Learning to Solve Multiple Goals

by

Jonas Karlsson

Submitted in Partial Fulfillment

of the

Requirements for the Degree

Doctor of Philosophy

Supervised by

Professor Dana H. Ballard

Department of Computer Science
The College
Arts and Sciences

University of Rochester
Rochester, New York

1997

Acknowledgments

I thank my advisor, Dana Ballard, for his unwavering patience, support, and assistance throughout my efforts on this work. As a constant source of enthusiasm, ideas, and insight, he allowed me to reach goals that would have been unattainable without his guidance. In addition, his willingness to read and comment on a seemingly endless succession of drafts of this document greatly helped to improve its scope, clarity, and readability. I would also like to thank my committee, James Allen, Robbie Jacobs, Henry Kyburg, and Josh Tenenberg, for their penetrating questions and comments, forcing me to think hard about the assumptions and motivations that lay as a basis for this work.

I am doubly indebted to Josh Tenenberg who first set me on the path of learning to solve multiple goals, and shares my excitement for the concepts and ideas involved. He has also proven to be a good friend and of a genuinely warm spirit. Steve Whitehead is the other collaborator on the initial work on learning multiple goal, and was the first person to get me interested in Machine Learning. To have been allowed the privilege of working with Josh and Steve is something for which I will always be grateful.

Designing and implementing a driving simulator, as well as running experiments, involves a lot of hard, sometimes tedious work. I was fortunate to have several helpers along the way. Andrew Kachites McCallum collaborated on all versions of the driving simulator as well as lending me the NeXTstation on which much of the initial work was done. Andrew McCallum was a kindred spirit from the beginning: *mon semblable, mon frère*. His presence here made all the difference, and I am greatly in his debt for his generosity. Tim Becker was also an integral part of the design and implementation of the Virtual Driving Simulator that was used in all experiments described herein. He taught me much about programming and design, and without his efforts none of this work would have been possible. He has also been a good friend and supporter throughout my presence at Rochester, and our hikes in the Colorado Rockies and our many long conversations will always be a highlight of my memories. Eric Ringger implemented some ideas for using lookahead, which I'm sorry we never had time to pursue and was always happy to give thoughtful responses to all sorts of wild ideas. I would also like to thank the undergraduates who developed small projects for me: Irene Krenskaya and the McNair Scholars: Shaun Gittens and Teresa Flores.

Though the department is always a hotbed of interesting projects, I would like to thank Chris Brown for organizing many interesting robot projects providing opportunities to tinker with real machines and play with legos, while exploring important research issues.

The administrative and technical staff of any organization provide the glue that keeps everything together. Pat Marshal, Marty Guenther, Peg Meeker, Jill Forster, Peggy Frantz, Liudvikas Bukys, Jim Roche, and Tim Becker were all of tremendous assistance in dealing with administrative and technical problems.

I could not have survived at Rochester without the support and companionship of my fellow students at the department. Karen Bogucz, Ricardo Bianchini, Brian Marsh, Bill Garrett, George Ferguson, Lou Hoebel, Justinian Rosca, Leo Hartman, and many others all provide many hours of fun, conversation, and shared frustration, making bad experience bearable and good experiences twice as enjoyable. Special thanks to my current office mates Maged Michael and Amit Singhal for letting me hog the office work station. Finally, Polly Pook was always an idol and a source of inspiration for me. Her accomplishments provided me with goals to strive for, and her encouragement, technical insight, and unique perspective helped me in my attempts to reach them.

Last, I thank my wife, Dragana. Her insight enhances my thought, her support strengthens my will, and her love brightens my spirit.

This material is based upon work supported by the U.S. Air Force - Rome Laboratory under Research Contract number F30602-91-C-0010, by NIH/PHS under Grant number 5 P41 RR09283, and by the National Science Foundation under Grant number IRI-8903582.

Abstract

In many domains, the task can be decomposed into a set of independent sub-goals. Often, such tasks are too complex to be learned using standard techniques such as Reinforcement Learning. The complexity is caused by the learning system having to keep track of the status of all sub-goals concurrently. Thus, if the solution to one sub-goal is known when another sub-goal is in some given state, the known solution must be relearned when the status of the other sub-goal changes.

This dissertation presents a modular approach to reinforcement learning that takes advantage of task decomposition to avoid unnecessary relearning. In the modular approach, modules are created to learn each sub-goal. Each module receives only those inputs relevant to its associated sub-goal, and can therefore learn without being affected by the state of other sub-goals. Furthermore, each module searches a much smaller space than that defined by all inputs considered together, thereby greatly reducing learning time. Since each module learns how to achieve a separate sub-goal, at any given time it may recommend an action different from that recommended by other modules. To select an action that best satisfies as many of the modules as possible, a simple arbitration strategy is used. One such strategy, explored in this dissertation, is called *greatest mass* which simply combines action utilities from all modules and selects the one with the largest combined utility.

Since the modular approach limits and separates information given to the modules, the solution learned must necessarily differ from that learned by a standard, non-modular approach. However, experiments in a simple driving world indicate that while sub-optimal, the solution learned by the modular system only makes minor errors when compared with that learned by the standard approach. A complex task can thus be learned very quickly, using only small amounts of computational resources, with only small sacrifices in solution quality, using the modular approach.

Table of Contents

List of Tables

List of Figures

1 Introduction

Many complex tasks can be subdivided into a set of sub-goals. Figure 1.1 shows a domain where the agent driving the car shown in the bottom right corner must navigate through city streets while avoiding obstacles, following street signs and negotiating streetlights.

The complexity of the task is a result of the the number of different types of objects that can appear in the world. Without any prior information about the layout of the world the agent must be prepared to handle all possible combinations of objects occurring in any given situation. Furthermore, it would be difficult to design a control program for an agent in this domain that could correctly handle all possible situations. A learning approach is more suitable, allowing the agent to gradually gain experience about what types of situations occur in the world and learn the appropriate course of action.

However, learning how to drive through the type of driving world shown remains difficult. Machine learning approaches that perform well on simple tasks typically become impractical to use once the complexity of the task increases. In the driving world each sub-task might be simple to learn in isolation; their combination dramatically increases the complexity of the problem. Since the core of the agent's computation in learning a task is search, it is clear why combining several tasks leads to this increase in complexity. Each task requires some state information (for example, there is an obstacle to the left, the streetlight is yellow, and there is an intersection ahead) all of which must be combined to fully describe any given state of the world. The more sub-tasks are present, the larger the space of possible world states, and possibly the number of states that must be searched before a solution is found. Furthermore, the agent will be forced to relearn solutions to sub-tasks when it is not necessary. For example, if the driving agent above learns to turn to follow a street sign while there is an obstacle ahead, it must relearn how to follow the sign if there is no obstacle in view, or if the obstacle is in a slightly different position. That is, any time the configuration of sub-tasks differs from what has been previously experienced, the agent is in an apparently completely new part of the search space, and must relearn how to achieve all of its

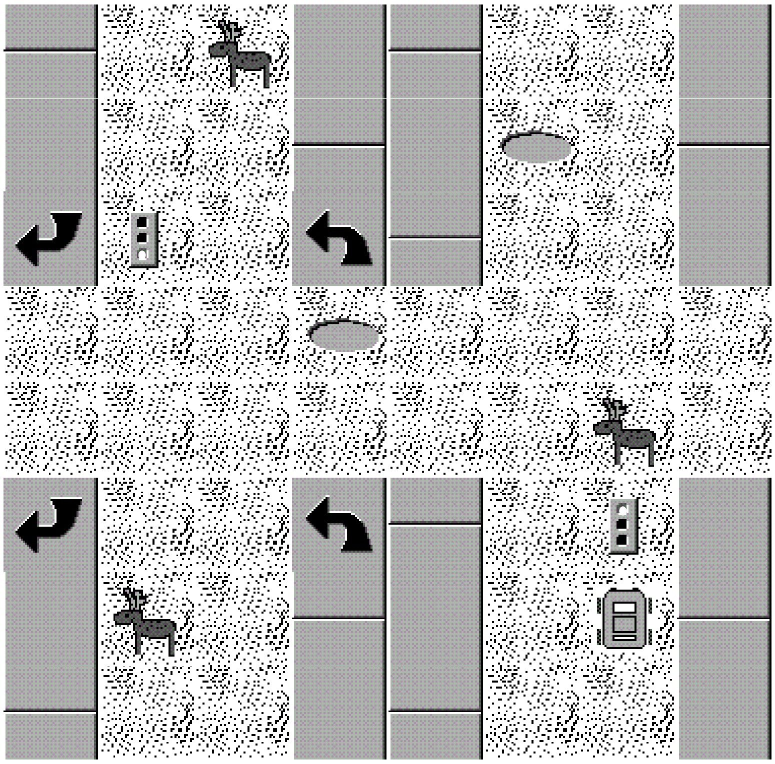


Figure 1.1: A simple driving domain. Though the agent's task is complex it can be decomposed into sub-tasks. The modular reinforcement learning architecture takes advantage of such a decomposition to learn a task that would be intractable for other, monolithic methods.

sub-tasks. In some cases it may be appropriate to relearn the sub-task. Avoiding an obstacle might require different types of maneuvering depending on whether or not the agent must also make sure that it does not go through a red light in the process, but often the previously learned solution will be appropriate.

This thesis focuses on the driving task described above (and specified in more detail in Chapter 5). As such, it represents an instance of a class of tasks and domains that are characterized by allowing themselves to be decomposed into fairly independent sub-tasks. Other examples are planetary rovers that must concurrently navigate a terrain, avoid obstacles, and gather samples while moving towards a goal location, or an office robot that delivers mail, keeps printers stocked with papers, and recycles soda cans. For these types of domains, the agent designer often has good domain knowledge regarding how the overall task separates into largely independent sub-tasks. This thesis investigates an approach using this domain knowledge to set up a learning framework for the agent so that unnecessary relearning can be avoided. The approach extends a form of *reinforcement learning* with a modular architecture that learns all sub-tasks independently but concurrently. The learned solutions are combined through a simple arbitration strategy, which selects the action to execute at each given step. Each sub-task is learned by a module that receives only those inputs relevant to its task, thereby avoiding the relearning caused by a change in irrelevant inputs. The approach is described in detail in Chapter 4.

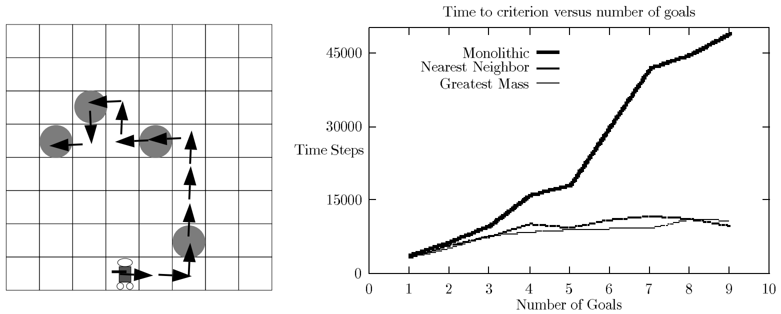
Reinforcement learning is used as the basis for our approach because the *a priori* knowledge required by the agent designer consists only of a good specification of the task to be learned, little or no information about the effects of actions, and no detailed semantic information about the agent's inputs. This type of domain knowledge corresponds to what is available in domains with multiple tasks as we described above. Given the driving domain, it is known that the task consists of driving on the streets while acting appropriately when faced with obstacles, streetlights, and other objects in the world. This knowledge can then be encoded as a *reward function* which is used as the learning signal in reinforcement learning. The reward function encodes task information by providing a scalar reinforcement value at each state, evaluating the agent's progress towards the given task.

Though information about the task might be available, the agent designer may not have good information about the effects of actions in all possible states or what action sequences actually accomplish the task. Using reinforcement learning, the agent can learn this information. Initially, the agent must explore its domain by blindly executing actions and observing the results. As reward sources are discovered, the agent starts to associate actions with utilities that indicate their value towards accomplishing the task. As the agent continues to explore its domain, it gradually refines its solutions to become more efficient. However, because of the amount of exploration necessary to both initially discover reward sources, and to find the most efficient way to reach them, the learning process can be quite slow.

Reinforcement learning can thus be characterized as a search in a space whose size is determined by the complexity of the domain. As we described above, adding sub-tasks can cause the size of the search space to increase dramatically.

In [Whitehead *et al.*, 1992] we illustrate the utility of the modular approach in a simple grid world domain. The domain is an $m \times m$ grid, with n cells designated as *goal locations*, as shown in Figure 1.2 (a). Goals are either active or inactive, with an active goal becoming inactive when visited by the agent. An inactive goal turns active with some probability at each time step. The agent can move horizontally and vertically, and has sensors providing it with its (x, y) coordinates as well as a bit vector indicating the activation status of each of the goals. The size of the state space is then $O(m^2 2^n)$, which can become very large, even for relatively small m and n. In experiments with a 20×20 grid and 10 goal locations, it would have been almost impossible for a standard reinforcement learning agent to learn the task. A modular approach to reinforcement learning was therefore proposed as a way by which the task could be learned. The modular approach takes advantage of the structure of the task by using separate modules to learn each sub-task. Each module considers only those inputs necessary for its sub-task (the activation status of the sub-goal and the agent's current location, in grid world) and receives reward only when its sub-task has been achieved. By limiting the inputs to each module, the size of a module's state-space is considerably smaller than the space defined when all inputs are considered together. Learning time is therefore reduced, and since all modules learn concurrently, but do not depend on each other, adding more goals to the task does not greatly affect performance. An approximation algorithm is used to combine the actions recommended by each module, selecting the single action that is most appropriate for all sub-tasks combined. Figure 1.2(b) shows the number of time steps needed by a monolithic agent and by two modular agents using different approximation strategies, to reach a predefined performance criterion. As the number of goals increases, the learning time increases almost exponentially for the monolithic agent, while increasing very slowly for the modular approach. If the number of goals increases further, learning the optimal paths between all possible combinations of active goal locations becomes infeasible. However, the modular system rapidly learned a solution that was a good approximation to the optimal solution.

The simple grid world example is a good example of the type of multi-goal domain for which our modular approach to reinforcement learning is intended. The problem is equivalent to the NP-complete Hamiltonian Path problem, meaning that learning the optimal solution is impractically expensive for large problem sizes [Garey and Johnson, 1979]. However, the sub-goals are clearly identifiable, and the associated reward functions and sensory inputs can be decomposed into components relevant to individual sub-goals only. Furthermore, in grid world, we can be satisfied with an approximate solution. The modular agent learned a "nearest neighbor" solution, which is known to be a good approximation for



(a) (b)

Figure 1.2: A simple grid world. The agent in the bottom row must visit all four "active" goal states (shown as grey circles) using as short a path as possible. As the size of the grid and the number of goal locations increases, the learning time using standard reinforcement learning becomes prohibitive. The arrows indicate the path learned by the modular policy.

the Hamiltonian path problem [Johnson, 1990]. A drawback to the grid world domain is that all goals are goals of achievement. There are no negative sources of rewards, or states the agent must learn to avoid. Therefore, an arbitration strategy designed to learn a nearest neighbor solution would necessarily perform well. However, in a more realistic domain, the reward structure is more complex, containing both positive and negative reward sources. In such domains, a nearest neighbor approximation would be less useful. Thus the grid world is too simple to exhibit the complexities of learning multiple goals. To extend this we developed a SImple Driving domain (SID) to explore these more complex issues. The different sub-tasks are different aspects of driving down a street, such as staying in the right lane and avoiding obstacles. In addition to the task being easily decomposed, the driving domain is one where an approximate solution is acceptable. As long as the agent avoids major collisions, occasional small errors such as running over pot holes, going through yellow lights, or driving in the wrong lane, can be forgiven. It may be possible to eventually learn the optimal way to traverse the domain, making no mistakes. However, the computational resources necessary to learn this solution makes it impractical to attempt it.

The modular approach is thus intended for domains where

- The task is too complex to be solved by a standard reinforcement learning system.

- Sub-tasks are identifiable *a priori*.

- It is clear which input sensors are relevant to which sub-tasks.

- The reward function can be decomposed into a set of independent reward functions, each of which evaluates a different sub-task.
- An approximate solution must be acceptable, since the optimal solution is unattainable.

1.1 Statement of thesis

This thesis defines a modular extension to the reinforcement learning algorithm known as Q-learning [Watkins, 1989]. By using modules, a complex task can be decomposed into a set of independent sub-tasks. Each module is devoted to learning one sub-task, and all modules learn independently and in parallel. Since learning how to accomplish one sub-task is separated from learning the other sub-tasks, the above described problem of having to re-learn sub-tasks does not occur. However, while separating the sub-tasks avoids unnecessary relearning, it also prevents *necessary* relearning. That is, in some situations sub-tasks interact in such a way that solutions that are sufficient independently do not produce the desired results when the sub-tasks are combined. It is the thesis of this dissertation that for the types of decomposable task we describe an agent using the modular approach with a simple arbitration scheme to select which action to execute based on information provided by the modules, can learn a good approximation of the optimal policy in a short time. Whether the approximation is good enough to be usable will depend on the specific task to be solved, but with complex enough tasks, the optimal solution is unattainable in any reasonable amount of time.

1.2 Contributions

The contributions of this dissertation are as follows:

- The development of a simulated driving domain as a test bed for agents learning to solve multiple goals.
- Experiments showing that a modular approach to reinforcement learning can be used to learn a strategy that achieves satisfactory performance.
- An extension to the modular approach using an activation function that dramatically improves performance to a level that in some cases surpasses that of the monolithic approach.
- An analysis of the modular approach that illustrates its limitations caused by hidden state, perceived shorter paths to reward, and approximation errors caused by combining policies for different sub-goals.

1.3 Outline

In Chapter 2 other techniques for learning tasks decomposed into sub-goals are described and classified according to what type of decomposition they are designed for.

Chapter 3 provides a brief review of reinforcement learning, focusing on the Q-learning algorithm. We subsequently describe in some detail different issues in designing an appropriate reward function, especially in a multi-goal domain. Since the reward function is how agent designers control how the agent should behave, it is important to specify it correctly. In a multi-goal environment where different goals receive different rewards of different magnitudes, it is especially difficult to predict the resulting behavior.

The modular approach to Q-learning is described in Chapter 4, with a brief analysis of the limitations of the approach. Though a numerically exact approximation is not necessary for the modular approach to produce optimal behavior, it is clear that an agent using modules only will learn a policy that occasionally makes mistakes. We describe some extensions to the modular approach, including the use of an activation function that can limit the number of sub-tasks being considered at any give time.

In chapters 5 and 6 we describe the simple driving domain used for experiments and present the results for the modular approach and its extensions. Though the experiments show that the significant differences between the modular policy and a policy learned using standard reinforcement learning are limited to the expected small set of states, the proposed extension to the modular system is largely unsuccessful in correcting those differences. Only when we can apply more domain knowledge to determine when modules have relevant information do we see marked improvements in behavior.

We further analyze the experiments with the modular approach in chapter 7 and describe some of the inherent limitations of the algorithm preventing it from learning the optimal policy.

In chapter 8 we conclude and map out possible future areas of research to more fully understand the modular approach, and to discover better ways to augment it.

2 Related Work

The problem we are investigating is that of achieving multiple goals by arbitrating between and combining separate, task specific, control modules. We assume that these modules are *independent* and can be *concurrently active*. The modules are independent in that the reward (in a reinforcement sense) they receive, does not depend on the state of any other module. Furthermore, the reward is relevant to that module only, in that it is a measure of how well the module is achieving its local task. Since the modules are independent, several can be active at the same time. This leads to the problem of arbitration — selecting which module should control the agent at any given instance. More interestingly, it is possible to combine the policies of several active control modules, to generate actions that satisfy several modules a the same time. This modular architecture is defined in detail in Chapter 4.

Below we discuss previous approaches to task decomposition and policy arbitration. These approaches are often described as being *behavior based*, because of the emphasis on the set of behaviors that need to be satisfied, rather than on a global goal which has been subsequently decomposed into sub-goals. We classify them into hierarchical, sequential, and independent approaches, depending on how they organize modules or behaviors. In contrast to our approach, most of them only allow one module to be active at a single time, and many do not involve learning the sub-tasks. Last, some of the approaches described below do not subdivide the state-space, but require learning in the entire global search-space.

2.1 Hierarchical approaches

Many complex tasks seem to lend themselves to a hierarchical decomposition. For example, buying groceries can be decomposed into the sub-tasks of driving to the grocery store, purchasing the needed items, and driving back home. Each of these tasks can then be further decomposed into more sub-tasks, for example pulling out of the drive-way, driving to the main road and on to the store, finding a parking space, and parking. Finally, at the lowest level, the tasks correspond to actual motor commands. Though the total amount of information that needs to be considered by the system as a whole may be staggering, each level of the hierarchy need only concern itself with a small amount of that information. When deciding how to drive to the store, the system need only consider navigational type information, and disregard sensory feed-back from the gas-pedal, for example. Similarly, it only needs to consider the behaviors possible at that level, and not low-level motor commands. Also, low-level behaviors need not be concerned with high-level information but can focus on tasks such as "move forward 1 meter". The information needed at each level is therefore limited, making the search-space smaller.

Brooks' *subsumption architecture* is a hierarchical behavior-based approach [Brooks and Connell, 1986],[Brooks, 1985]. Lower levels control "instinctive" behaviors and higher levels control tasks that are regarded as more abstract. In a mobile robot for example, a low level behavior might be obstacle avoidance, while a high level behavior controls path planning. Each behavior consists of a fixed finite state machine, that generates an action based on the current inputs. An obstacle avoidance behavior thus has a state where an obstacle is detected, which leads to an "avoid" state that generates a new heading for the robot, until the obstacle is no longer a danger. A higher level behavior, searching for soda cans to recycle, for example, would also be implemented as such a finite state machine. Since both behaviors attempt to control the heading of the robot, which can only move in one direction at a time, the obstacle avoidance behavior will set a status flag to "busy", when it needs to control the agent in order to avoid a collision. The search behavior will defer to the lower level behavior when the "busy" status is set. However, high level behaviors can influence low-level ones through *subsumption links*. These links can either inhibit the lower levels completely, or influence their behavior. The search behavior can therefore influence the direction in which the obstacle avoidance behavior turns away from an obstacle, perhaps also moving it closer to a soda can.

In the subsumption architecture, the hierarchy reflects authority rather than any notion of abstraction. That is, each behavior uses some subset of the inputs, but the inputs used by low-level behaviors are not a super-set of those used by high-level ones. Thus there is no sense in which the high-level modules use more abstract information. Instead, one module is higher on the hierarchy if it needs to influence another module. The hierarchy is therefore more of an artificial limitation on what types of interactions are allowed between behaviors. It is possible that for some tasks more complex interactions should be allowed, with modules influencing each other, or one module influencing modules that are both above and below it in the hierarchy.

The large limitation of the subsumption architecture is its complexity. Though each behavior may be implemented as a simple state-machine, creating correct subsuming and inhibitory links into other behaviors requires detailed knowledge of how the other modules, and the system as a whole, function. In order to implement the search behavior described above, the designer must know exactly where in the obstacle avoidance behavior a link to influence the robot's heading will be effective. The designer must also consider whether there are other behaviors also attempting to influence the avoidance module, and whether that will affect the system's overall behavior. Furthermore, since there is no centralized control, there can be no arbitration to determine which behavior is actually controlling the agent's actions. This arbitration seems to be handled by using status variables such as "busy", set by the individual modules. Without a global overview over which modules are simultaneously competing for resources, however, such a system will rapidly become very complex, and highly dependent on the exact configuration of behaviors.

Thus, the subsumption architecture depends on the designer to have a large amount of knowledge of how individual modules achieve their sub-tasks, and what the possible interactions are. Since abstraction is not taken advantage of, the hierarchical organization brings little advantage and may be too limiting for some tasks.

In contrast, Spector and Hendler discuss a hierarchical approach called "knowledge strata" [Spector and Hendler, 1990] that makes explicit use of the hierarchy. There can be several modules on any level of the hierarchy, each operating in parallel. Modules communicate with each other, *and* with the levels immediately above and below, through separate storage locations called blackboards. The modules do not need to know about each other's internal construction as in the subsumption architecture, but simply reason about the information available on the blackboards. Since there are separate blackboards at each level of the hierarchy, different types of information are kept separate, limiting the amount of reasoning that needs to be done. Each behavior can be implemented in whatever fashion is most suitable, whether they be neural-networks or full-blown planning systems (as long as they can be used with the blackboards). The hierarchy is fixed into levels corresponding to different parts of an event structure. The highest level is the *conventional* level, that is intended to reason about facts which are true by convention. This is meant to indicate high-level goals, such as empty soda cans "belonging" in a recycling bin. The remaining levels are *causal*, *temporal*, *spatial*, and *sensory-motor*. The names indicate what type of knowledge and goals are handled at that level. Path-planning goals reside in the spatial level, for example, whereas the sensory-motor level only reports direct sensor inputs and contains operators for manipulating the agents actuators.

