully be treated as one sub-goal, whereas separating each bit would probably not be a successful decomposition.

5.2 Selecting individual reward functions

Once a decomposition has been determined, reward functions must be assigned to each module, reflecting how well the particular module has achieved its sub-task. While the reward functions should not depend on the status of other sub-tasks, they need to reflect the global reward function so that the approximation strategy will be close to the global reward. If all sub-tasks are utility independent it is simply a matter of assigning the individual reward functions to the appropriate module.

If the sub-tasks are not utility independent it is not clear how to define the individual reward functions. One might try simply assigning equal reward to all sub-goals assuming that combinations of sub-tasks don't arise frequently (i.e. in SID the road is mostly empty but occasionally there is an obstacle), and if the relative magnitude of the rewards are incorrect the rare occurrences when the approximation fails can be handled by some other mechanism (as briefly described below). Another approach would be to try and approximate utility independence. Using the global reward function, the agent can sample states where different combinations of goals are active and attempt to find a set of reward functions that approximate the global reward.

5.3 Partitioning the state space among modules

Since most of the improvement in learning time when using a modular approach, results from the fact that each module only learns its task in a sub-set of the state space. The largest improvement results when the sub-goals are syntactically independent, and the state-space is partitioned into equally sized parts.

Unless the state-space partition is provided in advance by the human programmer, the agent can only gradually learn what part of state-space is relevant for each sub-task. Statistical methods such as discussed by [Chapman and Kaelbling, 1991] could be used, though the method will be complicated by the fact that the reward function might be an incorrect approximation. A reasonable heuristic however, would be to include those sensor bits that determine whether the sub-goal satisfied, and excluding those bits defining the other sub-goals. If all other state description bits are shared learning time is still reduced drastically, as demonstrated in the grid world experiments.

5.4 Selecting the approximation strategy

There are clearly any number of approximation strategies that can be used other than nearest neighbor and greatest mass. If the human designer has any knowledge about the

task, such as it consists only of positive reward sources (as in grid world), it might be best to select an approximation scheme that is well suited to it (such as nearest neighbor). If there is no domain knowledge to guide the selection of the approximation strategy, greatest mass seems to be a reasonable choice. It does not exclude information from any module (as does nearest neighbor), and it encodes the assumption that a simple linear combination of local utilities will approximate the global utility. Greatest mass furthermore has the desirable property that if in the situation facing the agent, only one module is relevant, that module will in general determine the agent's behavior. It is only in situations where several modules provide information that the strategy might select an inappropriate action. Since in many domains those situations will be rare, it may be possible to handle these exceptional situations using a different mechanism.

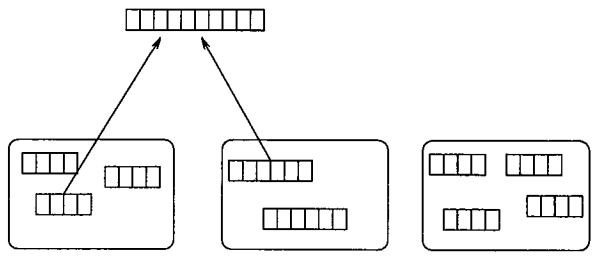

Figure 4: The state spaces of two modules are merged to form a new module. The ne module contains more information since its state description is larger, and its state-space is also larger than that of the individual modules. To avoid having to learn in this larger state-space, the merged module only supersedes the original ones in situations where the original decomposition fails.

6 Recovering from incorrect decompositions

As mentioned above, it would appear that in many domains situations where many sub-tasks need to be attended to concurrently are fairly rare. In grid world, the agent is usually far away from many of the goal locations and attempts to find the best path to a small set of close-by locations. In SID, it is not often that the agent is faced with both obstacles and street lights. Thus for a large part of the agent's life time it is only combining relevant information from a small set of modules (eg. if there are no obstacles in sight, the obstacle avoidance module has little preference over which action is executed, and thus has little to no influence on the agent's decision). In many cases where several modules are being combined, the approximation algorithm may still produce good results, but it is clear that there will be some cases where the loss of global information resulting from decomposition makes it impossible to learn the correct action to execute. The agent can then try to relax the decomposition in order to get the necessary information in order to solve the task. A new module, with a state space consisting of the combination of state-spaces of two or more modules, may have sufficient information to learn the correct actions in complex situation.

Figure 4 illustrates how two modules can be merged into a new module with a more precise state-description. If the two modules have state-spaces defined by the sensors

$$C_1 = \{b_1, b_2, b_3, b_4\} \text{ and} C_2 = \{b_3, b_4, b_5, b_6, b_7, b_8\}$$

Then the new module's state space is defined by the combined sensors

$$C_1 \bigcup C_2 = \{b1, b_2, b_3, b_4, b_5, b_6, b_7, b_8\}$$

Creating a new module violates the whole notion of decomposition. Though its state space is more informative, it is also much larger than the state spaces of the original modules. In order to avoid having to learn in this larger state space, the agent will only use the new, merged module, in those situations when it is necessary. To accomplish this selective use of the new module the agent must be able to determine at which states the original decomposition fails (in order to determine when to use the merged module), and which modules are relevant in that situation (in order to determine which modules to merge).