By limiting the set of operators at each level to those needed for the associated types of goals (`above`, `below`,or `on`, for example, at the spatial level), any type of search or planning done by modules is also limited, allowing the system to take full advantage of the decomposition. Furthermore, the lowest level controls the actuators, and thus implements the final decision on how they are used, avoiding the problems with resource allocation in the subsumption architecture. However, modules at the same level must still be aware of each other, in order to avoid mutual interference. There is no clear mechanism to arbitrate between competing goals (eg. wanting to move towards two different locations). Presumably, this arbitration occurs via communication on the blackboard, requiring each behavior to be aware of what other behaviors exist that might cause interference.

Thus, as with the subsumption architecture, individual modules must have good knowledge of the operations of the other modules. The knowledge necessary for these approaches thus concerns *solutions* to the individual sub-problems, rather than simply information about the problem itself. We are considering domains where solutions are unknown and must be learned. Approaches such as the subsumption architecture and knowledge strata that require modules to be programmed to, in effect, modify each other's solutions would therefore be difficult to apply.

Wixson [Wixson, 1991] proposes a mechanism to use a hierarchical organization of behaviors in reinforcement learning. The hierarchy is created by defining modules completely responsible for a subset of both inputs and outputs. To create a module i, the agent designer must identify a sub-task that is achieved by starting in some set of initial states, I_i and reaching some set of goal-states G_i, using only a subset A_i of the total actions available. The input-space is also decomposed, so that only a subset of the inputs S_i is used by the module. The actions A_i and inputs S_i are then unavailable to any other module, but the action $\mathtt{invoke_i}$ is added to the set of actions. Thus, another module can use the $\mathtt{invoke_i}$ action effectively using module i as a building block towards the solution of its own sub-task. When the $\mathtt{invoke_i}$ action is executed, the "calling" module is suspended, and the "called" module takes control and learns using the reward received. Once the sub-goal has been achieved, the calling module is resumed, with the total reward received by achieving the sub-task used as the reward for executing the $\mathtt{invoke_i}$ action.

This approach mirrors the concept of abstraction in classical planning ([Sacerdoti, 1974], [Sacerdoti, 1975], [Tenenberg, 1988]), where information is limited in higher levels using lower levels as building blocks. Though the approach appears simple, creating elemental building blocks out of frequently used sub-goals, the difficulty lies in identifying appropriate sub-tasks. Also, by not allowing modules to share inputs and outputs, the number of sub-tasks that can co-exist are severely limited. In domains where the only actions are navigational ones for example, one is limited to sub-tasks such as "go east or north to the goal" and "go west or south to the goal". While this type of decomposition may reduce the amount of search necessary to learn the goals, it does not reflect a natural decomposition of a task into sub-goals. Rather the decomposition is in the solution space, ie. the agent designer must decide what types of actions can be grouped together to form part of the solution. Furthermore, though the author intends for modules to be reusable by many other modules, this is not possible in the algorithm as stated. Once $\texttt{invoke}_\texttt{i}$ is used by one module, it is removed form the set of actions available to the creation of others. Lastly, the approach is highly tailored towards goals of achievement, decomposed into a sequence of sub-goals, with a clearly defined set of initial and goal states. It does not seem to be extensible towards domains with a more complex reward function, allowing partial goal-satisfaction.

Another approach to using learned sub-tasks as building blocks for learning higher level tasks is discussed in [Lin, 1993a]. A robot must learn to recharge its battery by finding a battery charger in another room. The state-space is large and continuous, thus making the problem too difficult for an agent not decomposing the task in some way. The agent designer decomposes the task into 3 sub-tasks: wall-following, passing through a door, and docking with the battery charger. Each sub-task is solved by a module with a pre-defined reward function and *application space*. The application space is not defined by a subset of the possible inputs, but is rather a subset of the state-space as a whole. It is thus analogous to a *precondition*, that indicates under which circumstances a sub-task must be achieved. The application spaces may overlap, and the elementary tasks are learned by training them separately in the corresponding application spaces. Once the elementary tasks have been learned, the agent must learn to apply them in order to achieve the global task. In contrast to the approach proposed by Wixson, elementary skills do not need to have a clear termination condition. The agent must therefore learn how to switch between its sub-tasks. This forces the agent to make a decision at each time-step, since a switch may be necessary at any given time. Thus the agent is learning in the monolithic state-space, but using the sub-tasks, rather than individual actions. This learning succeeds where monolithic learning does not, which is attributed by the authors to the fact that the agent must switch skills less often than switching between primitive actions. However, it seems that much of the performance improvement must be the result of the number of elementary skills being less than the number of actions, and the fact that the sub-tasks are already learned, making a complete solution easier to find.

Curiously, in this approach the definition of a module's application space is presumed to be unknown when learning how to compose the sub-tasks. Since this knowledge must be known in order to define the modules, there is no reason why it should not be applied in later stages of learning, thereby facilitating the process. The agent would then only have to learn to switch between modules whose application spaces overlap. Using the application spaces in this way, the method becomes similar to our extension to the modular approach using activation functions described in Section 4.3.2. However, the approach assumes that the task is decomposed sequentially, and there can be no concurrently active sub-tasks that can cooperate.

Decomposing a task hierarchically is a good way to limit the search-space that must be considered at any given level. However, as shown by the subsumption architecture and knowledge strata, the way hierarchies can be constructed often seems arbitrary. However, even when the hierarchy is strictly based on what sub-task can be used as a building block for another sub-tasks, there is no clear mechanism to allow sub-behaviors to be composed. Either the system requires a large amount of knowledge of how each module accomplishes its sub-task, or the main focus of the system is to simply switch between a sequence of sub-tasks. Since the modular approach we propose does allow for multiply active sub-goals, and requires only knowledge about the structure of the task, it is possible that it can be used within one level of a hierarchy.

2.2 Sequential approaches

If there is no clear way to decompose a task hierarchically, it is often possible to decompose it as a sequence of sub-tasks. At any given time, the agent only needs to try to accomplish the next sub-task in the sequence, rather than the complete task. Furthermore, the only arbitration between modules that is necessary is to determine when one sub-task is complete and the next one should be started. Since it is known in advance what the sequence of sub-tasks is, how to design such an arbitration strategy will also usually be known.

Mahadevan and Connell use a subsumption based design for an agent learning to find a box and push it against a wall [Mahadevan and Connell, 1990]. The task is divided into finding a box to push, pushing it against a wall, and "unwedging" from the wall. Each task is assigned to a module consisting of a transfer function and an applicability condition. The transfer function determines how the agent should act given the current sensory information. This function is learned using reinforcement learning, with a separate reward function for each behavior. The applicability condition determines whether the action output by the module's transfer function should be executed or not. The applicability condition is defined by the agent designers and is also similar to the activation function we describe in Section 4.3.2. The applicability conditions are not aware of other modules, and can thus not prevent several modules from being active at the same time. To ensure that only one module is in control at any given time, subsumption architecture suppression links are used to implement a priority ordering. Thus, unwedging suppresses box-pushing, which in turn suppresses box-finding. This prioritizing establishes the sequence of find a box, push the box against a wall, and unwedge to be free to search for a new box, that makes up the entire task.

The described architecture would allow each module to limit its inputs, reducing the size of the state-space, so that learning could be accomplished more quickly than with a monolithic learner. The authors do not perform any such limitation however, but rather use a "statistical clustering" algorithm to combine sets of similar states. The search-space then consists of the set of clusters which is much smaller than the set of all states. Since the similarity metric used for clustering includes the utility of taking an action, the clusters found by each module will differ from each other, and those of a monolithic agent. Unfortunately the authors do not provide any analysis of how the modular and monolithic agent differ in terms of the clusters found and the size of the resulting search space. Furthermore, in the described experiments the monolithic agent is given a reward function that provides much less information than that given to the modular agent. It is therefore difficult to assess how much the performance difference between the agent is the result of modularization, and how much is caused by the different reward functions. A different clustering method using only a weighted Hamming distance leads to similar results, suggesting that any performance difference is caused mainly by the difference in reward functions.

It might not be possible to use the simple priority ordering described above to control the sequence of sub-task if the decomposition is more complex than three behaviors, or the exact sequence is unknown. In [Maes, 1989], Maes describes an architecture that allows for more complex types of interactions between modules. Modules are arranged as nodes in a small network. A node can be linked to another node if executing the first node causes a precondition of the other node to be either true or false (making the node either excitatory or inhibitory). These links are used to spread activation through the network. The activation is generated by the current state of the world and unsatisfied goals. The current state activation is sent to nodes whose preconditions are being met by the current state, and then spreads forward through the excitatory links. Similarly, nodes whose execution would accomplish unsatisfied goals are also given activation which is then spread backwards through the network. Out of those nodes whose preconditions are all satisfied in the current state, the node that first reaches a pre-defined threshold, is allowed to execute. This network architecture allows for a sequential decomposition of a complex task, where the ordering of sub-tasks need not be completely specified. However, a large amount of domain knowledge is necessary to determine how sub-task interfere with each other so that the network can be constructed.

In later work, Maes has designed methods by which the topology of the network of links is learned, based on past experience [Maes, 1992]. Statistical methods are used to correlate actions with results corresponding to preconditions of other modules. Positive correlations lead to links being created and negative ones lead to the deletion of links. These correlations must be learned in the monolithic state-space however, requiring the large computational resources that modular approaches are intended to avoid. Furthermore, since explicit links between modules that interfere or assist one another are required, yet another combinatorial problem must be addressed. If learning to achieve the sub-tasks is desired in addition to learning the structure of the network, it seems likely that the task would become intractable with even only a small number of modules.

Singh proposes an architecture that reuses learned elemental tasks to learn arbitrary sequential tasks [Singh, 1992]. Elemental tasks are learned using Q-learning and are defined to accomplish goals of achievement, with only one goal state. A positive reward is given in the goal state of an elemental task and nowhere else. Composite tasks consist of a sequence of elemental tasks. A set of "augmenting" bits specify the task sequence that is to be achieved. For example, if the elemental tasks are to reach locations A, B, and C, a composite task might be to first goto location C, and then to A. A particular configuration of the augmenting bits denotes that this is the current task to be solved. The elemental tasks are learned by modules who receive the same perceptual inputs. A *scheduling module* is used to determine which module controls the agent at any given time and therefore receives any reinforcement signal. The scheduling module is adapted from the modular gating architecture by Jacobs, described below. It learns which module produces the best estimate of the Q-values for the current sub-task. The winning module receives the reinforcement, thereby learning to improve its utility estimate. The scheduling module thus partitions the state space into regions each of which is best handled by one module.

Singh's approach is intended for situations when elemental tasks need to be reused in many different composite tasks. If the agent was instructed to accomplish several composite tasks, each consisting of a different sequence of elemental tasks, a monolithic approach could not use any previously learned knowledge when attempting a new composite task. This modular approach allows the elemental tasks to be learned separately and be used to achieve the composite task. The scheduling module learns how to order the elemental tasks, which also continue to learned better and better.

However, the limitations on the structure of the task and reward function severely constrain the types of domains in which the algorithm is applicable. Only goals of achievement are allowed and there is no possibility of specifying that a goal can be partially satisfied. The strength of this algorithm is the ability to learn the task decomposition. Though this incurs extra learning time, it is still an advantage over a pre-defined decomposition (as long as the extra time needed is still significantly less than the time needed by a monolithic algorithm). However, since the algorithm assumes that the goal state of each sub-task be known in advance (so that the reward function can be properly defined), the task decomposition must effectively be part of the agent designer's knowledge.

In the sequential algorithms described above, the module decomposition depended entirely on separating the task into a sequence of sub-tasks. However, if the sequence is known *a priori*, there does not seem to be much need to learn how the sub-tasks are to be ordered, as is done in Singh's method. Maes' behavior networks learn how to order themselves based not so much on a pre-defined sequence of tasks, but depending on interactions between sub-goals that may not be known in advance. Even in that case however preconditions and effects of behaviors are known in advance making the task amenable to some planning algorithm or other off-line method.

All the described methods could benefit from limiting the state-space of each behavior (if appropriate) in order to improve learning time as compared with a monolithic approach. Decomposing only to limit what task is currently being attempted does not by itself simplify the task. It is only if the agent must repeatedly learn different sequences of the same sub-task that modularity brings any significant advantage (as in the experiments described by Singh), since learned knowledge can be reused.

2.3 Independent behaviors

Not all tasks can be decomposed hierarchically or sequentially. In many domains, the task consists of a set of independent sub-tasks that do not need to be achieved in any particular sequence. In these domains, the agent must learn to arbitrate between tasks, as well as learn to achieve the tasks themselves. At any given time, the agent must decide which of several active sub-tasks should be attempted, or whether there is some course of action which brings a group of them closer to completion. The agent thus needs an arbitration strategy as well as the knowledge necessary to accomplish each individual sub-task. Different approaches for artificial agents differ not only in the types of methods used for both of these parts, but also whether to use learning in either one.

Firby's reactive action packages (RAPS) use no learning for either the behaviors or arbitration strategy. Each behavior is handled by a RAP that encodes the knowledge necessary for achieving a task in several different ways. Each RAP is placed in a queue, and the arbitration is a simple "first-come, first-served" approach. When a RAP gets executed it either places new RAPs at the end of the queue or executes a primitive action. Each RAP detects whether it has succeeded and removes itself from the queue if its sub-task is achieved. In case of failure, the RAP places itself back at the end of the queue, to attempt to achieve the goal again, using a different method.

Firby's method emphasizes execution monitoring rather than learning behaviors or arbitration schemes. A lot of prior knowledge is required to design RAPs that contain several different solutions for one goal, and the ability to monitor the success or failure of each. Similarly, using a queue to decide which RAP to execute may not work well where the sub-task interact with each other and penalizes behaviors which call several sub-RAPs (since they are placed at the end of the queue).

Maes and Brooks designed a six-legged robot that learned the arbitration strategy for its behaviors [Maes and Brooks, 1990a]. Their architecture contains a set of binary perceptual conditions, a set of behaviors, a positive feedback generator, and a negative feedback generator. Each behavior has a precondition list testing the status of a set of perceptual conditions. A behavior may become active and perform some action if all of its preconditions are met. The feedback generators are binary and global so that at any given time all behaviors receive the feedback. The systems learns to gradually change the list of preconditions for behaviors so that behaviors only become active if they are *relevant* and *reliable*. A behavior is relevant if it is positively correlated to positive feedback, and is reliable if it receives consistent feedback. The agent learns to activate behaviors by monitoring how the preconditions affect the behavior's consistency and relevance. If the behavior receives positive or negative feedback in an inconsistent way, a new percept is added to the precondition list. The agent then monitors how well the new precondition correlates with positive or negative feedback. If a positive correlation is found, it is added permanently to the list of preconditions.

In the six-legged robot each leg has a move-forward and a move-backwards behavior, giving the agent a total of 12 behaviors. Positive feedback is received if forward motion is detected, and negative feedback results from the robot falling on its belly. At any given time the agent has to select which behavior affecting an actuator (ie. a leg) to execute and learn how to coordinate the motion of all six legs. Among the active behaviors, the agent chooses the one that is most relevant and reliable. With the above learning strategy the agent successfully learns to walk in a fairly short time (dependent on whether additional heuristic are used to determine in which order to monitor the perceptual conditions).

The above method works well because reinforcement is immediate and the individual behaviors do not need to be learned. It is therefore easy to compute correlations between various conditions and the agent's performance. It is also ideally suited for the task of coordinating multiple actuators, since despite the presence of a large number of behaviors, only two compete for any given leg. Furthermore, the behaviors are in direct opposition to each other (move forwards and move backwards) making the strategy of selecting one with the highest promise of generating positive feedback appropriate. However, in domains where several behaviors are using the same actuators this type of competitive approach may not be appropriate. Allowing behaviors to cooperate by searching for actions to further several sub-tasks may produce better results than simply trying to achieve each sub-task one at a time. Furthermore, as we discuss in Chapter 7 a "winner-take-all" strategy based on what behavior leads to the most positive feedback will ignore other behaviors that avoid negative feedback.

In contrast, Mataric uses a reinforcement learning approach to learn how to coordinate behaviors, allowing the use of more complex reward functions [Mataric, 1994]. The domain consists of a workspace of multiple robots gathering pucks and transporting them to a home location. At any given time the agent can chose between three behaviors: *search*, *disperse*, and *home*. The agent has three other behaviors, *avoid*, *grasp-puck*, and *drop-puck*, which are triggered automatically under the proper conditions (eg. if a puck is within the robots gripper it is grabbed, and if the robot is carrying a puck and is in the home location, the puck is dropped). The agent's reward is decomposed into a set of functions, and are classified as either rewarding a goal directly, or estimating the agent's progress. The first class rewards the pre-coded behaviors, giving positive rewards when a puck is grasped or dropped in the home location, and negative reward if a puck is dropped in some location other than home. The progress measurements are triggered by an external event (detecting another robot or grasping a puck), and give positive feedback if the agent moves in the right direction (away from the other robot or towards home) and negative feedback otherwise. After a set period of time, the agent is forced to switch behaviors and the monitoring is turned off. The separate reward functions are added together and used as the learning signal to learn which behavior to use at any given time. This learning is accomplished by simply keeping a cumulative sum of the reward received in each state, given which behavior was active.

Since the agent is learning when to activate behaviors, it must necessarily learn in the global state-space. The presented benefit of this work is the speed of learning achieved by using the progress monitoring reward functions. Though it is clear that the more information is provided through the reward function the faster the agent will learn, since this method of learning when to activate behaviors does not limit the state-space at all, it will not be practical in more complex environments. In the described domain, there is not much difference between the behaviors, so that it is possible to find a puck even if the agent is not explicitly executing a search behavior, for example. Thus learning is simpler since several behaviors can lead to good performance at any given time. Furthermore, since the learning scheme only counts cumulative reward for any given state, there is no mechanism for domains where the reward is delayed. While this work illustrates the benefit of using prior domain knowledge encoded as reward functions, simply using reinforcement learning to learn how to arbitrate between behaviors is probably not feasible in most complex domains.

Arkin proposes an arbitration scheme similar to our greatest mass technique [Arkin, 1990]. Each behavior is defined as a *schema*, implemented as a potential field. These fields can be easily added to each other resulting in a field taking information from all behaviors into account. Thus the arbitration scheme is a simple vector sum. However, each behavior has an associated weight, which the agent can learn to vary depending on the current situation.

The task domain is a simple navigational task so the potential fields can have a topology similar to the actual geometry of the work-space, with valleys pointing towards goals and peaks representing obstacles. Arkin discusses two possible learning schemes. The first modifies the different weights of behaviors using genetic algorithms. After 200 generations the number of collisions and number of steps required to reach the goal are both reduced significantly. This result demonstrates the value of dynamically modifying the weights on the behaviors to achieve a good performance. Arkin also proposes an online rule-based mechanism to modify the weights. Rules monitor certain pre-defined conditions such as `no-progress-with-obstacles` and `no-progress-no-obstacles`. Depending on which condition is triggered, weights of certain behaviors are increased or decreased. For example, if the system detects that little progress has been made towards the goal, but there are many obstacles nearby, the goal behavior's influence is lessened, while the obstacle behavior's influence is increased. This allows the agent to escape from the obstacles (which may have been creating a local minimum in the potential field) before moving towards the goal again.

The technique of using a vector sum to compose behaviors is simple and intuitive and also used successfully in our greatest mass approximation. It depends on being able to encode behaviors in a numerical form, such as potential fields or Q-values. Predefined potential fields as used by Arkin, are simple to generate in a navigational domain, but not in other tasks where the utility space being navigated has an unknown topology. The great improvements in performance achieved by learning how to weight behavior illustrates the importance of getting the relative magnitudes of the fields correct. Unfortunately the methods presented for learning these weights are probably not widely applicable. Genetic algorithms typically require a large number of test cases over several hundred generations, and the rule-based driven learning requires an amount of domain knowledge that usually will not be available.

The above methods are all similar in that they assume that the individual behaviors are pre-programmed and limit themselves to learning how to arbitrate between them. Jacobs and Jordan have developed a neural network architecture that learns not only the arbitration, but how to decompose the task into sub-tasks, and learn each individual sub-task as well [Jacobs and Jordan, 1991]. Arbitration is done by a "gating network" that selects the output of one of a set of "expert networks" assigned to sub-tasks. For any given state, the gating network selects the expert network that provides the best solution. Each expert network therefore learns to perform well in one part of the state-space only, effectively decomposing the task. Though all the expert networks receive the same input, the learning task is made simpler by only having to be accurate in a small region of the global state-space. This approach has also been extended hierarchically, with gating networks at the lowest level feeding into a higher level of networks [Jordan and Jacobs, 1993]. This approach has been applied successfully to various classification schemes, as well as robot control.

A significant difference between this approach and our modular reinforcement learning is the way the task is decomposed. Jacobs and Jordan partition the state-space into separate regions, each of which is handled by a separate expert network. Thus, at any given time only one network is relevant to the task. In the modular reinforcement learning approach however, each module sees only those inputs relevant to its sub-task, but is not relegated to any particular region of the global state-space. All modules therefore provide input to the current situation. The former approach is useful in those domains (such as classification tasks) where sub-tasks are clearly independent and need to be achieved separately, and the latter approach is better suited for dealing with tasks where many sub-tasks need to be considered at once. Lastly, Jacobs and Jordan depend on a neural network approach requiring a form of supervised learning where the error of the network's output is immediately available. As described above, Singh partially adapted this method to work in a reinforcement learning setting but was forced to apply limitations on the types of reward functions allowed. In more complex domains with reward functions allowing for partial satisfaction of goals (or progress measurement) and where there is much delayed reinforcement, the neural network approach is not applicable.

Humphrys has developed an approach similar to ours, using a fixed arbitration scheme, and learning the individual sub-tasks [Humphrys, 1996]. Modules limit their inputs and have individual reward functions that are separate from the global reward used as an evaluation metric. Several different arbitration strategies are attempted, varying which measure to use (eg. maximize happiness or minimize unhappiness) to determine a winner in a competition for control of the agent. This measure, called the W-value, is based on the Q-values learned by the individual modules. The module with the largest W-value executes an action, and all other modules update their W-values, so that they may potentially become the winner next time the state is encountered. Each module also updates its Q-value, based on its associated reward function. These reward functions are not directly related to the global reward function but seem to correspond to sub-goals, or components of sub-goals of the task. A genetic algorithm is used to determine the best magnitude for each of these rewards. The system manages to learn an approximate solution, though no mention is made of how the learning time compares with that of a monolithic agent. Humphrys regards the lessened memory requirements of the modular approach compared with a monolithic implementation to be the main advantage. However, the comparison is made with regards to the size of the space of all possible states. The space of states actually encountered during a typical experiments will be much smaller, making memory considerations less important. Furthermore, by only considering competitive arbitration strategies, the approach may be limited in what types of goals it can handle. As our experiments show, a competitive approach may completely ignore one or more sub-goals, potentially leading to poor performance.

Most of the approaches dealing with independent sub-tasks concentrate on behavior arbitration, assuming that solutions to the individual sub-tasks are already available. This assumption greatly simplifies the learning task since any errors made by the agent can then be attributed to the arbitration process which can then be adapted. Having to learn both the behaviors themselves and the arbitration the agent would need to handle the credit assignment problem to determine whether to adapt one (or more) behaviors, or the arbitration scheme. The approach proposed by Jacobs and Jordan is able to learn both since the supervised learning scheme allows them to determine which expert network performs best in any particular region of state-space and direct the arbitration based on that criteria. Our modular approach to reinforcement learning can be seen as related to Arkin's composition of potential fields, but differs from it (and the other architectures) by having a fixed arbitration scheme while learning each individual behavior. If decomposition is to be used to speedup the learning of complex tasks, one cannot assume that solutions to sub-tasks are available and must therefore be learned.

2.4 Arbitration and merging techniques

In chapter 4 we discuss two arbitration techniques: nearest neighbor and greatest mass. These are simple techniques that have been used in many other circumstances. Maes strategy of selecting the behavior that first reaches a certain threshold of activation, described above, is essentially a nearest neighbor approach, for example. An interesting property of nearest neighbor however, is its relationship to approximation algorithms of the Traveling Salesman Problem. The similarity of TSP to policy arbitration is perhaps easiest to see in a grid world example, with some set of cells corresponding to goal locations. An agent wanting to optimize its traveling distance to achieve all its goals, must essentially find the shortest tour of the equivalent traveling salesman problem (in fact, since we do not require that the agent returns to its original location, the agent needs to find the shortest *Hamiltonian path*). As we described in Chapter 1, our experiments show that the nearest neighbor strategy performs close to optimal in the grid world. This is not surprising when one realizes that nearest neighbor is a good approximation algorithm for TSP, achieving the optimal tour within a constant factor in Euclidian spaces, and shown experimentally to do extremely well on cases with large numbers of cities [Johnson, 1990]. Even outside the grid world, one might be able to generalize the problem of achieving multiple goals to a variant of TSP. If all goals are goals of achievement, the problem might be viewed as finding the shortest path in state-space to all goal states. However, the more complex the task, the less likely it is that the state-space is Euclidian, so nearest neighbor might no longer be a good approximation algorithm.

The greatest mass has also been used in solutions in other problems. Using potential fields for path planning [Khatib, 1986],[Arbib and House, 1987], uses the same techniques of gathering utilities, or potentials, from different sources (obstacles and goal locations) and then summing them.

Using the greatest mass technique is one possible way the agent can *merge* policies, rather than simply selecting one to execute. This may result in executing actions that are suboptimal to each local module, but have a high global utility. Foulser, Li, and Yang, have studied this type of policy combination, using traditional, non-linear plans as their representation [Foulser *et al.*, 1990]. However, their work has been concentrated on the problem of finding the set of merging operations, that produce the optimal combined plan. This assumes that it is already known what parts of the plans can be merged, and how that is accomplished. In our setting, we do not have access to a set of non-linear plans, but must perform the merging immediately prior to executing the action. Without using lookahead, it would be difficult to determine whether a particular merging operation should be deferred, so that a better one might be performed in the future.

3 Reinforcement Learning

This chapter reviews the learning technique known as *Reinforcement Learning*. Reinforcement learning describes a large class of techniques characterized by the fact that the main learning signal is a reinforcement or reward function. We will discuss the nature and the importance of the reward function as one of the main sources of domain knowledge. Though there are several reinforcement learning algorithms available, we have based our method on Watkins' Q-learning algorithm. We briefly review Q-learning here, and a more in-depth description is available in [Watkins, 1989].

Reinforcement learning draws much of its inspiration from conditioning mechanisms observed in animals subjected to positive and negative stimuli [Minsky, 1954]. Animals often can not be directly instructed on how to perform a task, but can be taught by administering feedback that is pleasurable or painful, depending on the animal's performance. As an intelligent control mechanism, reinforcement learning is also referred to as "learning with a critic" [Barto *et al.*, 1983]. For both biological and artificial systems the basic idea is to give the agent only some indication of how well it is performing or has performed a given task, without communicating the exact nature of the task or how it might be solved. The main advantage of this approach is that all task information has been condensed into a form easily interpreted by the agent. There is no need to design a more elaborate knowledge representation in order to describe the desired goal. As we will discuss below, a disadvantage is that it may be difficult to use the simple language of reinforcement to effectively and correctly describe more complex tasks.

An important aspect of Q-learning is its ability to cope with *delayed rewards*. Many supervised learning techniques, such as neural networks, learn to associate input-output pairs, (x, z) where x is the input and z the corresponding output. After the learning stage, the system can produce the correct output z when presented with only the input x. In this type of learning the feedback is immediate, since the system's output can be immediately compared with the correct output z. In many domains, including the driving world described in Chapter 1, such immediate feedback is not available. Instead of a simple input-output pair, the agent is presented with a sequence of steps and some corresponding outcome, eg. $(x_1, x_2, x_3, \ldots, x_n, z)$. In the driving domain, each step x_i would correspond to the i'th state visited, and the outcome z might describe an event such as crashing into an obstacle. A learning system presented with this type of input must decide at which point in the sequence the decision was made that is most responsible for the outcome (this decision is also known as the *credit assignment problem*). Though the state immediately prior to crashing into the obstacle seems a likely candidate, the cause of the crash might be traced further back in the sequence. If the agent decided to turn into an alley known to be completely blocked by obstacles, the state at which the turn was made is largely responsible for causing the crash.

By waiting until the entire sequence is complete, a learning system can restructure the inputs into pairs: $(x_1, z), (x_2, z), \ldots, (x_n, z)$, and apply normal supervised learning techniques. However, incremental approaches, where the learning can be made at each time step, before the sequence is complete and the outcome known, have been shown to be more efficient [Sutton, 1988]. Using an incremental approach, the learning system can't compare the output given the current state with the desired future outcome, but rather compares it with the output corresponding to the next state. If outputs are viewed as predictions of the ultimate outcome, then the learning process can be seen as gradually propagating improved predictions of the outcome back through the sequence of visited states, as the agent repeats its experience. One early application of an incremental approach was Samuel's checker player, which adjusted its evaluation of the current board state based on the evaluation of the board layout after the move [Samuel, 1963]. Sutton formalized the approach, terming it TD(λ), and proved its convergence to an optimal solution [Sutton, 1988]. TD(λ) has been further generalized (and the generalization shown to converge) to include cases where information used in learning is not limited to immediately adjacent time steps, but can stem from any arbitrary state [Dayan, 1992]. The approach has also been applied successfully in some domains, most notably in a backgammon player [Tesauro, 1992].

However, the TD(λ) algorithm still depends on the presence on a final outcome. Thus, it is mostly suitable for tasks with a single goal of achievement. Q-learning can be viewed as type of TD learning that has been modified to allow learning different types of goals, by predicting a discounted measure of future reward (described in more detail in section 3.3). However, Q-learning retains the incremental aspect of TD learning by making updates based on subsequent predictions rather than making comparisons with some final result.

In the below description of Q-learning we use the grid-world domain shown in Figure 1.2 as an illustrative example. In this domain, the agent can move to grid cells that are vertically or horizontally adjacent. Its task is to reach all those goal locations currently marked as "active". An active goal location becomes inactive once the agent reaches it, and may be active again with some small probability at any given time step. The agent's inputs include a list of all of the goal's current activation status, as well as the row and column of the cell it currently occupies.

3.1 The agent model

Though in general, reinforcement learning allows for complex models of the agent and its domain, we simply model the world and the agent as interacting finite state machines (see Figure 3.1). The agent executes discrete actions to change the state of the world. The agent detects the current state of the world, modifies its internal state, and selects a new action to execute. In reinforcement learning, the agent's action selection mechanism is controlled by a learning system whose inputs are derived from the external sensors and internal state variables, and a reinforcement signal. The output of the system is the action to execute. The learning system's inputs and outputs are thus abstractions of the inputs and outputs of the agent's embodiment (whether it is a physical machine or a software agent). When we refer to the agent's inputs and actions, we will be referring to these abstractions, since our interest focuses on the learning system. We can thus describe a reinforcement learning agent as a triple, (S, A, R), corresponding to the agent's inputs, actions, and reinforcement, or *reward function*.

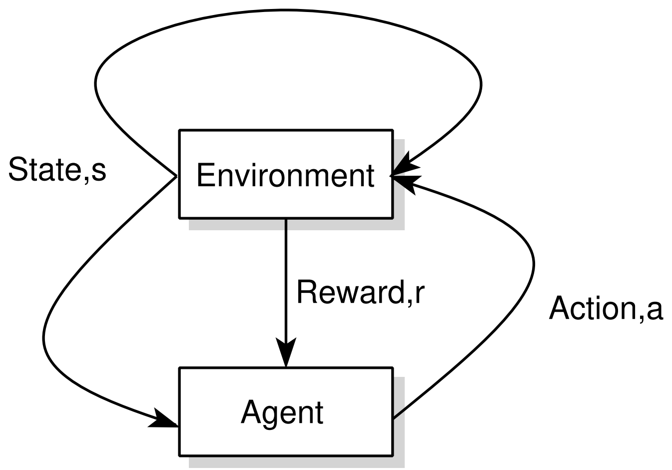


Figure 3.1: The reinforcement learning model. The agent in state s, executes action a, and receives a reward r. The action changes the state of the world, and the sequence repeats.

3.1.1 Inputs

Though it would be possible to use the raw sensor data as inputs, it is usually more practical to abstract and simplify the agent controls as much as possible. Thus the agent can be provided with sensors that report high-level information, such as the distance to the nearest wall, the color of the streetlight, or whether some location is clear or obstructed. How this information is gathered, whether its by using cameras, sonar, or laser range finders for example, is irrelevant to the learning system, and only affects the accuracy of the inputs. In our grid-world example the agent's inputs are

$$S = (\mathtt{row}, \mathtt{col}, \mathtt{a_1}, \mathtt{a_2}, \ldots, \mathtt{a_n})$$

where `row` and `col` describes the agent's location and $\mathtt{a_i}$ indicates whether the i'th goal location is currently active. The agent's location may be determined by any number of methods, such as GPS positioning, dead-reckoning, or reading barcodes placed in each grid cell. The choice of which method is used is determined by what resources are available and how much accuracy is needed.

It is possible that the same inputs result in situations that differ from each other to some degree. It is the agent designer's responsibility to ensure that, any differences in situations that result in the same inputs are insignificant relative to the agent and its task. However, it is unrealistic to assume that all sensors can perform flawlessly in all situations, so any control system depending on the sensors for input needs to be able to handle uncertainty and noise from its inputs.

The set of inputs $S = \{s_0, s_1, \ldots, s_{n_S}\}$ is thus a combination of processed perceptual data and internal state variables. We assume that each input can take on only a discrete range of values and that we can neglect the time needed to compute the input values, so that at any given time the inputs reflect the current state of the world. The values of the inputs thus determine the agent's current state, and we denote the set of all possible states, or *state-space*, by $X = 2^{n_s}$. Thus, in the grid-world, the state-space consists of $m \times m \times 2^n$ states each describing a unique agent location and set of currently active goals.

There is no assumption of independence between inputs and therefore not all combination of inputs will actually occur. For example, one input may simply report the presence or absence of an obstacle, and another gives the distance to the obstacle, or 0, if no obstacle is present. States where there is no obstacle, and the distance input reports value other than 0 will therefore never occur normally. Though it is possible to design the inputs to avoid such dependencies, other dependencies are domain dependent and cannot be avoided. For example, inputs detecting two different types of objects would be dependent in some domains where those objects never occurred concurrently, but independent in domains where they did. Such dependencies can not be known without a large amount of prior domain knowledge. It is often easier to assume that the inputs are independent, and let the agent learn any dependencies that occur in the domain.

The agent is given no semantic knowledge about the inputs, and thus cannot *a priori* treat different inputs as being more salient to the task than others. As the agent explores the world, it attempts to learn the transition function $T : X \times A \to X$, that governs how it can traverse the state-space in order to accomplish its task.

3.2 Actions

The agent's set of actions, $A = \{a_0, a_1, \ldots, a_{n_A}\}$ are assumed to be discrete and to need equal time to complete. Any cost of executing actions can be encoded in the reward function, as described in Section 3.3. If the agent's physical embodiment is a machine or a robot, actuators are likely to be controlled by, for example, regulating the current going to a set of motors. Though it would be possible to learn to control the robot using this low-level interface, one can often provide a simpler, more abstract interface, without losing too much generality. Thus, the learning system may be learning when it is appropriate to execute high level actions, such as "move forward 50 centimeters", "turn 10 degrees to the left", or "pick up the coke can". In the grid-world domain, the actions are even more abstract: $A = \{\texttt{up}, \texttt{down}, \texttt{right}, \texttt{left}\}$.

In a robot, it is likely that all actions will be limited by some level of uncertainty. World states that appear the same to the agent may differ enough that the same action leads to different results. Other environmental conditions such as the ground texture, or dust accumulating on gears and axles, may also lead to inconsistent results. As with the input sensors, the agent designer must attempt to limit the uncertainty as much as possible, but the control system still needs to be able to cope with actions with non-deterministic results. The agent is given no prior knowledge about the expected results of actions, but attempts to learn the range of possible results through experimentation in the domain.

3.3 The reward function

Often researchers describing their model of agent/world interaction, depict the reinforcement signal as coming from the environment or world to the agent [Peng and Williams, 1992][Watkins, 1989]. In a biological system this may be true. The brain translates different stimuli into pleasant or unpleasant sensations depending on the current task and context. In artificial reinforcement learning, this is modeled by a reward function that translates the agent's perceptions into a real number. Many researchers recognize that for any real artificial agent implementation, there is no convenient reward function in the world inserting appropriate values into the agent. The reward function is thus depicted as a part of the agent that translates the observed state into a reward value [Whitehead and Ballard, 1989][Kaelbling, 1989]. That is, part of the agent's *a priori* knowledge is the ability to recognize states and assign a value to them.

A reinforcement learning agent must be provided with its reward function from an external source, typically a human designer. In order to define the reward function, the designer distills a large amount of task specific knowledge into a number. For example, the reward function designer will encode a notion of what classes of states represents the agent's goal. The designers of the walking robot Genghis constructed a reward function that encoded the knowledge that keeping a touch sensor on the robot's "belly" from being activated, and keeping a motion sensor on the robot's tail activated, was the most desirable state [Maes and Brooks, 1990b]. Similarly, agents in simulated worlds have been given reward functions encoding the information that desirable states are those where the sensors detect simple features, such as: the presence of cheese [McCallum, 1992], a particular color block in the agent's gripper [Whitehead, 1991], or some other distinctive marker [Lin, 1993b][Tenenberg *et al.*, 1992]. In all these cases, the human designer of the reward function had a clear idea of which classes of states (defined by the values of a small set of the sensors available to the agent) corresponded to the agent correctly accomplishing some task.

In general, the reward function, R, can be as complex as desired. It can be defined to depend not only on the current state, and the action that led to it, but also any other circumstances in the agent's past history. However, the learning method we use as the basis of our system, Q-learning, requires that the reward depend only on the current state. The cost of the action executed to reach that state may also be included in the reward. In our experiments, we assume that any such costs are negligible. Given a state therefore, the reward function returns a scalar real value ($R : X \rightarrow \Re$).

Since the reward function is the agent's only indication of how well it is meeting its goals, it must be designed with great care. For all practical purposes, the reward function *defines* the agent's task. For complex problems, it may be necessary to refine the reward function as an agent discovers ways to get the reward without accomplishing the actual task. For example, a garbage collection robot gets a small reward for picking up a piece of trash, and a larger reward for depositing it into a garbage can. The robot may easily discover that it can quickly accumulate the small reward several times in succession, by repeatedly dropping and picking up the garbage (perhaps even repeatedly retrieving and depositing it into the garbage can!). The reward function then needs to be modified, for example by giving a large negative reward, or punishment, for dropping garbage anywhere but the garbage can (and for picking up objects from the garbage can).

In order to design a correct reward function, there must first be a clear idea of what the agent's goals should be. In the field of decision theory and planning, there is a growing amount of research on how to properly assign utility to goals, whether they be goals of achievement, avoidance, or maintenance [Haddawy and Hanks, 1990]. Given a simple goal, it is fairly straightforward to define a reward function leading to the proper behavior. For an achievement goal for example, the reward function returns a positive value in the goal state, and 0 in all other states. Similarly, an avoidance goal can be encoded by a negative value in the state that is to be avoided, and 0 reward elsewhere. The task in grid-world is to reach a state where all the goals are inactive (ie. all goal locations have been reached at least once), so the reward function could be designed to give out a reward of 1 if all goals are inactive, and 0 otherwise.

As long as there is one goal state, or a class of states where the goal has been achieved equally well, the reward function remains simple. However, reinforcement learning gains much of its power from the ability to use the reward function to specify different degrees of goal satisfaction, and identify sub-goals of a complex task. Thus, if our grid-world task was setup as a set of repeated trials with a fixed time-limit, the reward function could assign a reward at the end of each trial that was proportional to the number of goals that had been reached. This would speedup learning, because partial solutions would be seen as useful, and used as starting points for more successful solutions.

In a similar fashion the reward function may be defined to give out rewards when sub-goals of the overall task have been accomplished. Rewarding sub-goals is one way to encode domain and task knowledge. In the garbage collecting robot described above, it seems clear that the agent must first pick up a piece of trash, before it can deposit it into the garbage can. Giving a small positive reward when the sub-goal of picking up the trash is achieved is akin to giving the agent a "hint" that it is moving in the right direction. The agent will thus learn more quickly to pick up the trash than if it only got rewarded after it has successfully picked up and deposited the trash in the garbage can. Similarly, in the grid-world, rewarding the agent at the moment it reached an active goal-location would be a good way to specify that reaching each individual goal is a sub-goal towards the task of reaching all goal-locations.

Apart from the risk of making the type of errors in the reward function definition described previously, rewarding sub-goals may also bias the agent against finding efficient but unforeseen solutions to the problems. In our example, if the garbage robot could use a leg to kick the trash into the garbage can, attempts to learn this solution would be abandoned once it was learned that reward can be acquired by picking up the trash with an arm. Thus, though kicking the trash might be more efficient, the reward function is biased against any solution that does not involve picking up the trash first.

Increasing the complexity of the reward function can clearly speed up learning

[Mataric, 1994]. The rewards can be used to guide the agent in what is deemed to be a desirable direction towards the goal, thereby limiting the amount of random exploration that must be undertaken before a solution is found. Also, rewards can be used to steer the agent away from situations that are known to be undesirable (eg. the edge of a cliff, or walls).

However, great care must be taken to ensure that the reward function leads to the desired behavior. The relative magnitudes of the reward need to reflect the relative importance of the states being rewarded. One could prioritize sub-goals by assigning rewards of greater magnitude to the more important ones. It is not sufficient however, to use the reward to create an ordering among sub-goals. If the reward for one goal is twice as large as for another, then an agent that achieves the second goal twice, will receive as much reward as one achieving the first one once. In this case, the performance of the two agents must be considered to be of equal quality, since the same amount of reward is accumulated. Therefore, the relative magnitude of rewards is significant, in addition to the ranking they create.

The implicit "meta-goal" of a reinforcement learning agent is to maximize some measure of the reward. In order for this goal to be meaningful, there must be some finite time limit after which this measure is assessed [Seidenfeld, 1985]. A correct reward function therefore, must rank all possible trajectories through the state-space of a length corresponding to the finite time limit in order of desirability. That is, if the agent is to learn a task within N time-steps, then the total reward for a sequence of N steps should exceed the total reward for any other sequence, only if the second sequence is less desirable (in terms of achieving a task) than the first. For example, in the grid-world domain, at each time step, an inactive goal becomes active again with some small probability p. If the task is modified so that while the most desirable outcome is that all goals are inactive when the time limit is reached, there is one location, in a corner for example, that is deemed less important than the others, and can thus be ignored, if necessary. If a reward of 1 is given for the other goals, the lower priority of the corner goal could be indicated by giving a smaller reward when that goal is reached. The difficulty lies in determining the magnitude of that reward, given that it is unknown with what frequency the corner goal will be reactivated. If the reward is set to 0.1, for example, reaching that location 11 times, as well as reaching 4 other goal locations, is defined to be preferable to reaching 5 goal locations, not including the corner goal. That reward setup could therefore lead to undesired behavior if the corner goal was both easily reached, and reactivated with a very high frequency. Thus, when there are several sub-goals that must be prioritized, and whose frequency of occurrence is unknown, designing the reward function can become very difficult. Often it is not practical to attempt to rank all possible sequences of state and design a reward function that produces that ranking. In domains with complex reward functions, it may be necessary to perform experiments testing the correctness of the reward. Only by placing the agent in the world and observing undesirable behavior due to improper reward magnitudes, can one gradually adjust the reward function so that a good solution to the task can be found.

3.4 Q-learning

The reward function may provide some non-zero reward for only a subset of states (the states where the task has been achieved, for example), so the agent must learn how to accomplish its task though the reward function may only provide a useful learning signal in a very small part of the domain. To accomplish this, the agent estimates the expected utility, called the Q-value, of executing an action from a given state. The Q-value, $Q_f(x, a)$ is defined as the expected discounted reward of executing action a, from the state x, and then following the policy f, thereafter. A policy is simply a function that determines which action to execute in any given state.

Q-learning is designed to maximize expected future discounted reward. This quantity is also called the *return*, and is defined as

$$r(t) = \sum_{i=0}^{\infty} \gamma^i R_{t+i}$$

where R_{t+i} refers to the immediate reward received at time $t+i$, and γ is a temporal discount factor, with magnitude between 0 and 1. Thus, Q-learning attempts to learn a policy that will gather as much reward as possible, but prioritizing, according to the value of γ, rewards reachable in the short-term over more uncertain rewards that may be achieved further in the future. In highly dynamic domains the temporal discount factor may be low, reflecting the uncertainty of whether rewards can be attained in the future.

The policy is defined to execute the action with the highest Q-value in the current state,

$$f(x) = \arg\max_{a \in A} Q(x, a).$$

If action a is executed in state x, moving the agent to state y, the Q value is updated according to the following formula,

$$Q_t(x, a) = (1 - \alpha)Q_{t-1}(x, a) + \alpha \left[R(y) + \gamma U(y)\right]$$

where α is the learning rate, γ the temporal discount factor, $R(y)$ the reward associated with state y, and

$$U(y) = \max_{a \in A} Q(y, a)$$

.

It is assumed that the domain is such that at any given time, the agent has enough information to be able to decide what the optimal action is. This is known as the *Markov property*, and entails that any time an agent is in a given state, the optimal action remains the same. The Markov property does not preclude stochastic domains, since the optimal policy maximizes *expected* future reward. Given a stochastic domain with a distribution of possible outcomes for each action in a state, the Q-values will reflect the expected utility of each of the actions. If the Markov property does not hold, the result may appear similar to the stochastic case: at different times the same action will lead to different results. The difference is that there may not be a well defined distribution of possible outcomes, and thus the Q-value computed will not reflect a real expectation.

By also assuming that the agent employs an exploration strategy that occasionally deviates from the policy determined by the Q-values (thereby allowing the discovery of new solutions), and an infinite number of time steps, Q-learning can be shown to converge to an optimal policy [Watkins and Dayan, 1992]. The two last assumptions are necessary to ensure that once a solution is found, the agent continues to attempt to improve it. Without exploration, the agent would stay with the first solution found, which typically has a lot of room for improvement. The agent must search for an infinite number of time steps in order to provably find the optimal solution. Clearly this is not possible in practical applications. Typically however, a finite number of steps suffice, and the learning rate and exploration rate are both decayed, so that after a certain time point the agent is not searching for new solutions [Thrun, 1992].

It is possible therefore, that an optimal solution is not found by an agent that limits its exploration and learning. However, since optimality is defined by "maximizing expected future discounted reward", it is often the case that sub-optimal solutions can still be used to achieve the task. Though there might be a loss of efficiency, the optimal solution may differ from a sub-optimal one in insignificant details only. As described in Section 3.3, defining reward magnitudes can be difficult, and different amounts of total rewards may not be a good measure of the difference in quality of solutions. Furthermore, in some cases, an optimal solution might not be achievable given the resources available. This is often true in the complex domains where several different tasks need to be achieved. In these situations, sub-optimal solutions that nonetheless achieve the task must be acceptable.

Q-learning is an attractive learning algorithm, because it requires only very little prior knowledge of the domain. However, as the agent's task and domain increases in complexity, the learning time may increase dramatically. Since the size of the state-space may be as large as 2^{n_s}, adding more features to the domain, necessitating an increase in the agent's inputs, can cause an exponential increase in the size of the state-space. Since the agent may need to explore the entire state-space to find the optimal solution to its task, the increase in the state-space size may also lead to an exponential increase in the learning time.

4 Modular Q-learning

The main weakness of Q-learning is the amount of time necessary to learn the task. In the grid-world example shown in Figure 1.2, the task is simply to find a path from the starting position that visits each goal location at least once. The agent begins each trial at the starting position, and a trial ends when all goals have been reached. Given no prior domain knowledge, and no reward until the goal has been reached, the agent must essentially perform a random search, until the goal has been found in the first trial. It can be shown that in domains with certain properties (such as having reversible actions), the time to find the goal state in the first trial is exponential in the number of states in the domain [Whitehead, 1991].

To a large extent, the long search time cannot be avoided. Given that the agent has no domain knowledge, it must explore every state in order to guarantee that the optimal solution is found. If there is some prior knowledge available about what sub-goals need to be achieved, or other forms of partial solutions, the reward function or initial Q-values can provide that information to the agent. However, in the complex tasks we are studying, the available domain knowledge does not pertain to the solution to the task that the agent must learn, but rather to the *structure* of the task. With tasks that are composed of largely independent sub-tasks, such as keeping the office printer stocked with paper, recycling soda cans, and delivering mail to individual office, the agent designer may not know how each individual sub-task can be accomplished by the agent, but does have some knowledge about the independence of the task. Yet, using standard Q-learning the independence of tasks cannot be exploited. If the agent learns to find and dispose of a soda can while the printer is fully stocked, that knowledge will be unavailable when the printer is only half full. Because the input indicating the status of the printer has changed, the agent is now operating in a part of the state-space it has previously not seen. It will have to relearn how to dispose of soda cans, since it does not know that the status of the printer is largely irrelevant to the process of picking up garbage. Similar changes in inputs relevant only to other tasks, will also necessitate relearning the same procedure. The more concurrent sub-tasks the agent must handle, the larger the state space grows, as more inputs are needed to handle specific tasks. Standard reinforcement learning provides no mechanism allowing the agent to utilize a previously learned method to accomplish a sub-task when the only inputs that differ are only relevant to other sub-tasks.