To determine when to create a new module, the agent can try to detect when its local approximation scheme (eg. greatest mass) fails to accurately approximate the global utility of actions, as defined by the global reward function. To detect this failure it might be sufficient to notice any large differences in the amount of reward given by the local reward functions, and that given by the global reward function. If at some point, when, for example, the two modules in the description above are in states

$$C_1 = \{b_1, b_2, \neg b_3, \neg b_4\}\, C_2 = \{\neg b_3, \neg b_4, b_5, b_6, b_7, b_8\}$$

a large negative reward is given by the global reward function, but not by the local reward, the agent could decide that the discrepancy is due to an interdependency between the two modules, and create a new merged module. This module would only have one "active" state:

$$\{b_1, b_2, \neg b_3, \neg b_4, b_5, b_6, b_7, b_8\}$$

At all other times, the smaller modules would be used, but when the agent reached a situation corresponding to the "active" state in the merged module, the smaller modules would be superseded by the larger one. The agent would thus avoid learning in the large state-space, except in those few situations when learning in the smaller state-spaces proves to be unsatisfactory.

In order to decide which modules should be merged to form a new module, it must be determined which modules are active, or relevant at any given point. If one

considers that each module is trying to learn where there are sources of positive or negative reward, one can imagine that modules for which reward sources are distant, should be less influential in determining the agent's action. Thus, modules where the current state only has actions with a utilities below some threshold value can be regarded as not relevant to the current situation. In figure 4 the third module is regarded as irrelevant, and thus does not contribute to the merged module (however, its learning algorithm is still active, allowing for the possibility that some reward is received that increases the utilities above the relevance level).

Though many details of this module merging mechanism need to be worked out, the above outlines a promising method by which the agent can gradually refine its decomposition as it discovers more about the structure of its task.

7 Conclusion

There has been relatively little work done on decomposition in reinforcement learning. Given the seemingly small amount of information provided to a reinforcement learning agent it is difficult to see how the agent could discover the necessary information for a successful decomposition. However, the prime source of information to the agent, its reward function, can provide hints as to what the possible subgoals are and how they interact. The reward function specifies the goal states, allowing us to find candidates for decomposition, and it may also give some indication as to which sub-tasks are independent, by the way it assigns reward to them. The agent can thus use this information to attempt to decompose the task. Decomposing necessarily leads to loss of global information, so the agent must thus use local approximations of the global reward function in order to learn the task correctly. We hypothesize that in many domains the agent will be faced with only simple situations most of the time, where a simple approximation algorithm such as greatest mass will perform well. For the exceptional cases where the approximation fails, we are developing a mechanism to gradually merge states of individual modules to provide a more detailed description of the failure state. The agent will then start of with as good an initial decomposition as can be attempted given the available information, and will then gradually refine the decomposition as it learns more of the structure of the task.

References

[Chapman and Kaelbling, 1991] David Chapman and Leslie Pack Kaelbling, "Learning from delayed reinforcement in a complex domain," In *Proceedings of IJCAI*, 1991.

[Kaelbling, 1989] Leslie P. Kaelbling, "A formal framework for learning in embedded systems," In *Proceedings of the Sixth International Workshop on Machine Learning*, pages 350–353, 1989.

[Karlsson, 1993] Jonas Karlsson, "Learning to solve multiple goals," Unpublished thesis proposal, May 1993.

[Mahadevan and Connell, 1991] S. Mahadevan and J. Connell, "Scaling reinforcement learning to robotics by exploiting the subsumption architecture," In *Proceedings of the Eighth International Workshop on Machine Learning*, 1991.

[Sing, 1992] S. P. Sing, "The efficent Learning of Multiple Task sequences," In *Proceedings of NIPS 9*, 1992.

[Tenenberg *et al.*, 1993] Josh Tenenberg, Jonas Karlsson, and Steve Whitehead, "Learning via task decomposition," In *Proceedings of the Second International Conference on Simulation of Adaptive Behavior*. MIT Press, 1993.

[Watkins, 1989] Chris Watkins, *Learning from Delayed Rewards*, PhD thesis, Cambridge University, 1989.

[Whitehead *et al.*, 1993] Steve Whitehead, Jonas Karlsson, and Josh Tenenberg, "Learning Multiple Goal Behavior via Task Decomposition and Dynamic Policy Merging," In S. Mahadevan and J. Connell, editors, *Robot Learning*. MIT Press, 1993.

[Whitehead, 1991] Steven D. Whitehead, "Complexity and cooperation in reinforcement learning," In *Proceedings of the Tenth National Conference on Artificial Intelligence*, 1991.

[Whitehead and Ballard, 1989] Steven D. Whitehead and Dana H. Ballard, "A Role for Anticipation in Reactive Systems that Learn," In *From Animals to Animats: Proceedings of the Sixth International Workshop on Machine Learning*, Ithaca, NY, 1989. Morgan Kaufmann.