This chapter describes the modular architecture. This architecture allows an agent designer to apply domain knowledge about sub-tasks, by creating independent modules whose inputs are relevant to a specific sub-task only. Since no module depends on any sub-task other than that assigned to it, the relearning described above can be avoided.

In the below sections, the following topics are addressed:

- The definition of the modules. We describe the input and output of the modules in this architecture.
- The definition of the modular algorithm. We show how the modules are used in the system as a whole.
- The definition of *approximation functions*. Since each module learns how to accomplish its sub-task only, a mechanism is needed to merge the knowledge gathered by the modules so that the best action overall can be executed. This is the role of the approximation function, which, since it arbitrates between the different modules is also referred to as an arbitration strategy. We describe two different arbitration strategies called *greatest mass* and *nearest neighbor*.
- Extensions to the approach. We describe two extensions to the modular architecture, designed to improve its performance. In the first, modules are used to initialize a monolithic system, and in the second an activation function is used to limit interactions between modules.

4.1 Module definition

We have developed a modular Q-learning architecture that allows the agent to take advantage of the independence of sub-tasks. Figure 4.1 illustrates an agent using the modular architecture. As with standard Q-learning, the agent executes actions to alter the state of the world, and detects the new state through its sensors and other inputs. However, rather than considering all inputs concurrently, thereby learning all sub-tasks in the same monolithic state-space, the modular approach separates inputs according to what sub-task they are relevant to. A separate module is pre-defined for each sub-task, which receives only that input information relevant to its task. Thus, the sub-tasks are learned independently, but in parallel with each other.

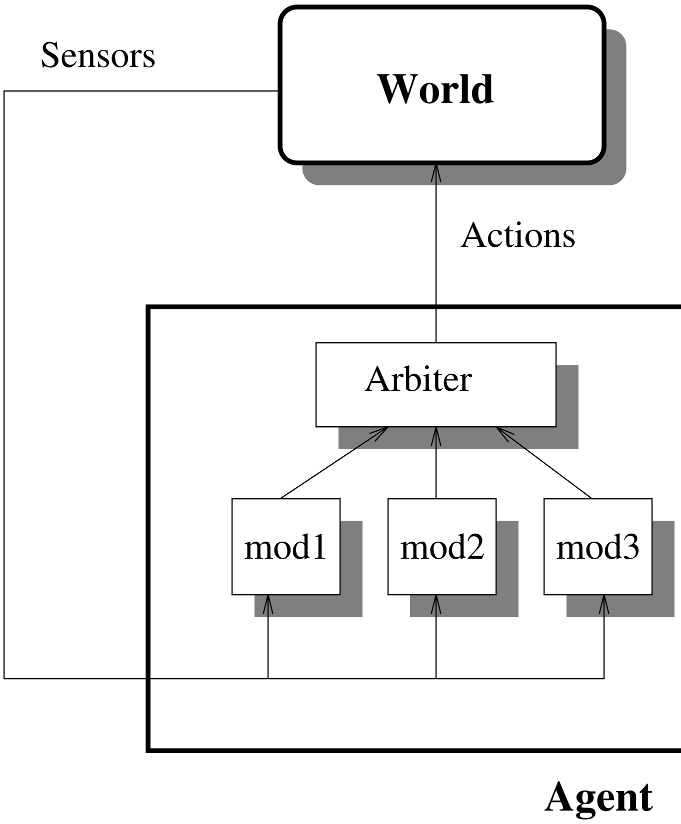


Figure 4.1: An agent using the modular architecture receives inputs from the world and executes actions to change its world state. However, the inputs are separated into modules which then learn in much smaller state-spaces.

More formally, we define for each module i, three functions that specify its associated sub-task and relevant information:

- m_i is a selection function, that selects only those parts of the state-description relevant to the given sub-task. Each m_i thus defines a new state-space which is an abstraction of the original state-space.
- R_i is a reward function that assigns reward based only on how well the sub-task is being accomplished, independent of the state of the task as a whole.
- Q_i is a Q-function that estimates expected reward from the sub-task reward function.

Figure 4.2 illustrates how these functions are used in the modular architecture. Given the input from sensors and internal state variables, each module uses its selection function and reward function to maintain its own Q function. Therefore, while m_i and R_i are defined by the agent designer, the modules learn their own Q-functions. These Q-functions are then used by an arbiter to determine which action the agent should execute. We will refer to the state-space, reward function, and Q function, related to the modules as *local*, as compared to *global* equivalents of a non-modular system.

4.1.1 The selection function

The agent's inputs $S = \{s_0, s_1, \ldots, s_{n_S}\}$, determine what it can distinguish in its domain. In the above office robot example, the agent would have inputs determining the paper level in the printer, detecting soda cans, measuring the distance to obstacles, and many others. However, the agent's task consists of many sub-tasks, and not all inputs are relevant to all sub-tasks. For example, it is not necessary to know about the presence of soda cans in order to keep the printer stocked with paper. In order to use the modular approach to reinforcement learning, the agent designer must be able to identify the sub-tasks, and what inputs are relevant to them. In the grid-world domain, the inputs are

$$S = (\mathtt{row}, \mathtt{col}, \mathtt{a}_1, \mathtt{a}_2, \ldots, \mathtt{a}_\mathrm{n})$$

and the sub-tasks correspond to reaching one of the goal location. Therefore, for each sub-task, the relevant inputs consist of the `row` and `col` inputs, and one of the activations a_i, only.

This filtering of the inputs is accomplished with a selection function m_i. The selection function is defined to return only those inputs relevant to the given

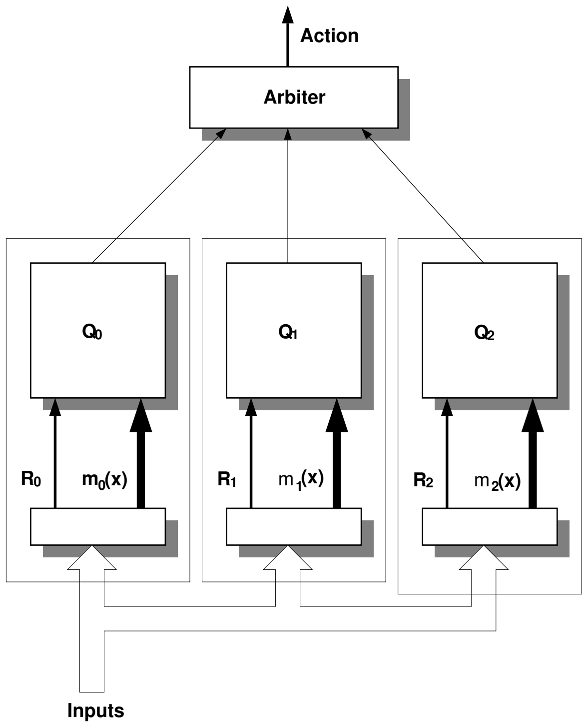


Figure 4.2: In the modular architecture, each module uses only that part of the input x, returned by the selection function m_i, and its reward function r_i, to estimate Q-values relative to its sub-task. These Q-values are then used by the arbiter to select an action to execute.

sub-task. Thus, just as the set of inputs S define a state-space X, each module considers only a subset of inputs S_i, defining a smaller state-space X_i. The selection function can then be viewed as a projection of the global state-space X onto the local state-space X_i. Each state in X_i corresponds to a set of states in X. For any state $x_i \subset X_i$ this set is easily defined as

$$\{x | x \subset X, m_i(x) = x_i\}$$

The state-space X_i can be thought of as an *abstraction* of the global state-space, since it disregards part of the available information. In machine learning, having a set of states be represented by a single state is typically referred to as *hidden state* or *perceptual aliasing*. Hidden state is usually the result of the agent having insufficient information about its current state, leading to different results when the same action is executed in apparently the same state. Since this inconsistency can hamper learning, it is usually combated by attempting to gather more information about the state [Whitehead, 1991], or using some form of short term memory [McCallum, 1996]. However, sometimes hidden state can be useful to an agent, eliminating irrelevant detail from the state description (This case is described as *passive abstraction* by Agre [Agre, 1988]).In our modular approach for example, we deliberately introduce hidden state in order to reduce the size of the state-space each module must learn in. This reduction of the state-space size is a way to achieve generalization, since one state in the local state-space X_i covers a set of states in the global state-space X. Since the information that is omitted from the module is information that has been deemed irrelevant to the associated sub-task, it is unnecessary to distinguish between the global states corresponding to any given local state. For example, if a module was defined to handle keeping the printer stocked with paper, one of the local states might be that the paper tray was completely empty. This state could correspond to several global states, distinguished perhaps by whether or not a coke can was on the floor of the printer room. However, for the sub-task of keeping the printer stocked, it would not be necessary to distinguish between those states.

Of course, it is often difficult to determine whether or not some information is relevant to a sub-task. One might imagine that the presence of a coke can in the room might be relevant if the printer is almost out of paper. The agent could then judge whether the optimal path would be to first dispose of the coke can and then refill the printer, or go to the printer first since it is likely to run out of paper before the coke can is taken care of. These types of interactions between tasks are handled by the *approximation function* and thus do not necessitate the inclusion of more inputs in one or more modules. We cannot define an exact algorithm to determine what the selection function m_i should be for each module. A reasonable heuristic seems to be that to only include that information which would seem necessary if the given sub-task were the only task the agent needed to handle.

4.1.2 The reward function

The reward function is how the agent designer defines the agent's task. In a simple task, the reward function will simply assign no reward to most states and some positive reward to a goal state. In a complex task composed of multiple, concurrent sub-tasks, the reward function might consist of some combination of several sub-reward functions. The office robot described above, for example, could have separate reward functions evaluating the different sub-tasks. The global performance is evaluated by some linear combination of the individual reward functions. It is these types of tasks that the modular architecture is best suited to. In these domains, there are a number of sub-tasks that must be attended to, each of which is clearly delineated from the others. A reward function can be designed to evaluate each sub-task, independently of the status of the other sub-tasks. The total reward of the agent is defined as

$$R(x) = \sum_{i=1}^{n_m} R_i(m_i(x))$$

where x is the state being evaluated, n_m the number of modules, and m_i the selection function defined previously. In the grid-world example, the individual reward functions could be defined as:

$$R_i(\{\texttt{row}, \texttt{col}, \texttt{a}_\texttt{i}\}) = \begin{cases} 1 & \text{if } \texttt{row} \text{ and } \texttt{col} \text{ correspond to the location of goal } i, \\ & \text{and } \texttt{a}_\texttt{i} = 1 \text{ (ie. goal } i \text{ is active)} \\ 0 & \text{otherwise} \end{cases}$$

The agent's total reward then consists of the sum of the individual rewards.

It is possible to assign weights to each of the individual reward functions, indicating priorities, or even use a more complex algorithm than a simple sum to evaluate the agent's global performance. Unless otherwise indicated however, we assume that a simple sum is sufficient, and that any relative priorities between sub-tasks are reflected in the magnitude of the rewards given by the reward functions.

4.1.3 The approximation function

Each module has its own state-space and its own reward function, and is thus equipped to learn its sub-task using any reinforcement learning mechanism. Using Q-learning, a module applies the standard update formula to define a local Q-function. This Q-function is defined for the module's state-space and reflects estimates of the module's reward only. We can thus write the update formula for the local Q-function, Q_i, as

$$Q_i(m_i(x), a) = (1 - \alpha)Q_i(m_i(x), a) + \alpha(R_i(m_i(y)) + \gamma U_i(m_i(y)))$$

where

$$U_i(m_i(y)) = \max_{a \subset A} Q_i(m_i(y), a)$$

As the agent performs actions in the world and receives reward, all modules update their Q-functions independently and in parallel. Since the agent has no global state information, any estimation of the global Q-function must be done using the local information gathered by the modules.

Though there are many different ways one might combine the local Q-functions to estimate the global Q-function, we have focussed mainly on an algorithm termed "greatest mass", while also briefly examining a method termed "nearest neighbor".

The greatest mass approximation algorithm simply sums the local Q-values to estimate the global Q-value. The estimate, Q_{gm}, is thus given by

$$Q_{gm}(x, a) = \sum_{i=1}^{n_m} Q_i(m_i(x), a)$$

As discussed in Chapter 2, the greatest mass algorithm is similar to potential field methods used in path planning, for example. In path planning one can assign attractive potential fields to goal locations and repulsive fields to obstacles. Combining the fields leads to a "path of least resistance" avoiding the obstacles, leading to the goal. Analogously, the local Q-functions define positive and negative utilities, which when combined point to a path towards the goal in state-space. Adding the Q-values to estimate the global Q-value can lead to actions being selected that, while sub-optimal relative to the sub-tasks, represent a compromise that takes into account information from all the sub-tasks. For example, one module might assign a large positive utility towards going forward (to reach a soda can perhaps), while an obstacle avoidance module assigns a large negative utility to the same action (if there is an obstacle in the way). In the greatest mass approximation, the positive utility from the first module would be offset by the negative utility by the obstacle avoidance module. However, an action that moves around the obstacle would be given a positive utility by both modules, since it moves in the general direction of the goal, and avoids the obstacle. The sum of these Q-values would then exceed that of the action moving directly forward. In this manner, actions that are sub-optimal to any given module, but the best when all sub-tasks are considered together, can be selected by the greatest mass approximation method. Finally, since the global reward is derived from the sum of the individual local rewards, and the local Q-functions are dependent on the local rewards, it is not unreasonable to assume that the sum of the local Q-functions will be a good approximation of the global Q-function in many cases.

Another approximation strategy we have investigated, though not to the same extent as greatest mass, is the nearest neighbor strategy. In this approach the global Q-value estimate is arrived by choosing the maximum of the local Q-values. That is,

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

This approximation algorithm is called nearest neighbor since it essentially leads the agent to attempt to reach the nearest reward first. This can lead to the agent achieving one sub-task while ignoring the others. In some domains, such as grid world, where all the goals are goals of achievement, for example, this is a reasonable strategy, and analogous to the nearest neighbor approximation strategy for solving the traveling salesman problem. However, in domains where there is a mixture of goal types, including avoidance goals, ignoring any sub-task is unwise.

For both of these approximation methods, and any other method of estimating the global Q-function using the local information, it is not necessary that the estimate is numerically equivalent to the true global Q-function. In order for the estimate to be accurate, it need only provide the same ranking of actions in any given state. In fact, as long as the both the estimate and the true Q-function simply agree on what is the best action in all states, the same behavior will result.

4.2 The modular Q-learning algorithm

The modular Q-learning algorithm is outlined in Fig 4.3. It proceeds along the same lines as the regular Q-learning algorithm, differing only in how the policy is updated and the action to execute is selected. We have left the issue of *exploration* largely unexamined. Exploration is done by executing a random action with some probability, instead of the action recommended by the policy. There are much better exploration strategies available for Q-learning (see [Thrun, 1992] for an overview) that could possibly be adapted to the modular approach. For example, counter-based approaches, that try to guide the exploration towards actions which have been executed less frequently could also be used in the modular system. However, since different modules act in different state-spaces, it is possible for the agent to be in a well explored part of the state-spaces of some modules, and in relatively unknown parts of other modules state-spaces. Some mechanism would have to be designed to properly arbitrate between the differences in the value of exploration in a given direction for the different modules.

4.3 Extensions

As will be seen in the experimental results and analysis, the modular approach is necessarily limited in the level of the performance it can reach. By decomposing the task into modules, separating reward and input information, we may have

1. For the current state x, select the action to execute. Either:
   (a) with some probability (eg. 10%) perform exploration by randomly selecting an action, or
   (b) find the action a such that $a = \arg\max_{a \in A} Q_{gm}(x, a)$. If there is a set of actions corresponding to the maximum value, let a be randomly selected from this set.
2. Execute action a, leading to the new state y
3. Update each module according to the update formula: $Q_i(m_i(x), a) = (1 - \alpha)Q_i(m_i(x), a) + \alpha(R_i(m_i(y)) + \gamma U_i(m_i(y)))$
4. repeat, with y set to the current state.

Figure 4.3: The modular Q-learning algorithm.

rendered the system incapable of learning the optimal policy in situations where all information needs to be considered in conjunction. We have therefore developed some extensions to the modular approach that allow the system to relax the decomposition and recombine the information. The first method completely abandons the modular approach after it has reach a certain level of performance. The learned policy is used to initialize the monolithic system, which then continues to learn. The second extension we discuss, in effect removes information from the system, rather than trying to add it. A standard difficulty in systems that use decomposition is that of interfering sub-goals. It may be that in some situations a given module may be influencing the approximation method (greatest mass or nearest neighbor) adversely, even though it may not be relevant. To avoid such interference, one can provide an *activation function* that ensures that only relevant modules influence the current action selection.

4.3.1 Initializing a monolithic system with modules

In this method, the modular system is used to avoid the initial poor performance of the the standard monolithic system. The algorithm proceeds as follows:

1. Use the modular approach to try to learn the task
2. At some point in time, switch to the standard monolithic approach, initializing the system with the policy learned using the modules
3. Use the monolithic approach to continue learning the task

If the modular system can learn a policy that is optimal in most cases, and only makes mistakes in a small set of states, this approach can work well. By using the modular approach initially, most of the policy is learned correctly in a short period of time. Once the switch to the monolithic system is made, it is only necessary to correct the policy in a few states.

One difficulty lies in determining when to switch to the monolithic system. In our experiments, we simply switch at a given time-step. However, it might be more efficient to attempt to monitor the agent's performance and make the switch at the point in time when learning using the modular method does not lead to any improvements.

When the switch is made, the monolithic Q-function is initialized using the Q-estimate produced by the modular system. If the greatest mass algorithm is used for example, the monolithic Q-function, Q, would receive an initial value, for each state/action pair, according to the formula

$$Q(x,a) = Q_{gm}(x,a)$$

or

$$Q(x,a) = \sum_{i=1}^{n_m} Q_i(m_i(x),a).$$

When the monolithic policy has been initialized in this way, it will lead to the same behavior as the modular policy. As the agent arrives in situations where the modular policy leads to a sub-optimal action, the monolithic system will be able to make the necessary correction.

This extension assumes that all that is necessary to correct the modular policy, is to alter the recommended action in a small set of states. It is possible however, that the modular policy has learned a sub-optimal solution that differs greatly from the optimal one, and small corrections will not be sufficient. In these situations, the entire policy will have to be re-learned, and initializing with the modular policy will harm more than it helps.

4.3.2 Activation

A complex agent, biological or artificial, has a large number of goals it attempts to satisfy over time. At any given time however, only a few of those goals influence the agent's behavior. As a human drives to the store, his actions aren't influenced by the way he will park the car once he reaches his destination. Similarly, the effect of food gathering goals on the action selection process may vary depending on how hungry the agent is. In an artificial agent using the modular approach, each module corresponds to one sub-task. We can extend the modular system by adding an *activation level* for each task. A module is active if it is relevant to the current situation, and inactive if not. It is possible to have the activation be real-valued, indicating varying levels of influence, as in the example with hunger. In the following we only consider binary values for the activation levels however.

The activation levels are added to the modular system by introducing an activation function, p_i, for each module. Each activation function returns the module's activation level in the input state. Each module continues to learn and update its policy regardless of its activation level; it is only in the action selection step of the algorithm that activation has an influence. Thus, when the greatest mass approximation of the global Q-values are computed, each module's Q-value is weighted by its activation level:

$$Q_{gm}(x, a) = \sum_{i=1}^{n_m} p_i(m_i(x))Q_i(m_i(x), a).$$

With binary activation levels, irrelevant module's will thus be disregarded in the action-selection process.

The activation levels is another mechanism by which domain knowledge can be introduced. In the grid-world example (and when using activation to indicate hunger as mentioned above), the activation level is part of the task definition. That is, the agent receives no reward for achieving an inactive task, such as reaching a previously visited goal. In this extension however, activation levels do not influence the reward function. Instead they are intended to provide a heuristic mechanism allowing the agent to ignore certain goals. In an office robot for example, the goal of checking the paper supply of printers could be set to be active only when the robot was in a room with a printer present. This might prevent the robot from learning an optimal route to the printer rooms incorporated with accomplishing its other tasks. However, using the activation function would decrease the time to learn the task as a whole, since the time necessary to learn the optimal path can be eliminated. The danger of course is that the robot will *never* enter the printer room if it is ignoring its goal of checking the printers. However, if it is known that the printers are in a central location that will be visited during the performance of the robot's other tasks (such as in the mail-room), the activation function can be used with some assurance that the printer goal will be attended to.

It should be stressed that these activations are a heuristic mechanism that could prevent the agent from learning an optimal policy. Strictly speaking, a module is only "irrelevant" in situations where all the actions have the same utility, and will therefore not influence the action selection mechanism. Activations can be used by the agent designer to trade-off learning time with solution quality, by specifying situations where certain goals should be ignored so that the agent can learn solutions to other, more immediate goals.

In our experiments with this extension, the activation functions were defined by the agent designer, using available domain knowledge. The activation could also be set by some higher level planning module, or perhaps even be learned. In the above formulation, the activation functions p_i are given the same perceptual inputs as the corresponding module. This assumes that the information given to each module is sufficient to determine whether it should be active or not. However, if it is necessary to use the activation to control complex interactions between modules, the activation functions may need access to all inputs.

4.4 Summary

In the modular approach, sub-tasks are identified by the agent designer and are learned by modules using complete Q-learning systems. Since the modules learn independently of each other, in separate state-spaces, the enormous state-space size necessitated by many sub-tasks in the monolithic approach is avoided. Learning can therefore proceed much more quickly, though the solutions may not be optimal. By decomposing the inputs into separate modules, the global overview of the task is lost. The system thus depends on the approximation function to produce Q-values from the modules' local Q-values, that lead to a good approximate policy. We described two approximation strategies: greatest mass and nearest neighbor. The first has the capability to consider several sub-tasks concurrently, and the second implements a simple greedy approach. Which strategy is appropriate depends on the type of task given the agent.

Since the modular approach produces an approximate solution, we consider two extensions to attempt to improve their quality. In the first, the modular policy is used to initialize a monolithic system. The monolithic agent could then possibly avoid the initial random search in order to find a solution to the task. This extension could then combine the fast learning time of the modular architecture with the promise of an optimal solution given by the monolithic approach. The second extension uses an activation function to limit the number of modules that need to be considered at any given time. The agent can therefore be capable of solving a large number of sub-tasks, but at any given time, only a relevant subset needs to be considered. This will reduce interference between modules which might otherwise lead to poor performance.

The modular architecture and its extensions thus provide different techniques by which domain knowledge about the task can be applied to improve the agent's performance.

5 The Simple Driving Domain

The modular architecture is designed for domains where the task to be learned is composed of a set of relatively independent sub-tasks. Furthermore, since the modular approach trades-off the increased learning speed with a lesser quality of the problem solution, the most suitable tasks will be those that have either alternate possible solutions, or allow for partial satisfiability. The simple driving domain (SID) was designed with these constraints in mind, to provide a good test-bed for experiments with the modular architecture.

SID is a discrete world that is simple enough to enable analysis of different possible situations and agent behaviors, yet still allows for complex tasks and goal interactions. There is no overriding global goal for the agent in SID, but simply a set of different tasks that must be handled properly in order for it to traverse the world correctly. It is thus easy to add new tasks to be solved, or to experiment with only a subset of the possible tasks.

5.1 SID world organization

The SID domain consists of a network of unevenly distributed vertical and horizontal roads, separated by sidewalks. The world is divided up into discrete cells, and any object in the world, including the agent, can occupy only one cell at a time. Figure 5.1 shows an example situation, where the agent is approaching an intersection. The scene shows a grid of 4x4 cells (the agent car and the moose each occupy one full grid cell). A street sign indicates that the agent should make a right turn, and the moose in front of the car represents a large obstacle that must be avoided. At the beginning of each experiment, a new 80x80 grid of roads is created. All the roads span the entire length or breadth of the world, so that there are no dead-ends or U-turns. The world is toroidal, so that an agent driving off one edge of the world, immediately appears at the opposite side.

Time is discrete, and agents decide how to proceed in the time between two consecutive time steps. We assume that the time needed for the agent to select an

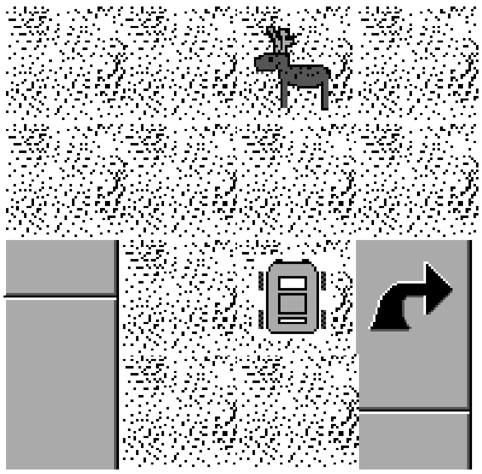


Figure 5.1: An example situation in SID. The agent is approaching the intersection and must follow the sign while avoiding the obstacle.

action is several orders of magnitude smaller than the time to actually execute it. Since reinforcement learning, and our modular approach, are reactive techniques that do not include a large amount of deliberation, it is reasonable to ignore the time needed for computation.

Figure 5.2 shows all the objects that can appear in the world. Objects can in general only appear in certain locations relative to other objects, as described below:

1. **street** Street objects form streets that lead either vertically or horizontally through the center of a block. Each street is two street objects wide, so that there are two lanes.

2. **sidewalk** Sidewalks generally border the streets. A block containing no streets will contain only sidewalk.

3. **small and large obstacles** Obstacles appear only in the street. Small and large obstacles differ only in the effect they have on cars that run into them.

4. **streetlights** Streetlights appear only in the street. Their state changes from green, to yellow, to red, and back to green at each time step. Streetlights can appear in either lane of the street, but are intended to regulate the flow of traffic in either direction regardless of which lane they appear in.

5. **turn sign** Turn signs can appear only at intersections pointing either to the left or to the right. A turn sign will only be placed on the sidewalk at the corners of intersections.

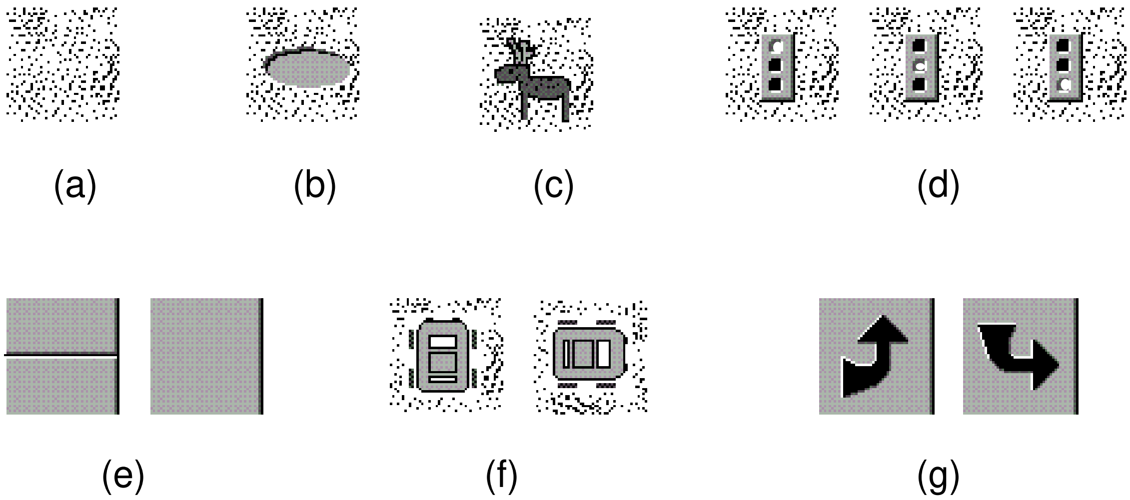


Figure 5.2: The different objects in SID. (a) street, (b) small obstacle, (c) large obstacle, (d) streetlights (green, yellow, and red) (e) sidewalk, (f) agents car, (g) street signs

6. **agent car** The agent can move throughout a block. If the agent's movement cause it to land on a sidewalk, it will bounce back to its previous position at the next time step, regardless of its action. The action executed can change the car's direction however (if the car runs into the sidewalk, and then turns right, the car will end up in it's previous position, but turned towards the right). The car will have the same behavior when running into a large obstacle or another car. Running over a small obstacle will have no effect on the next action.

5.2 The SID driver

The agent in the SID domain has actions and sensors that are highly specific to its task. In the current implementation, the agent's reward function can define up to four sub-tasks, though in any given experiment, only a subset of the sub-tasks may be part of the task as a whole. There is thus no global goal, such as reaching some specified location or finding an object — the agent's only task is to navigate through the world, coordinating the sub-tasks that are currently defined.

5.2.1 Actions

The agent has 6 actions available, as shown in Fig 5.3. Apart from knowing the number of actions available, the agent has no prior knowledge about the expected effect of the actions. Each action takes one time-step, and the agent must execute one of them at each time step (the `stop` action can be used to stand still). The actions are deterministic and always lead to the destination cell, except in two cases: 1) If the agent is "crashed", ie. a previous action has brought the agent to a cell that is occupied by sidewalk, or a large obstacle, the agent will bounce back to its previous position, no matter what action it executes. 2) The `jump-left` and `jump-right` actions cause the agent to move similarly to a knight on a chess board. Unlike in chess however, the path must be clear for the agent to reach its destination cell. Thus, if the agent attempts one of the `jump` moves, and the cell diagonally to the right or left (depending on the direction of the jump) is occupied by sidewalk or a large obstacle, the agent stops in that cell, and is considered "crashed".

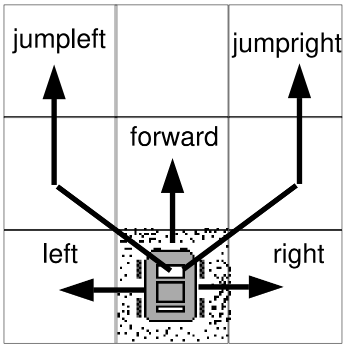


Figure 5.3: The arrows indicate five actions available to the agent in SID. The sixth action is `stop`, which causes the agent to remain in the same cell.

5.2.2 Sensors

The agent has a large number of highly specialized sensors providing information about the environment. As with the agent's actions, we do not model any form of noise, so that the information given by the sensors always accurately reflects the state of the world. The agent's sensor range is limited to a small area around it, and coordinates named in sensors are relative to a coordinate frame where the agent is at the origin (thus, $(-1, 0)$ refers to the cell to the agent's immediate left). The agent has no semantic information about the meaning of the sensors and their values, though from the agent designer point of view, a large amount of knowledge and assumptions about the domain were used to select what sensors should be provided to the agent.

Table 5.1 shows all the defined sensors.

sensor	values
`roadAt(`x`, `y`)`	`true`, `false`
`obstType`	`large`,`small`, `none`
`obstRow`	$0, 1, 2$
`obstCol`	$-1, 0, 1$
`lightColor`	`red`, `green`, `yellow`, `none`
`lightRow`	$0, 1, 2$
`signRow`	$0, 1, 2$
`signDir`	`up`, `down`, `right`, `left`, `none`

Table 5.1: The sensors available to the SID agent

The `roadAt(`x`, `y`)` sensor, is short-hand for a set of sensors, detecting the presence of a road or a sidewalk at the cells described by the x, y coordinates. The range of this sensor is limited to a 3x3 grid with the agent in the center of the bottom row (as shown in Figure 5.3). When the sensors that report `none` as one of their possible values, `obstType` for example, the corresponding sensors that otherwise report the position of the object, return 0. Though the return values shown in the table are highly descriptive, they are returned to the agent in integer form. The agent thus has no prior semantic knowledge that would enable it to interpret the different significance of the values corresponding to `none` and `large`, for example.

The design of the sensor is relatively unimportant for reinforcement learning. Given the above sensors, there are certain states that will never occur, suggesting that the sensors define a state-space that is too large. However, the agent is not learning in the space of all possible sensor value combinations, but in the space of all possible world states. We may therefore design sensors that perhaps encode the world state inefficiently, without affecting the agent's learning performance. The design of the sensors are important for the modular approach however. It must be possible to separate the sensory information into the different modules. If the sensory input is just the raw image from a camera for example, it would be impossible to separate the input into modules in any meaningful way. The modular approach depends on the availability of inputs returning information on a level of abstraction commensurate with the level at which the sub-tasks are defined.

5.2.3 The reward function

The agent does not have a global goal to accomplish, but is instead driven by a set of sub-tasks. We can describe each sub-task as a *behavior* that the agent must master and be able to coordinate with other behaviors. We have defined the following behaviors for the agent to coordinate as it is navigating through the world.

- **Driving down the road.** The agent should strive to drive straight down the road, in the right lane, and not run up on the sidewalk. It should not turn unnecessarily, or stop for no reason. This behavior will be learned by the *road module* whose inputs come from the `roadAt` sensor.
- **Avoiding obstacles** The agent should avoid all obstacles, large and small. While driving into a large obstacle constitutes a crash, a small obstacle can be considered equivalent to a pot-hole — it does not crash the car or slow it down to run over one, but in the long run the effect can be damage to the car. The *obstacle module* will learn how to avoid obstacles, using the `obstType`, `obstRow`, and `obstCol` sensors.
- **Negotiate streetlights.** The agent should not drive through red lights, and avoid going through yellow lights if possible. This behavior will be learned by the *streetlight module* using the `lightColor` and `lightRow` sensors
- **Follow street signs.** At certain intersections, arrows will point to one of the four possible directions. The agent should follow the direction of the arrow, turning if necessary. The *sign module* will be used to learn this behavior using the `signRow` and `signDir` sensors.

These behaviors are defined by the agent's reward function. We define the reward function so that by learning to maximize reward, the agent will develop a policy that coordinates all of the above behaviors. Since the behaviors are fairly independent of each other, we can define reward functions for each one. The agent's reward function will then be a linear combination of all the sub-reward functions:

$$R(x) = R_{\texttt{road}}(x) + R_{\texttt{obstacle}} + R_{\texttt{light}} + R_{\texttt{sign}}$$

In SID, the individual reward functions are defined as follows:

$$R_{\texttt{road}} = \begin{cases} 0.06 & \text{if the agent is in the right lane of the road moving forward} \\ 0.04 & \text{if the agent is in the left lane of the road} \\ -0.06 & \text{if the agent is on the sidewalk} \end{cases}$$

$$R_{\texttt{obstacle}} = \begin{cases} -0.2 & \text{if the agent is on a small obstacle} \\ -0.6 & \text{if the agent is on a large obstacle} \\ 0 & \text{otherwise} \end{cases}$$

$$R_{\texttt{streetlight}} = \begin{cases} -0.6 & \text{if the agent is passing through a red light} \\ -0.2 & \text{if the agent is passing through a yellow light} \\ 0.6 & \text{if the agent is passing through a green light} \\ 0 & \text{otherwise} \end{cases}$$

$$R_{\texttt{sign}} = \begin{cases} 0.6 & \text{if the agent is passing by an } \texttt{up} \text{ sign} \\ 0 & \text{otherwise} \end{cases}$$

References to "being on" something indicate that the agent and the object occupy the same cell. Similarly, the agent is "passing" an object if it occupies the same row (relative to the agent's orientation).

The relative magnitudes of the reward constants prioritize the behaviors. The reward values must be carefully chosen so that no behavior is ignored and ensuring that an easily attainable reward source does not dominate the agent's global behavior. In our experiments, those events considered to be important were given rewards of magnitude 0.6, and those of lesser significance were given rewards of magnitude 0.2. After the first run of experiments it was evident that the reward for the road module completely determined the agent's behavior. The road rewards were then scaled down so that despite the ease of obtaining the rewards, they did not over-shadow the rewards for the other behaviors.

Using the individual reward functions each module defines and learns its own local Q-function, Q_i. The local Q-values estimate the utility of executing actions with regard to achieving the sub-task.

6 Experimental Results

This chapter presents the results of experiments with the modular architecture in the SID domain. For simple achievement tasks, it is customary to organize experiments into sets of "runs", each consisting of several "trials". During each run, the goal configuration remains constant, and the agent retains what it has learned previously over all trials. Each trial is terminated after the goal has been reached or after some fixed time period. Since in SID the agent's task is to drive around the world as well as possible, rather than to achieve a specific goal-state, the experiments are not organized in runs or trials. Rather, each experiment consists of the agent taking a fixed number of steps, with the performance being measured continuously. Unless otherwise stated however, the results are averaged over 10 experiments.

The most frequent measurement used in our experiments is cumulative reward. While the reward is a useful to gauge relative performance between methods, it does not indicate what the agent's actual behavior is. The fact that one method leads to a lower cumulative reward than another, does not indicate how much worse we would judge the actual performance. Depending on the magnitude of the rewards, the difference can be the result of only a few mistakes, such as occasionally running a red light, or of systematic bad behavior. Therefore, in addition to the reward we also provide results showing how often certain situations (such as going through a green light, or running into an obstacle) occur. Comparing two policies and noting which states lead to different recommended actions provides a direct way of comparing the performance of a modular agent with that of a monolithic one. However, because of the non-deterministic nature of the experiments and learning procedures, neither the monolithic, nor modular strategy will learn exactly the same policy in every experiment. When policy comparisons are done, we attempt to present results representative of several experiments.

In order to evaluate the performance of the modular approach, a comparison with an optimal solution is desirable. We know that because of the design of the modular architecture that the optimal solution is unattainable, but in order to judge the quality of the approximate solution learned, some comparison must be made. Though the premise of this thesis is that the optimal solution is practically unattainable in a complex domain such as SID, we use the performance of the standard monolithic approach as our optimality measure. By running the simulator on a powerful SGI Onyx multiprocessor, extending the experiments to 30,000 steps becomes possible. We assume that this is sufficient time for the monolithic approach to reach its performance limit, providing us with the desired comparison. In a real-world domain however, extending the learning beyond a few thousand, or even a few hundred, steps would be considered excessive. For an agent moving in the real world, even taking one step can last several seconds, making thousands of repetitions seem interminable. Therefore, while the performance limits shown in the experiments are useful to gauge the quality of solutions, special attention should be paid to the system's initial behavior. The performance level reached after a few thousand steps indicates what is attainable if the modular approach is employed in a non-simulated domain.

We describe the results of the following experiments:

- A comparison of the monolithic approach with the modular architecture using "greatest mass" approximation. We show the effects of having different numbers of behaviors to learn, and compare the differences in policies learned by the modular and monolithic approaches, as well as the differences in total reward accumulated.

- A comparison of the monolithic approach with the modular architecture using the "nearest neighbor" approximation strategy.

- The effect of decaying the exploration rate. By gradually decreasing the rate of exploration the agent can focus on achieving the task rather than searching for more efficient ways. However, if exploration is decreased too quickly, only poor solutions will be found. We compare the effect on the monolithic and modular approaches using different rates of decay.

- The effect of using a modular system to initialize a monolithic one. The time at which the initialization is done is varied to see if there is a point at which the modular approach reaches a good approximation.

- The effect of an activation function. The monolithic approach is compared to a modular system using a pre-defined activation function.

The results indicate that the modular strategy very quickly learns a good policy for driving through the world. The monolithic strategy learns a slightly better policy given enough time, but the benefit of the improved policy is overshadowed by the increase in learning time.

6.1 Modular vs. Monolithic

6.1.1 Greatest mass

Figure 6.1 shows five sample states with the action prescribed by the greatest mass policy learned at the end of the experiments in a domain with a road, obstacles, and streetlights. The selected states are not intended to be representative of the most "important" or most frequently encountered states, but simply illustrate a few situations that might occur.

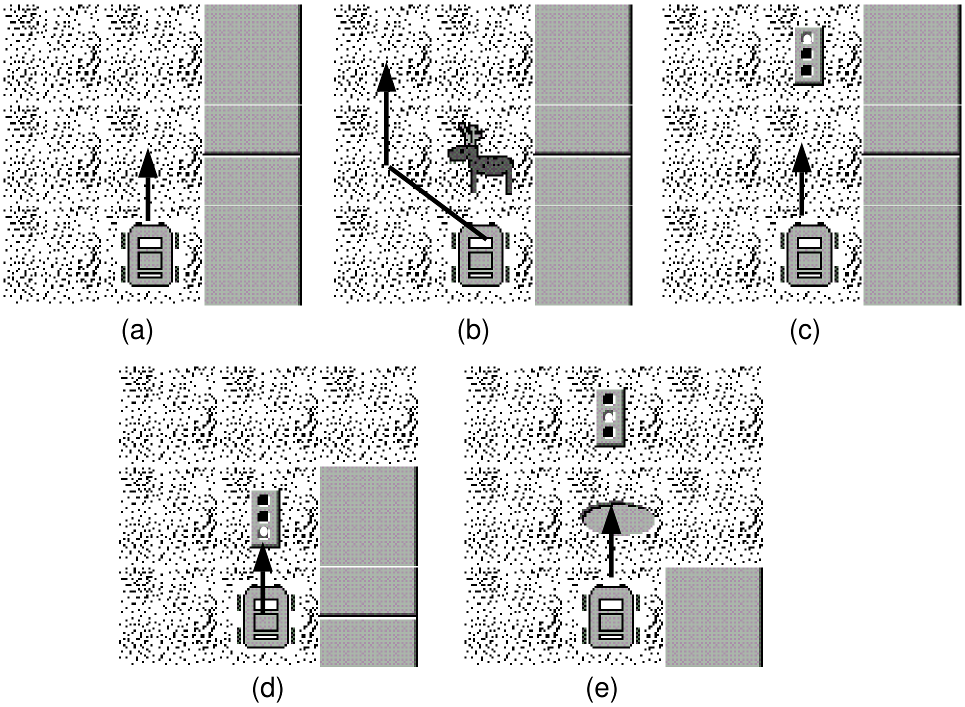


Figure 6.1: A few sample states and the action learned by the greatest mass policy. The grey circle in the streetlight indicate its current state, (ie. green, yellow, and red from bottom to top). The light changes color at each time step, so by moving forward when a green light is seen in (d), the agent will pass through the light when it is yellow. In the states shown, the modular policy has learned appropriate actions. In (e), the agent crosses over a small obstacle, but avoids the more severe mistake of running a red light.

As the figure shows, the modular policy picks appropriate actions in these sample situations. In particular, it avoids the large obstacle, and does not run through the red light. Though the agent runs the yellow light in (d), it favors crossing a small obstacle to running a red light (avoiding the obstacle with a `jump-left` action, for example, places the agent by the streetlight as it turns red.) (e). This is consistent with our stated goals of learning a policy that attempts to partially satisfy all sub-goals if the optimal performance is unattainable. Viewing the simulation in progress also bears this impression out.

For a more quantitative measure of the modular policy's performance we can examine the cumulative reward as well as simply counting the number of times significant events occur. Figure 6.2 compares the cumulative reward of agents using the modular and monolithic approach, in a domain with a road, obstacles, and streetlights.

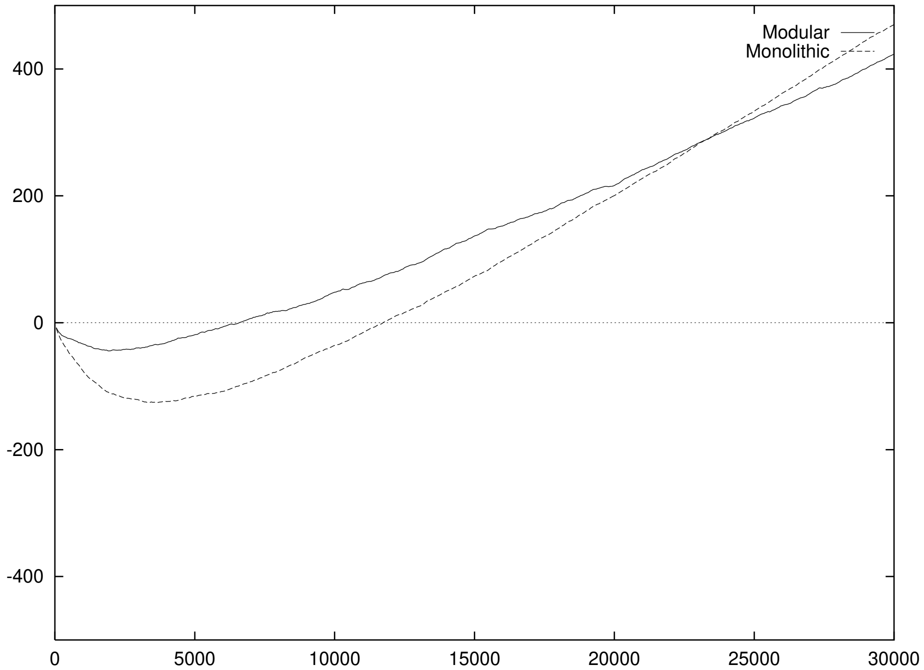


Figure 6.2: Cumulative reward over time for modular and monolithic agents. In this experiment, the agent needed to learn the road, obstacle, and streetlight behaviors. The large negative cumulative reward gathered by the monolithic agent in the first 3000 steps indicate its poor initial performance. In contrast the modular agent does well initially, and quickly reaches a level of performance where its net reward gain is positive.

Initially, the modular agent's performance is better than that of the monolithic agent, and quickly reaches a limit. This performance limit is indicated by the constant slope of the graph, suggesting that the modular approach quickly learns the best policy possible using the greatest mass policy. The monolithic agent gradually improves and eventually surpasses the modular agent's rate of reward intake.

Thus the modular agent achieves better initial performance and faster learning, but learns an apparently slightly worse policy than that ultimately reached by the monolithic agent. To illustrate the differences in the policies learned, we can look at the individual reward functions. Figures 6.3,6.4, and 6.5 show the cumulative rewards from the three individual reward functions.

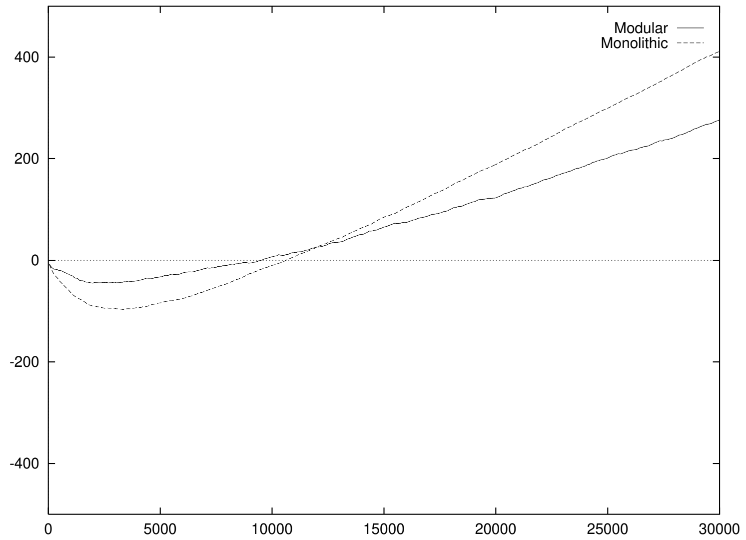


Figure 6.3: Cumulative reward for modular and monolithic agents generated by the road reward function only. Though the agent is learning road, obstacle, and streetlight behaviors, the graph shows that it is differences in the 'road' behavior that accounts for most of the overall difference in global performance.

These figures indicate that the modular policy performs as well as, or better than, the monolithic policy in terms of avoiding obstacles and negotiating streetlights. The largest difference however, is due to less reward being gained from the road reward function.

Differences in cumulative reward gathered during an experiment do not provide much information on how the two policies differ in quality. We can count the number of times the agent entered a state with certain properties considered good or bad by the reward function. Figure 6.6 shows the number of times the agent entered a state considered less than optimal by the road reward function. This includes all states where the agent is not in the right lane of the road, moving forward. As can be seen, the main differences in the modular and monolithic policies is that the modular agent spends a larger amount of time in the left lane, and also spends slightly more of its time not moving forwards (ie. turning or standing still). However, in terms of crashing into the sidewalk the modular agent performs as well as, or better than, the monolithic agent. Thus the modular agent

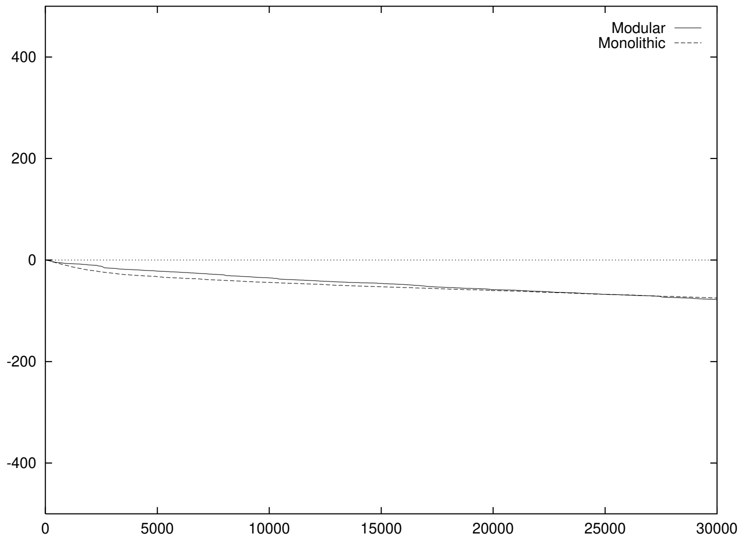


Figure 6.4: Cumulative reward for modular and monolithic agents generated by the obstacle reward function only. The modular agent performs better than the monolithic agent initially, and quickly reaches a level of performance which is eventually equaled, but not surpassed, by the monolithic agent.

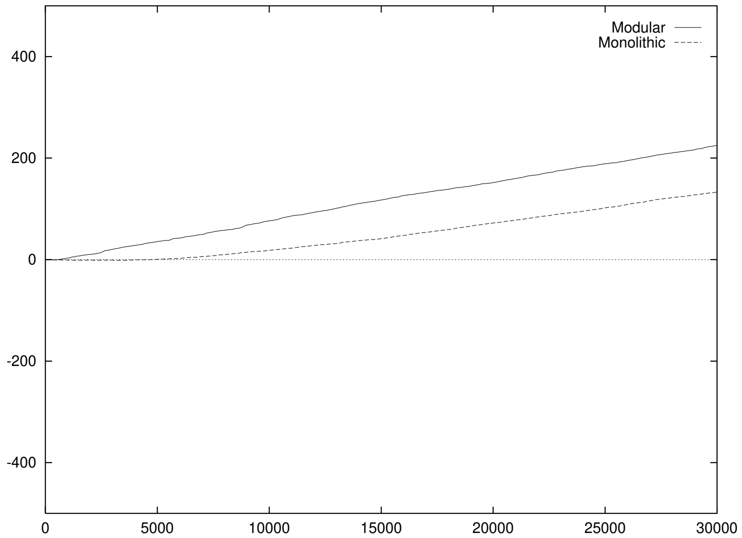


Figure 6.5: Cumulative reward for modular and monolithic agents generated by the streetlight reward function only. The modular agent performs better than the monolithic agent which requires longer time to reach the same level of performance.

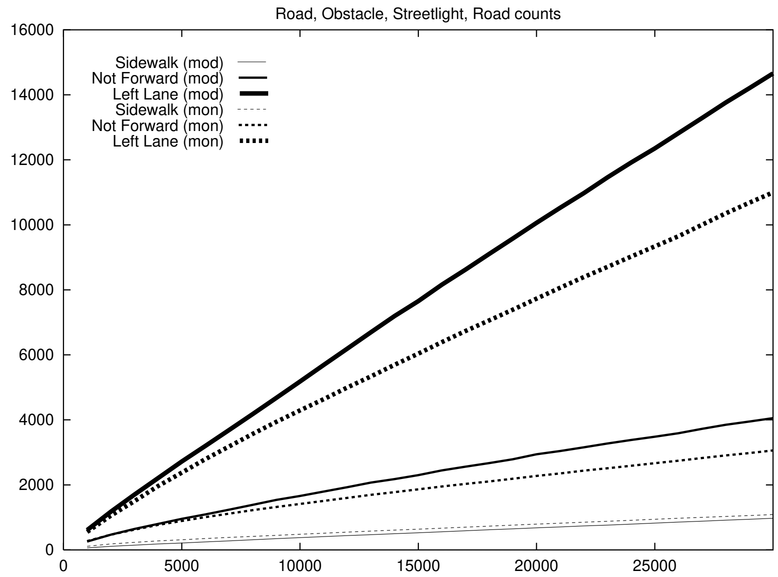


Figure 6.6: Counts of events significant to the road reward function over time. The graph shows that over 30,000 steps, the modular agent spent almost 15,000 in the left lane. While this relatively minor error occurred often, the agent crashed into the sidewalk only around 1,000 times.

accomplishes the most important part of the sub-task (crashing into the sidewalk generates the largest magnitude reward), while making some compromises on the other parts of the task.

Examining the counts of events relevant to the other modules reveals that the modular agent's performance is close to that of the monolithic agent in terms of achieving their associated sub-goals. Figure 6.7 shows how often the modular and monolithic agent run into small and large obstacles. Again, the modular agent's superior initial performance is evident, and the performance difference at the end of the experiment does not seem significant.

Figure 6.8 compares the modular and monolithic agent's performance with regards to streetlights. Though the modular agent runs through more red lights than the monolithic one, it does better with yellow and green lights (leading to a higher cumulative reward, as seen in Figure 6.5). It is interesting to note that the modular agent encounters around 100 streetlights more than the monolithic agent, suggesting that its policy may be seeking out streetlights in order to gain reward (though a difference of 100 events over 30000 steps may not be significant).

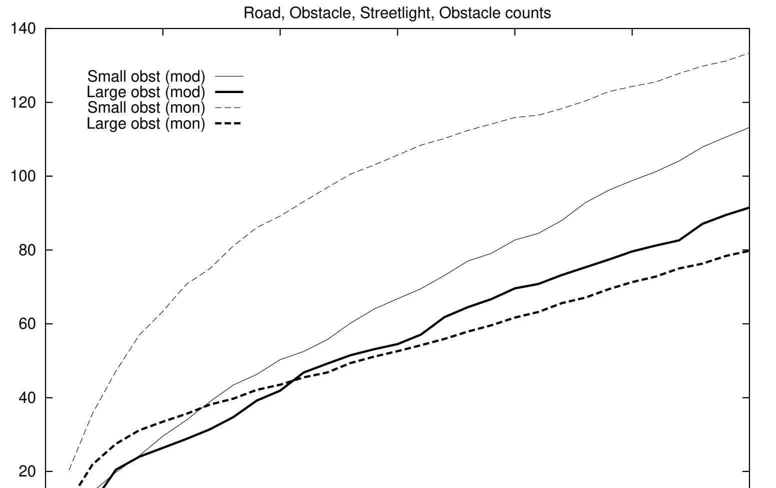


Figure 6.7: Counts of events significant to the obstacle reward function over time. The modular agent quickly learns to avoid obstacles almost as well as the policy eventually learned by the monolithic agent.

It is clear therefore that the modular agent learns a policy that is almost as good as that of the monolithic agent. Furthermore, the modular policy is learned quickly, leading to a much better initial performance than the monolithic policy.

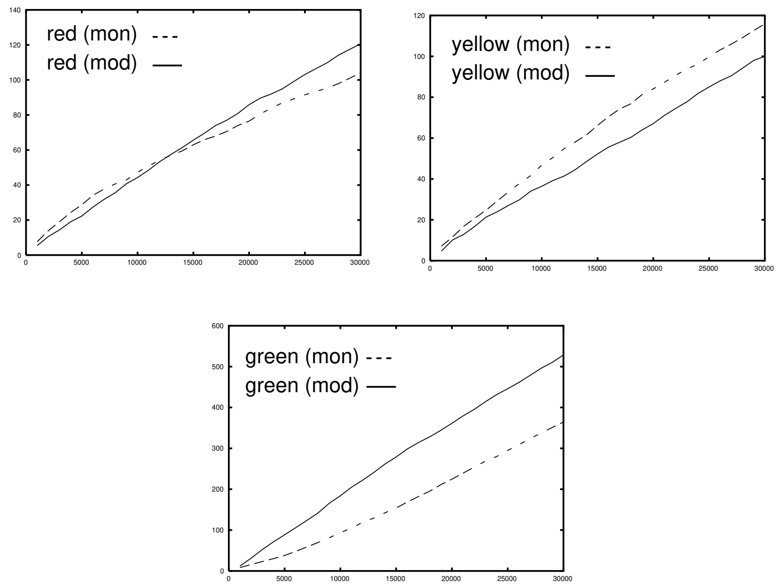


Figure 6.8: Counts of events significant to the streetlight reward function over time. The modular and monolithic agents seem to avoid going through red and yellow lights almost equally well. The modular agent seems to be more adept at finding green lights to go through, however.

To further evaluate how much the policies learned by the modular and monolithic agents differ, it is useful to count the number of states in which the two different policies recommend different actions. However, for this count to be meaningful, the number of times a state is visited must also be considered. Though the modular and monolithic policies might agree over 90% of the states, if the agent spends most of its time in the remaining 10% of the states, the modular policy will probably not compare well to the monolithic one.

The number of all possible states, ie. the size of the global state-space, can be computed by counting all possible combinations of the inputs listed in Table 5.1. Considering only the road, obstacle, and streetlight behaviors, the number of states is slightly less than 100,000. Depending on the domain however, the number of states that can actually occur is far less. In SID, for example, the agent has 9 sensors detecting the presence of a street or a sidewalk in a 3x3 grid, leading to a theoretical 512 possible different configurations. However, because the roads are constrained to be a of a fixed width with no dead-ends, the number of possible configuration of those sensors that can actually appear in the domain is 27. Therefore, the number of states actually encountered by the agent can be several orders of magnitudes smaller than the theoretical size of the state-space. Without extensive domain knowledge however, it is usually difficult to predict the actual state-space size without experimentation.

When comparing policies of modular and monolithic agents in the same experimental domain, in a run lasting 30,000 steps, we find that both agents encounter around 300 states. Out of these states, the policies have conflicting recommendations for about 100 states. However, the agent does not visit each state an equal number of times. In an average run, the modular agent disagrees with the monolithic one less than a third of the time. However, more than 50% of those time steps where the modular agent acted differently than the monolithic one, were spent in 5 or 6 states. The vast majority of states where the two policies differ, are states the agent encountered less than 1% of the time of the experiment. Figure 7.1 shows an example of the states where the policies differ, responsible for the majority of performance decrease. It should also be noted, that among all the states were policies differed, the worst error made by the modular agent, is running over a small obstacle. This is a rare occurrence, and most of the time the error is making an unnecessary turn, stopping, or switching into the left lane.

In this instance therefore, it seems that most of the difference in performance between the modular and monolithic policies are caused by the modular policy failing in a very small number, but frequently visited, states.

In the previous experiments, the agent learned three behaviors. Below we examine the performance of the modular approach when the number of behaviors to be learned is varied. First, a modular agent's performance in learning only two behaviors is examined, followed by experiments with four behaviors.

In experiments with only two modules, the performance of the modular approach is not significantly different from that of the monolithic. For example, Figure 6.9 shows the cumulative rewards in experiments where the agent was dealing only with the road and obstacles. As before, the modular agent has better initial performance, but it reaches a level of performance that is not surpassed by the monolithic agent.

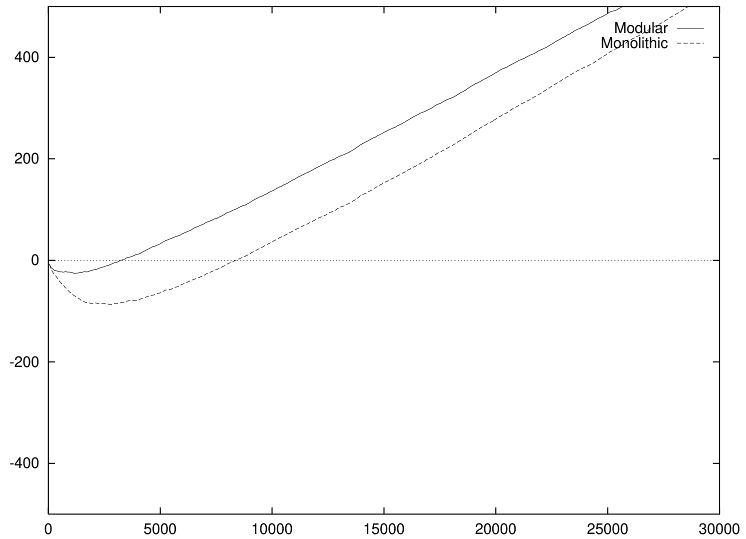


Figure 6.9: Cumulative rewards for modular and monolithic agents over time. In this experiment only the road and obstacle behaviors needed to be learned. When only two behaviors are present, the modular approach learns a solution as good as that of the monolithic agent.

However, adding a fourth behavior, that of following street signs, drastically worsens the performance of the modular agent. Figure 6.10 shows the cumulative rewards in experiments where all four behaviors are necessary.

Figures 6.11, 6.12, 6.13, and 6.14, show the rewards from the individual reward functions, illustrating that the the main difference lies in the rewards related to driving on the right side of the road. Figure 6.15 shows that not only is the modular agent spending more time in the left lane of the road, it is making a lot more turns as well as stopping more (ie. not moving forward) than the monolithic agent. Thus, while the modular agent is making errors of limited severity (ie. associated with small negative rewards), the large number of these errors produce the large difference in cumulative reward when compared to the monolithic agent. As before, however, the majority of the mistakes were made in a small set of states. As described in chapter 7 in more detail, these mistakes are caused by the sign module not having enough inputs to disambiguate between

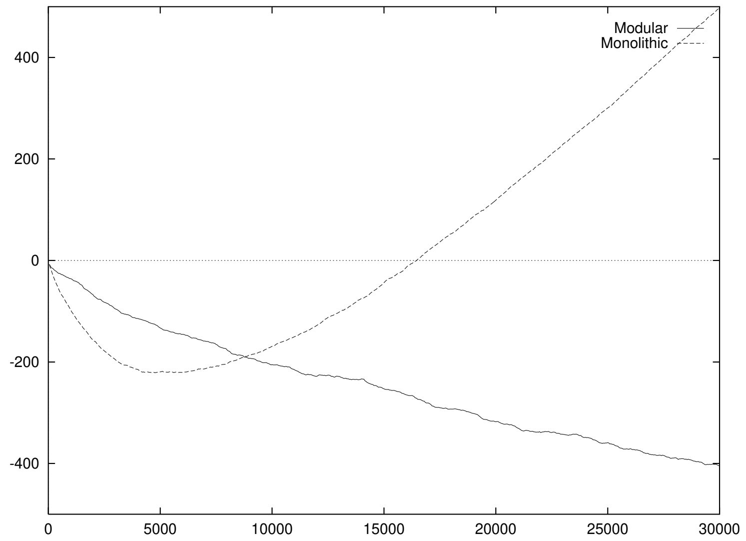


Figure 6.10: Cumulative rewards for modular and monolithic agents over time. In this experiment the road, obstacle, streetlight, and sign behaviors needed to be learned. The addition of the sign behavior worsened the performance of the modular agent significantly. It should also be noted that the initial performance of the monolithic agent is much worse than in experiments with fewer modules, indicating that the task has gotten more difficult.

certain states. Since the optimal actions in those states conflict with each other, the agent makes mistakes most of the time those states are encountered.

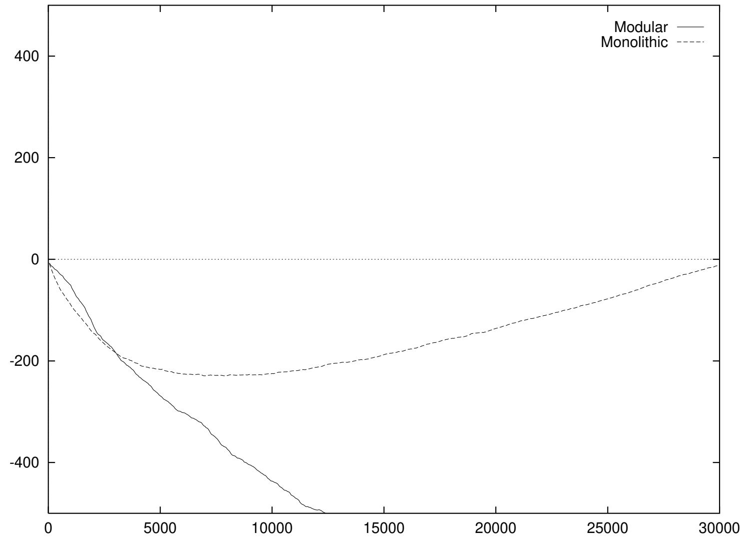


Figure 6.11: Cumulative reward for modular and monolithic agents generated by the road reward function only. The experiment included roads, obstacles, streetlights, and signs. It is clear that modular agent is not learning the road behavior satisfactorily.

6.1.2 Nearest neighbor

The other approximation strategy we have discussed is "nearest neighbor". Nearest neighbor performed well in a simple grid world domain, because of its similarity to a nearest neighbor search strategy. The strategy essentially focuses on one sub-goal completely, until it has been accomplished. Such an approach does not work well in a domain where two or more sub-goals must be considered concurrently. Figure 6.16 shows the cumulative reward of a modular agent using nearest neighbor compared with a monolithic agent.

Figures 6.17, 6.18, and 6.19, show that the road and obstacle reward functions are the source of the large negative rewards being accumulated by the modular agent. This results from the nearest neighbor approximation completely ignoring negative Q-values in favor of positive ones. Thus, if the obstacle module assigns negative utility towards going forwards, but the road module assigns a large positive reward to the same action, the agent will go forward, crashing into the obstacle, since nearest neighbor simply picks the action with the highest utility.

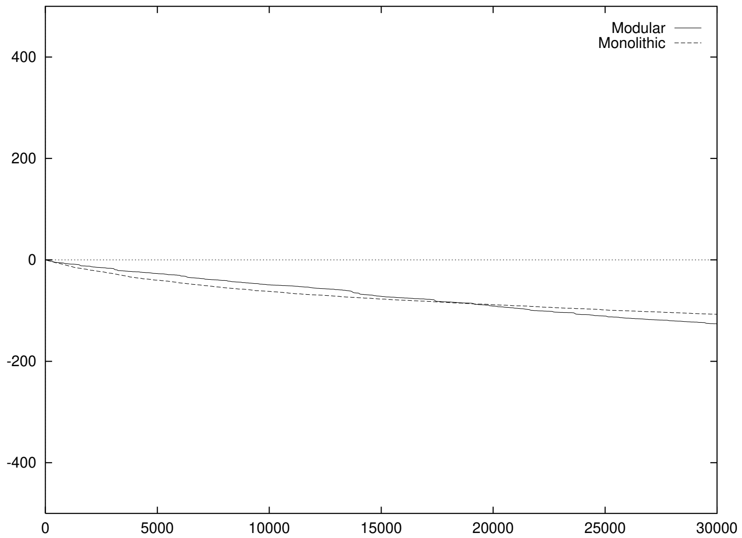


Figure 6.12: Cumulative reward for modular and monolithic agents generated by the obstacle reward function only. The performance of the two agent is similar, indicating that the obstacle behavior is not a cause for the difference in global performance.

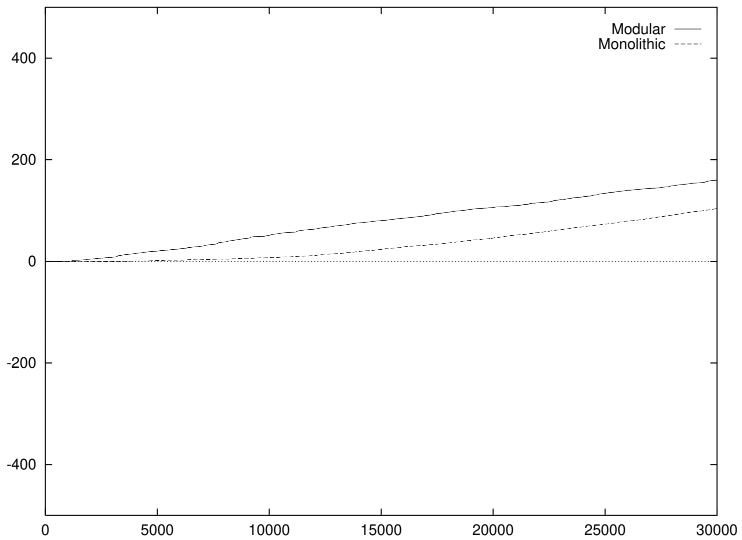


Figure 6.13: Cumulative reward for modular and monolithic agents generated by the streetlight reward function only. As in the case with only three behaviors to be learned, the modular agent achieves a slightly better initial performance before the monolithic agent catches up.

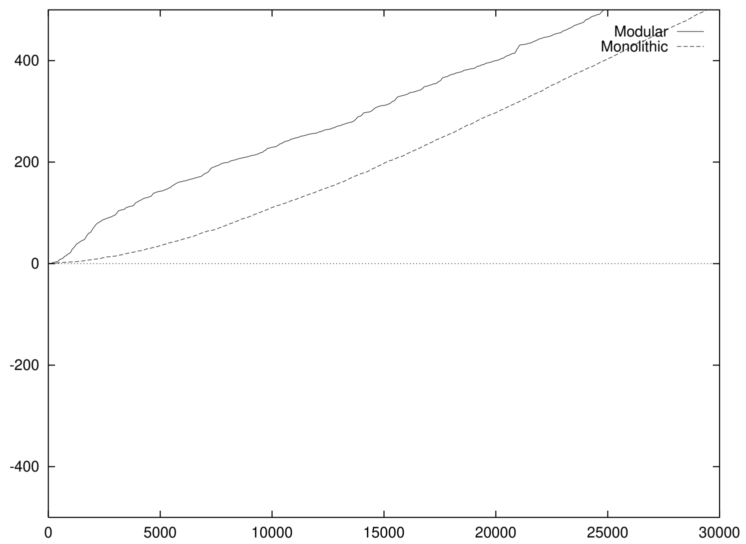


Figure 6.14: Cumulative reward for modular and monolithic agents generated by the sign reward function only. Again, the modular agent achieves a better initial performance in following the signs. This behavior does therefore not contribute to the difference in global performance.

It is thus clear that the nearest neighbor strategy is only useful in domains where the goals are all goals of achievement, and the agent can safely achieve each goal separately without considering the other sub-goals.

6.2 Decaying exploration rate

Exploration has a large effect on the experiments in this domain. In our experiments exploration is done by picking a random action at each step with some probability p, instead of the action recommended by the policy. By decaying this probability, one can ensure that the agent spends sufficient time initially to discover action sequences leading to improved performance, but then spends less time exploring as time goes on. This strategy of gradually diminishing exploration is in accordance with what is known about so called "two-armed bandit" problems [Feldman, 1962], for which it has been demonstrated that while it is good to explore initially, as time passes it is better to select the action that is thought to be optimal with an increasing frequency.

Figure 6.20 shows the effect of decaying the exploration rate for the modular system. A relatively slow rate of decay (0.0001) leads to the best performance, allowing the agent sufficient time to explore the domain, before settling on a policy.

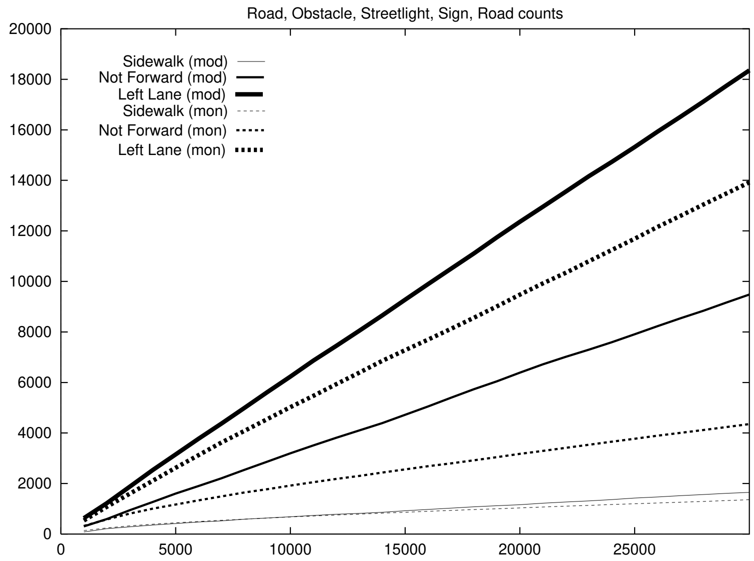


Figure 6.15: Counts of events significant for the road reward function.The experiment included roads, obstacles, streetlights, and signs. Though the modular agent spends more time in the wrong lane and making unnecessary turns than the monolithic agent, the two methods avoid crashing into the sidewalk equally well. This is yet another indication of the modular agent's erring when small reward magnitudes are involved, while performing correctly with regards to large reward magnitudes.

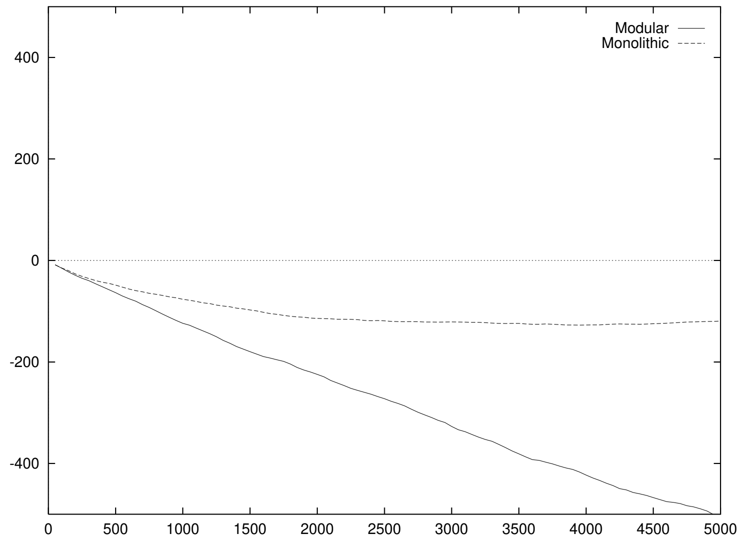


Figure 6.16: Cumulative reward for modular and monolithic agents. The modular agent used the nearest-neighbor approximation strategy. Note that the graph only extends to 5,000 time steps. In contrast to the experiments using the greatest mass approximation strategy, the modular agent does not seem to have learned the task at all. This suggests that for the SID domain, nearest-neighbor is not a good approximation strategy.

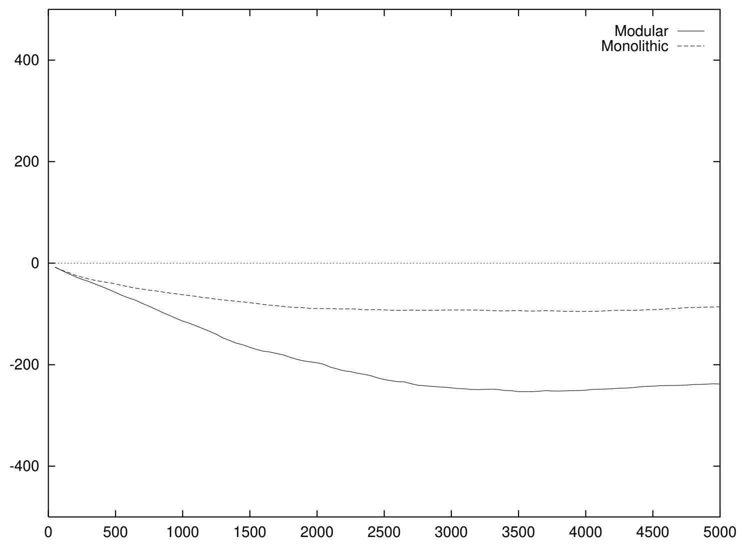


Figure 6.17: Cumulative reward for modular and monolithic agents generated by the road reward function only. The modular agent used the nearest-neighbor approximation strategy. Though the modular agent does not perform as well as the monolithic one relative to the road behavior, this only partially accounts for the vast difference in global performance.

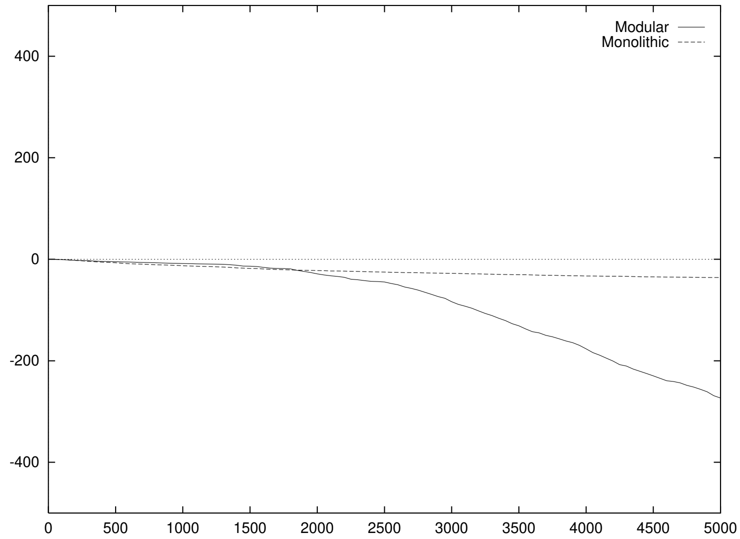


Figure 6.18: Cumulative reward for modular and monolithic agents generated by the obstacle reward function only.The modular agent used the nearest-neighbor approximation strategy. The modular agent's failure to deal with obstacles is shown by the consistent accumulation of negative rewards. It is this inability to avoid obstacles that is the main cause of the difference in global performance between the modular and monolithic agents when the nearest-neighbor approximation strategy is used.

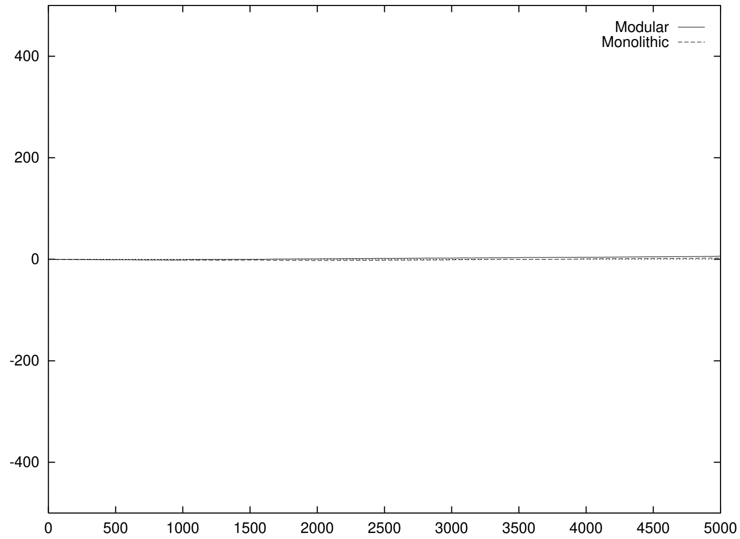


Figure 6.19: Cumulative reward for modular and monolithic agents generated by the streetlight reward function.The modular agent used the nearest-neighbor approximation strategy. There is no difference in performance between the modular and monolithic agents. However, the modular agent using the greatest mass approximation strategy performs slightly better, indicating that the difficulties in avoiding obstacles when using nearest-neighbor is hindering the agent from reaching any streetlights.

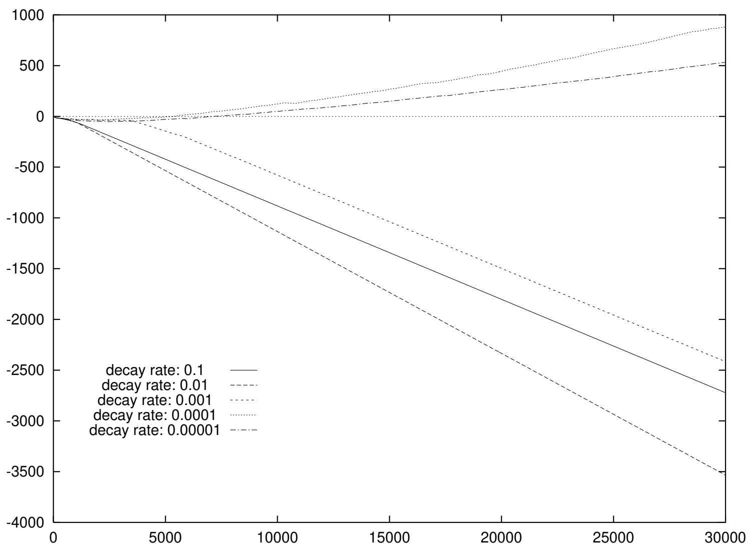


Figure 6.20: Decaying the exploration rate for the modular agent. If exploration is turned off prematurely, the quality of the learned policy suffers. If exploration is continued too long, performance is hurt by taking unnecessary sub-optimal steps.

Figure 6.21 shows that with the exploration rate seen to be best for the modular strategy, both the modular and monolithic approaches improve their performance. This improvements is due to the avoidance of unnecessary, sub-optimal, exploration steps in the later parts of the experiment. Examining the event counts shows how each behavior is affected, when compared with the figures with exploration performed continuously (Figures 6.6-6.8). Figure 6.22 shows that the road behavior does not benefit much from the decaying exploration rate in the modular agent, except that the rate of running into the sidewalk is reduced even further.

In the case of obstacle avoidance, Figure 6.23, the monolithic approach almost completely eliminates any collisions as the exploration rate is decayed. The modular approach also improves, with the small obstacle collision rate approaching that of the monolithic agent. The rate at which large obstacles are avoided is also significantly flattened. The steep increases in the slope in the curve for the large obstacles are due to the large variance in the data for the modular approach. When viewed individually, the data indicate that there is typically one large steep increase in collisions with large obstacles, but not at any fixed point in time. Averaging the data produces several small steep increases in the slope of the curve. After the increases however, all experiments show the curve to be generally flat. This may indicate that the exploration rate is decayed too quickly in terms of the obstacle module.

Figure 6.24 shows that for the streetlight module, the modular approach im-

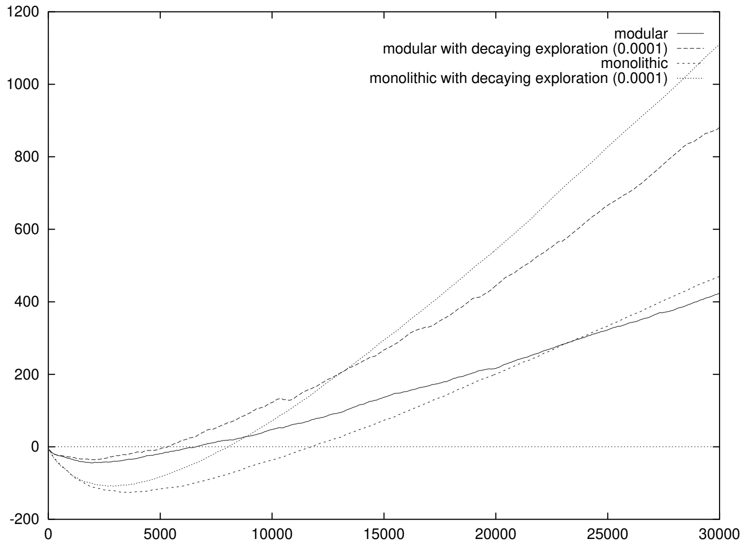


Figure 6.21: Decaying the exploration rate improves both modular and monolithic performance. This indicates that some of the errors made by the modular policy are not due to inherent limitations of the approach, but are simple caused by making unnecessary exploratory actions.

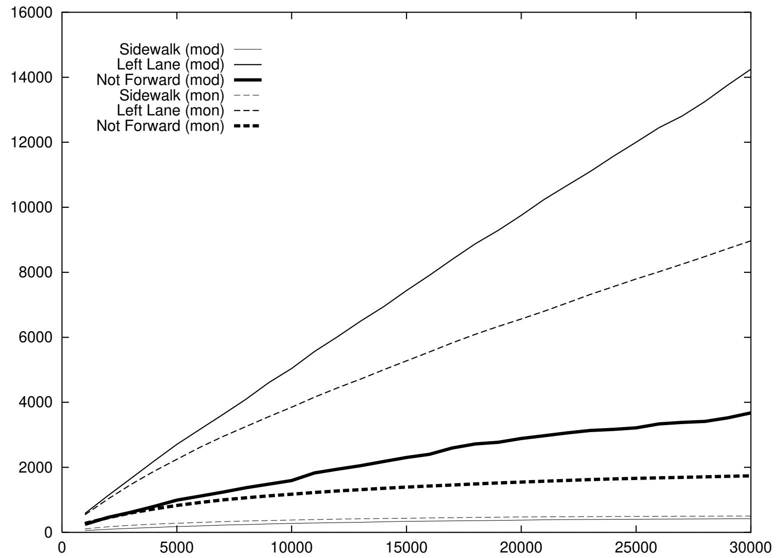


Figure 6.22: Counts of events significant to the road reward function over time, as the exploration rate is decayed. Eliminating exploration does not greatly reduce the amount of time spent in the wrong lane by the modular approach, but further limits the number of crashes into the sidewalk.

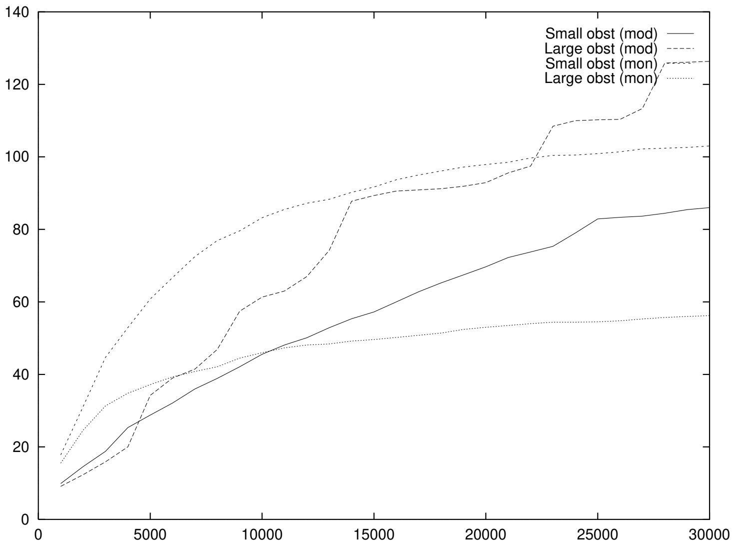


Figure 6.23: Counts of events significant to the obstacle reward function over time, as the exploration rate decays. The number of times running into both large and small obstacles is reduced in the modular policy when exploration is eliminated.

proves slightly in its ability to go through green lights, while the rate at which red and yellow lights are encountered is significantly lessened.

As the graphs show, in most cases the monolithic agent improves similarly when the exploration rate is decayed. However, these results clearly indicate that the quality of the policy learned by the modular agent is better than indicated in the experiments with continuous exploration. The rate of several undesirable events is greatly reduced as exploration is gradually eliminated, showing them to not be the result of the limitations of the modular approach as such, but simply caused by taking unnecessary exploratory actions.

6.3 Improving the modular approach

The above experiments indicate that of the differences between the policies learned by the monolithic and modular approaches, a small set of states accounted for most of the performance difference. We therefore attempted different mechanisms to extend the modular approach, trying to take advantage of the fast learning time of the modular system, but spending extra resources in order to learn better actions in the few significant failure states. These approaches, described in section 4.3, are using modules to initialize a monolithic system and using activation functions.

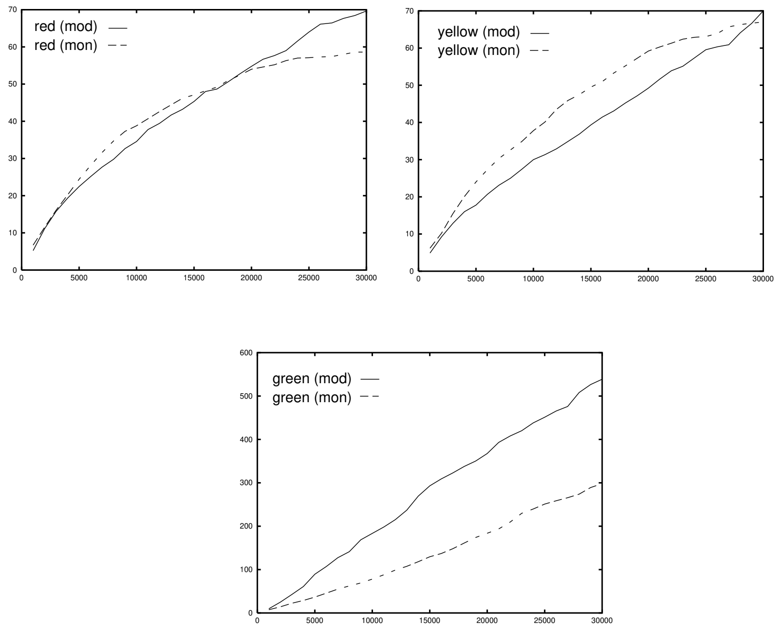


Figure 6.24: Counts of events significant to the streetlight reward function over time. The modular policy negotiates red, green, and yellow lights better when exploration is eliminated.

6.3.1 Using modules for initialization

As described in section 4.3.1, since the modular system learns a good policy relatively quickly, using the modular policy as an initialization for a monolithic system could improve learning time. The modular policy differs from the monolithic one in only a few significant states, so only those states would have to be relearned by a monolithic system initialized with a modular policy.

However, Figures 6.25 and 6.26 show that the monolithic system relearns the entire policy.

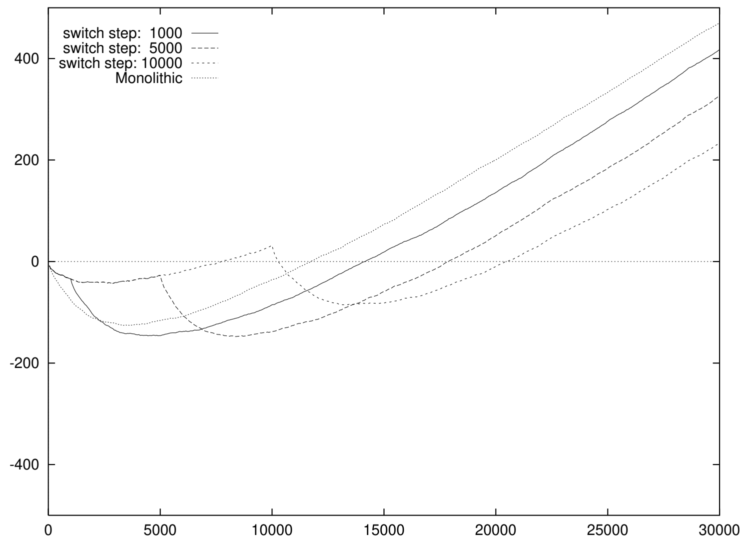


Figure 6.25: Cumulative rewards for monolithic and modular agents, where at different time steps the policy learned by the modular agent is used to initialize a monolithic system. In this experiment, road, obstacle, and streetlight behaviors needed to be learned. The graphs indicate that there is no advantage to using the learned modular policy as an initialization for the monolithic system. It appears as if the agent completely relearns the policy after the switch, thereby losing the advantage of the better initial performance.

As described in Chapter 7, the modular approximation tends to produce inflated Q-values. When these are used to initialize the monolithic system, relearning is necessary to lower the magnitudes of the Q-values appropriately.

Table 6.1 shows the Q-values for a state where the modular approximation Q_{gm} leads to the same action being recommended as in the monolithic system. However, the approximations are much larger than the Q-values learned by the monolithic system. Thus, when a monolithic system is initialized by the approximations learned by the modular system it will start to decrease the Q-values.

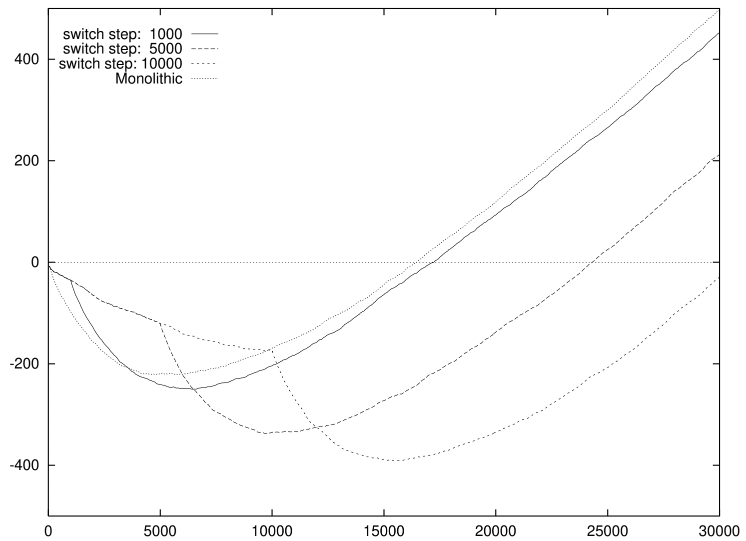


Figure 6.26: Cumulative rewards for monolithic and modular agents, where at different time steps the policy learned by the modular agent is used to initialize a monolithic system. In this experiment, road, obstacle, streetlight, and sign behaviors needed to be learned. As before, the agent completely relearns the policy after switching the a monolithic system.

	Q-values					
Policy	`stop`	`forward`	`left`	`right`	`jump left`	`jump right`
Road	0.4096	0.5602	0.2515	-0.2357	0.5188	-0.0941
Obstacle	0.2195	0.2017	0.2241	0.2079	0.2209	0.2103
Streetlight	0.5126	0.4614	0.5078	0.5096	0.4614	0.4614
Q_{gm}	1.1418	1.2233	0.9834	0.4818	1.2012	0.5775
Monolithic	0.4388	0.5705	0.2809	-0.1966	0.5597	-0.0224

Table 6.1: Q-values for a state where modular and monolithic policies agree on the best action. The modular approximations of the monolithic Q-values are of much greater magnitude however, so that when the modular policy is used to initialize a monolithic system, the agent must explore all actions, in effect relearning the policy, in order to bring down the Q-values to their appropriate levels.

Once the optimal action has been executed a sufficient number of times, its Q-value will no longer be maximum in that state. The other actions will then be selected, until the optimal action is the maximum again. This cycle will repeat until the Q-values are decreased to their correct magnitude.

Therefore, using the modular policy to initialize a monolithic system leads to relearning of the entire policy, rather than just the small set of states where the modular policy differs from the monolithic one.

6.3.2 Using activation

Many of the states in which the modular policy differs from the monolithic do not seem to be ones where all modules can provide relevant information. The states shown in figure 7.1 are almost all cases where the modular agent makes a mistake when there are no obstacles, streetlights, or signs present. Thus, the differences in behavior do not seem to appear when there are strong reward sources available nearby, but rather seem to be caused by strong, but not immediately relevant, reward sources interfering with weaker ones. Thus, the road module is overwhelmed by the streetlight module, leading the agent to jump to the left lane of the road.

If we have sufficient domain knowledge to determine *a priori* when a module is relevant or not, we can design an activation function, ensuring that modules do not inappropriately influence the action selection procedure. In SID, it is reasonable to assume that the obstacle, streetlight, and sign modules need only be active if any obstacles. streetlights, or signs are currently in view. Figures 6.27, 6.28 and Figure 6.29 compare the cumulative reward gathered by a monolithic agent and a modular agent using an activation function based on the above assumption. The figures show the results for experiments with the road, obstacle, and streetlight modules, with the road, obstacle, and sign modules, and with the road, obstacle, streetlight, and sign modules.

Using the activation function resulted in dramatic improvements in all cases. In the experiments with just the road, obstacle, and streetlight modules, the modular agents performance exceeds that of the monolithic agent. However, when the sign module is involved, the modular agent's performance still lags behind that of the monolithic agent. This is due to the problems caused by ambiguities in the reward functions at road intersections, discussed in more detail in chapter 7. Since signs appear only at intersections, the activation function cannot help in reducing this problem.

Though the hand-coded activation function worked well in SID, in other domains the information needed to design such a function may not be available. Furthermore, using an activation function will clearly prevent the agent from learning certain policies encoding unexpected strategies. In SID for example, if

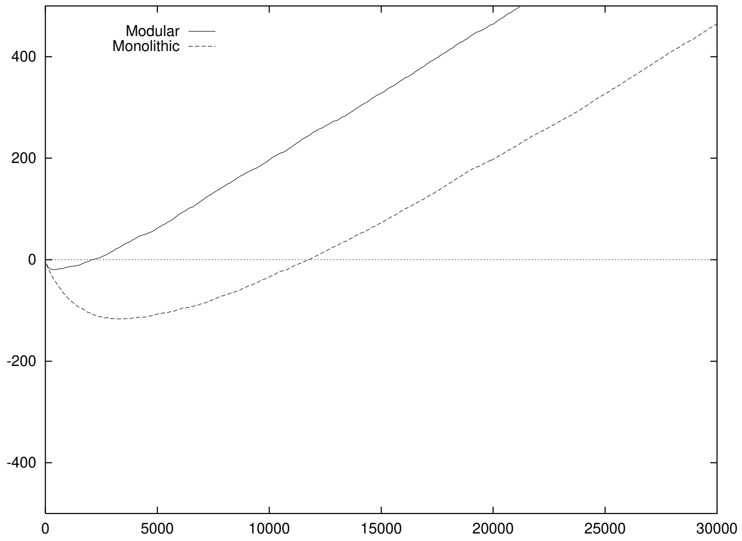


Figure 6.27: Cumulative rewards for monolithic and modular agents, with activation. Road, obstacle, and streetlight behaviors were learned. The activation function dramatically improved performance of the modular agent.

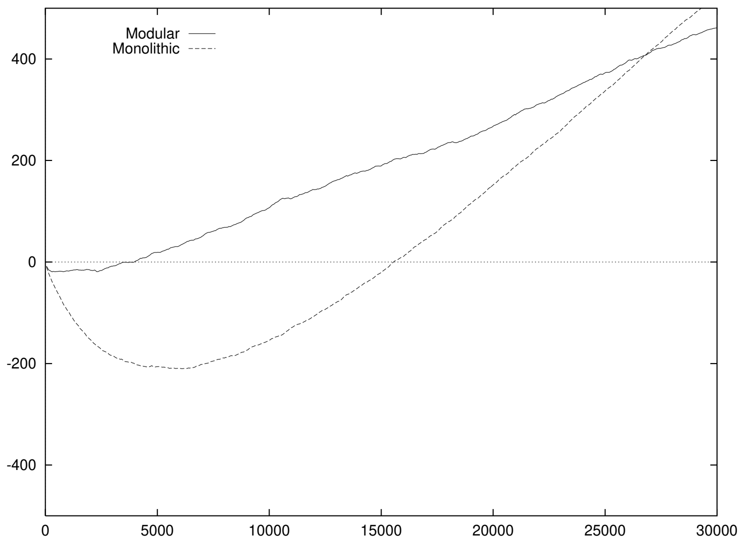


Figure 6.28: Cumulative rewards for monolithic and modular agents, with activation. Road, obstacle, streetlight, and sign behaviors were learned. The modular agent's performance is improved by using the activation function, but is still inferior to the monolithic agent's performance.

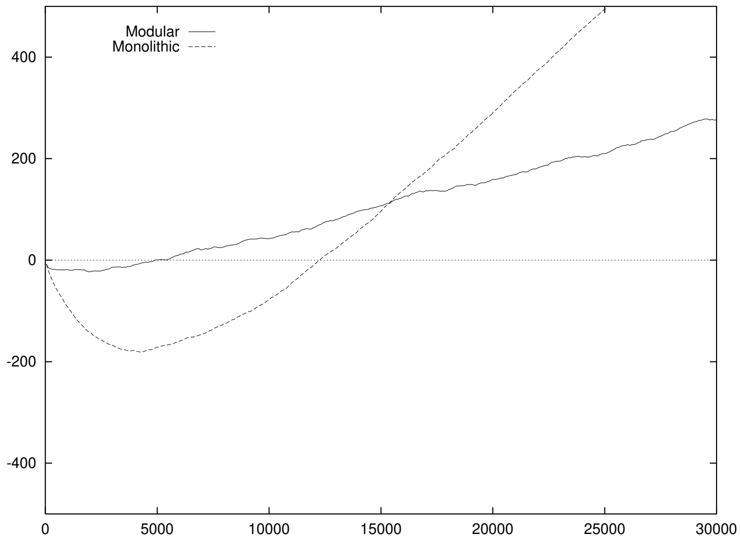


Figure 6.29: Cumulative rewards for monolithic and modular agents, with activation. Road, obstacle, and sign behaviors were learned. As before, the modular agent's performance improves, but does not exceed that of the monolithic agent. This is due to ambiguities resulting from the definition of the sign module, which are not resolved by the use of an activation function.

obstacles tended to appear in the right lane with a high enough frequency, an optimal policy might steer the agent to the left lane, even if no obstacles were in sight. The activation function used in the above experiments would prevent such a policy from being learned.

6.4 Summary

The experiments show that the modular approach, with the greatest mass approximation function, learns a policy that performs well compared with that learned by the monolithic agent. In all experiments, the modular policy is learned faster than the monolithic one. In experiments with two behaviors, the quality of the policies learned by the modular and monolithic is virtually the same. When learning the three behaviors associated with roads, streetlights, and obstacles, the modular policy receives less cumulative reward than the monolithic. However, when examining the actual events that occur during the experiments, it is clear that the modular agent handles streetlights and obstacles almost as well as the monolithic agent. The main difference in behavior is a tendency of the modular agent to drive on the left side of the road. However, for all major sub-goals, such as avoiding the sidewalk and obstacles, not running red lights and going through green lights the modular agent does as well as or better than the monolithic agent. When trying to learn the sign behavior in addition to the other three behaviors the modular agent's performance worsened. As before, the differences between the modular and monolithic policies are relegated to sub-goals of lesser magnitude (ie. not moving forward and driving on the left side of the road). Otherwise, the modular agent accomplishes the major sub-goals as well as the monolithic one. It is clear therefore that the modular agent learns a policy that achieves most of the sub-goals, and partially satisfies some of them. The small sacrifice in performance is well worth the big improvement in learning time and computational resources needed.

The nearest neighbor approximation strategy performed poorly because its inability to take negative utilities into account. The fact nearest neighbor strategy performed well in the grid-world domain, demonstrates the importance of selecting an approximation function that matches the types of tasks in the domain.

Since the modular agent's performance could be improved by correcting errors made in a small set of states, two extensions to the approach were attempted. The first, initializing a monolithic system with a learned modular policy, did not perform well. It was intended that the initialized monolithic system would only need to relearn the policy in those few states were mistakes were made. However, because the Q-values generated by the modular approach differed in their magnitude from those of the monolithic system, the entire policy had to be relearned in order to generate the correct Q-values.

The second extension, using activations, greatly improved the modular performance. This method used hand-designed activation levels to allow the agent to ignore information from "irrelevant modules". Since in many of the states where the modular policy differed from the monolithic at least one of the street light and obstacle modules was deemed irrelevant (due to the absence of any street lights or obstacles), the agent could ignore those modules. By thus being able to consider fewer modules, the agent lessened the amount of information needed to be considered, thereby reducing the risk for error. The result was a policy that performed as well as the monolithic one in the case of three modules, and a large improvement in performance in the case of four.

Thus, the modular approach can lead to good performance in a complex domain such as SID. The inherent limitations of the modular approach leads to sub-optimal behavior in some situations, though still partially satisfying the subgoals. In the following chapter we describe the types of approximation errors made by the modular algorithm, and provide an analysis of their cause.

7 Analysis

An agent using modular reinforcement learning can learn how to drive in the SID domain much faster than an agent using standard, monolithic, learning. The behavior learned however, is not optimal. While the agent avoids "catastrophic" behavior, such as crashing into large obstacles or driving onto the sidewalk, it commits lesser driving errors, such as driving on the left side of the road. We can identify in what types of situations the modular approach fails by comparing the policies of a modular and a monolithic system. There are several possible explanations for why the modular approach fails in these situations. By definition, decomposing the global state-space into a set of smaller spaces, means that each module is operating in a state-space where one state can correspond to several global states. The modular approach achieves generalization this way, by assuming that the one local state can provide sufficient information to allow the right action to be executed in all the corresponding global states. Having one state correspond to a set of actual world-states is typically referred to as hidden state or perceptual aliasing, and can lead to poor performance when it is necessary to distinguish between the world-states in order to select the optimal action. In addition to the well known problems this perceptual aliasing causes, it also leads to apparently shorter paths to reward, causing utility estimates to be inflated.

Furthermore, the greatest mass approximation strategy can also produce misleading estimates of the utilities of states. Greatest mass estimates the global Q-values by using a simple linear combination of the local Q-values. However, approximating the global Q-value this way, violates an intrinsic assumption in the definition of the local Q-values: they represent the utility of an action, given that the local policy is followed afterwards. Since it is impossible that all the local policies will be followed concurrently, combining the local utilities can lead to incorrect estimates of the global utility.

We analyze the behavior of the modular approach below as follows:

- Failure states are identified. These are states where the modular policy differs from the monolithic one. Only a small set of states account for most of the difference in performance and a representative sample is shown.

- Three causes of failure are described:
  - Hidden state leads to failure when the modules are defined poorly leading to ambiguities between states with conflicting optimal actions.
  - In the modular state-spaces the paths to reward appear shorter than they actually are, leading to inflated Q-values
  - The greatest mass approximation strategy produces incorrect estimates by assuming that Q-values can be added.

7.1 Failure states

Figure 6.2 shows how the performance of a modular system reaches a performance level worse than that of the monolithic system. It is clear that the modular approach is sub-optimal, but the cumulative reward does not indicate how the modular and monolithic policies differ. By determining in what states the two policies recommend different actions, we can determine how the behaviors produced by the policies differ.

Comparing the policies learned after 30,000 steps, reveals that the modular approach produces mostly the same type of behavior as the monolithic approach. There are no "dramatic" failures where the modular policy drives the car into a large obstacle or onto the sidewalk. The most significant (and most frequent) error seems to be of driving on the left side of the road. Comparing the states in which the modular and monolithic policies differ produces a different set of states for each experimental run. There is thus no particular state, or set of states, that always cause the modular policy to behave differently than the monolithic one. However, Figure 7.1 shows a representative sample of states where the policies differ. These are states that appear in several experimental runs among the most frequently visited states where the policies differ. In these states (as in case of most of the other states where the policies differ), the agent clearly prefers driving on the left side of the road. This propensity to switch into the left lane unnecessarily as well as the reluctance to return to the right lane, accounts for most of the difference in cumulative reward between modular and monolithic policies.

In the experiments involving the sign behavior, most of the negative reward was accumulated in the states shown in Figure 7.2. When the agent is in the last row of an intersection with a sign pointing forward, facing the sidewalk, it tends to execute a `stop` action rather than simply using `jump-left` or `jump-right`, to follow the sign. This is due to the sign module not having enough information to differentiate between the states in Figure 7.2 and those in Figure 7.3. The sign module only knows how far away a the sign is, not whether the agent is approaching or in the middle of an intersection. The correct action in each case

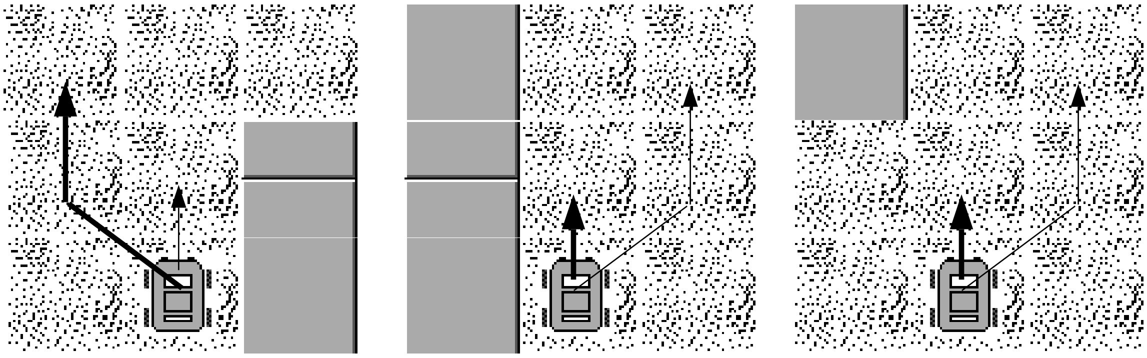


Figure 7.1: Some examples of the most frequent states in which the modular and monolithic policies differ. The thick and thin arrows represent the actions selected by the modular and monolithic policies respectively.

leads to a crash into the sidewalk in the other case. Adding more inputs to the sign module, allowing it to differentiate between these situations, should solve the problem.

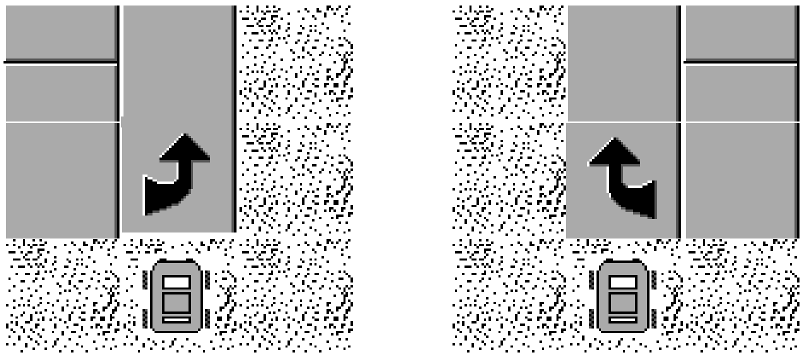


Figure 7.2: The states where most negative reward is accumulated in experiments where the sign behavior needs to be learned.

Though learned policies may differ over different experimental runs, in all of the cases analyzed, the differences between those policies learned using a modular approach and those using monolithic one have the same characteristics as described above: There are few or no situations where the modular policies lead to the car crashing into the sidewalk or a large object, and most errors occur in situations where there is only a relatively small negative reward incurred (such as driving on the left side of the road). Unlike previously expected therefore, the

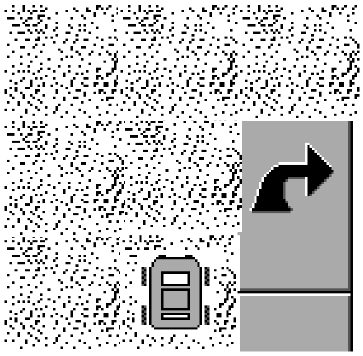


Figure 7.3: The modular agent's sign module cannot distinguish the above state from those where it is in the middle of the intersection. The correct action in one state likely leads to a crash in the other.

modular approach seems to be capable of learning the correct actions to execute in highly complicated situations with many strong reward sources (eg. large obstacles and street-lights). It is the cases where there are only relatively weak rewards available that mistakes occur.

7.2 Causes of failure

The modular approach depends on the decomposition of the task into largely independent sub-problems. The state-space is separated into modules and the reward function is designed to parcel out reward to only the relevant modules. Each module can thus learn its sub-problem quickly, ignoring the state of other sub-problems. However, since it is very difficult to design a perfect decomposition, there will be situations that cannot be handled properly by the modular approach. Among the causes of these failures are increased perceptual aliasing, shorter perceived paths to reward sources, and improper estimates of utility.

7.2.1 Hidden state

Since the state-space of each module is defined by only a subset of the available inputs, it can be considered an *abstraction* of the global state-space. Given a set of inputs s_i, $0 \leq i \leq n$, and a module that only uses the first k of those inputs, for example, any state S', in the module's state-space, corresponds to all the global states with the same value as S' for the first k inputs, but with all possible values for the remaining inputs. In a good decomposition, it is not necessary to distinguish between all of those states in order to learn the correct action to execute.

The sign module in SID is an example where hidden state leads to poor performance. As described in Section 7.1, the sign module does not have enough information to distinguish between two states where the optimal actions conflict with each other. The information from the road module is not sufficient to prevent the agent from making an error, leading to the agent crashing into the sidewalk. It is possible that this could be avoided by providing more inputs to the sign module. However, for most domains, including SID, it is impossible to design a decomposition where each module gets exactly the amount of information necessary at all times.

7.2.2 Shorter paths to reward

Another effect of decomposing the state-space into smaller spaces, is that the path between any two states appears to be shorter in the modular space, than in the monolithic one. Figure 7.4 illustrates such an example. In the monolithic space, the agent moves through 3 states before arriving at a state containing a streetlight. For the streetlight module however, it appears as if the agent has remained in the same state for 3 steps, before a new state is encountered. Because of the perceptual aliasing, the `move-forward` action appears to have a non-deterministic effect: for the most part, the outcome of the action is simply a return to the original state, but occasionally the state with a streetlight is the result. The effect on the Q-value of the `move-forward` action is to oscillate between a high value immediately after receiving the reward, and the low value reached immediately prior to receiving the reward again. After having reached the state with the light, and having received a reward, the Q-value is very high, reflecting the learned experience that reward is one step away from the state where there is no light to be seen. As the sequence of steps is repeated, the Q-value is decremented each time no reward is received, only to be increased again, as the sequence completes and reward is received once more. Thus, because it appears as if there is a short path to the reward state, the Q-value is initially too high, leading to inappropriate behavior. Then, as more steps are taken, the Q-value decreases, even though each step brings the agent closer to the reward state again, which should be indicated by an increase in the Q-values, as shown for the monolithic system. In experiments with SID, the result of this effect has been the preference for driving on th left side of the road. Executing a `jump-left` action, brings two new rows into view, as opposed to one row with the `move-forward` action, increasing the probability that a streetlight will appear. Once this happens, the Q-value of the `jump-left` action is greatly increased, leading it to dominate the agent's behavior.

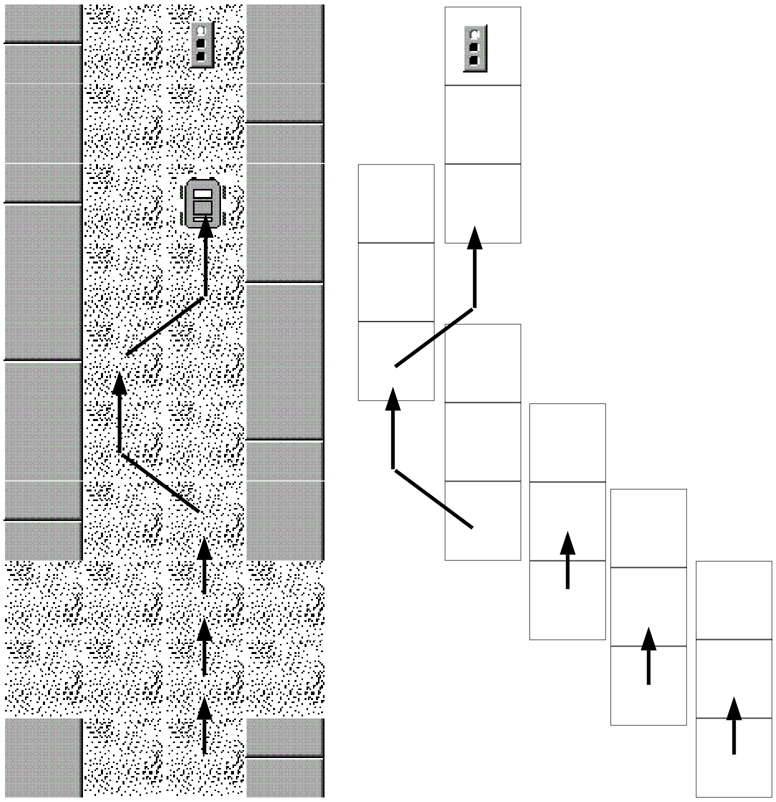


Figure 7.4: As the agent moves five steps forward in the world, it is perceived as five separate states by the monolithic system. In the modular system however, the streetlight module only detects two states. The appearance of the streetlight is seen as a non-deterministic, possible result of the move forward action.

7.2.3 Incorrect utility estimates

In addition to the problems with perceptual aliasing introduced by decomposing the state-space, aberrations from the optimal behavior are also introduced by incorrect utility estimates. Since the reward function is also decomposed into functions associated with each module, the modules compute utilities relative to their associated state-space and reward functions. These local utilities are combined into an estimate of the global utility using an approximation function. We have mainly investigated the *greatest-mass* function, which simply adds up the utilities of all the modules. Because of the decomposition of the reward function, and the approximation algorithm, incorrect estimates of the utilities may result.

Figure 7.5 shows an example of how the global utility can be incorrectly estimated, even if there is no perceptual aliasing.

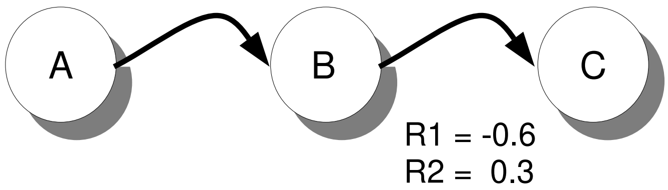


Figure 7.5: Even in the simple example shown, with two modules with identical perceptual inputs, the modular approximation produces incorrect results. The agent has 6 actions available to it, but executes the same one to move between states A, B, and C. When state C is reached, the indicated rewards are generated, which causes incorrect utility estimates in the Q-values for state A.

To simplify the example, we have defined both modules to have the same inputs as the monolithic system (ie. the monolithic system and both modules have the same state-space). Table 7.1 shows the Q-values for two different states, A and B, at different time steps, computed by the monolithic system, the two modules, and the estimate produced by the modular system. At time 0, action 0 was executed from state A, leading to state B, no reward was distributed. Since all Q-values at state B were set to their initial value of 0, the Q-values at state A do not change. At time 1, action 0 is executed from state B, leading to some state C, where all Q-values are also 0, but the two reward functions give rewards of -0.6 and 0.3, respectively. Using the standard update formula for Q-learning, with the learning rate α set to 0.8, and the discount factor γ set to 0.9, new Q-values are computed. In state B, the modular estimate of the global Q-value is completely correct. At some later time k, however, action 0 is again executed from state A, leading to state B, with no rewards. The Q-values from state B are propagated backwards, but now lead to an incorrect estimate in the modular system. The error results from the $U(y)$ term in the Q update formula. $U(y)$ is the utility of the resulting state, defined as the maximum Q-value at that state. In module 1, the negative Q-value does not affect the utility of the state as a whole, which remains at 0 — the Q-values of the remaining actions. The same is true in the monolithic system, where the Q-value of the action also remains at 0. In module 2 however, the utility of the new state is determined by the positive Q-value of action 0. This non-zero utility is propagated back to state A in module 2, and also determines the approximate Q-value defined by the greatest mass function. The result in this simple example is that the modular system assigns the highest utility to an action whose utility is actually identical to the other actions in that state.

state	time step	Q estimator	Q-values					
			0	1	2	3	4	5
A	0	monolithic	0.0	0.0	0.0	0.0	0.0	0.0
		module 1	0.0	0.0	0.0	0.0	0.0	0.0
		module 2	0.0	0.0	0.0	0.0	0.0	0.0
		Greatest Mass	0.0	0.0	0.0	0.0	0.0	0.0
B	1	monolithic	-0.2400	0.0	0.0	0.0	0.0	0.0
		module 1	-0.4800	0.0	0.0	0.0	0.0	0.0
		module 2	0.2400	0.0	0.0	0.0	0.0	0.0
		Greatest Mass	-0.2400	0.0	0.0	0.0	0.0	0.0
A	k	monolithic	0.0	0.0	0.0	0.0	0.0	0.0
		module 1	0.0	0.0	0.0	0.0	0.0	0.0
		module 2	0.1728	0.0	0.0	0.0	0.0	0.0
		Greatest Mass	0.1728	0.0	0.0	0.0	0.0	0.0

Table 7.1: Each row contains the Q-values estimated by listed method. There are 6 Q-values shown, one for each possible action available to the agent, but since in the example only the first action is executed, only its corresponding values change. The modular system correctly estimates the Q-values in state B, but fails in state A.

Preliminary experiments in SID indicate that incorrect utility estimates play a big role in the modular system's performance. In trials with only the road and streetlight modules, eliminating perceptual aliasing by giving both modules the same inputs, resulted in no significant difference in performance. Further experiments and analysis are needed to fully quantify the relative impacts of perceptual aliasing and incorrect utility estimates.

8 Conclusions

Standard, monolithic, Q-learning is impractical in any complex domain. Unless there is no limit on the computational resources needed to learn, as well as represent, a good policy in a large state space, the optimal solution guaranteed to be found by Q-learning is unattainable. This thesis demonstrates the value of the modular approach for such complex domains, using the Simple Driving domain (SID) as an instance, in several ways:

- The driving simulator (SID) allowed the design of many different types of experiments, illustrating the effects of various learning parameters, as well as different strategies. The speed of simulations enabled experiments to last for up to 30,000 steps, allowing performance limits to be clearly seen.
- Experiments show that the basic modular approach learns a policy with satisfactory performance, quickly and efficiently. As the monolithic system finally manages to reach its performance limit, it manages to accumulate more reward than the modular system. However, the difference in cumulative reward does not give a clear indication of actual differences in performance. One must count and classify the different types of events the agent encounters, relative to their relevance to sub-tasks. This counting of events shows that difference in behaviors are limited to a small set of states, and mistakes are limited to only some components of sub-tasks. The policy learned by the modular system at least partially satisfies all sub-goals, with most mistakes made in less important parts of the task. Varying the exploration rate shows that performance can be further improved to almost eliminate negative reward sources such as going through red and yellow lights or running into the sidewalk.
- Experiments also show that extending the modular approach with an activation function greatly improves the performance of the modular policy. The activation function can therefore be put to good use when further behaviors are added to the task. It also provides a natural control mechanism for a

higher level component (using planning methods) to specify what sub-tasks should be considered at the current time.

A crucial benefit of the modular approach is its initial good performance and learning speed. Though the experiments described in Chapter 6 track the agent's performance for 30,000 steps, in a real-world learning system, more than a few thousand (or even a few hundred) steps for learning might be considered excessive. A monolithic agent's initial performance suffers greatly from having to blindly search a very large state-space until some sources of rewards are discovered that can then be propagated through the state-space.

The problem with the monolithic approach is the lack of generalization. If even one bit of the input changes, the agent must assume it is in a completely different state than any previously encountered, and cannot extend any learned behavior to the new state. Therefore, the agent may unnecessarily have to relearn how to act in states, though previously not encountered, similar enough to states where the optimal action is already known.

The modular approach tries to achieve generalization by separating input and reward information into largely independent components. We assume that the modules can learn independently. It is clear that (except for contrived examples), the modular approach cannot learn a policy identical to that learned by the standard, monolithic, method. Limiting each module's inputs results in a much smaller state-space leading to perceptual aliasing and over-estimation of Q-values. Separating the reward functions and Q-tables amounts to an assumption that each module has complete control of the agent. However, despite these inherent flaws of the modular approach, it can still be useful to learn how to solve complex problems. It is not practical, nor necessary, to demand that an optimal solution be found, when the computational resources necessary are unavailable. In situations with little or no reward, it may not be important whether the strictly optimal action is selected. Similarly, we may be able to forgive a certain frequency of errors for certain type of situations. In the SID domain, the modular system did not make any major mistakes, ie. crashing the car into a sidewalk or an obstacle, or running a red light. The errors that were made involved minor events such as driving on the left lane of the road (forgivable in a one-car environment), and occasionally driving over small obstacles (eg pot-holes). The experiments indicate that important sub-goals, defined by high reward magnitudes, are achieved reliably, and some errors occur with low magnitude reward goals.

The experiments only show the applicability of the modular approach to the driving domain we defined. It is difficult to determine exactly what qualities of the domain makes it amenable to be learned by a modular agent. The grid world experiments show that if the task can be decomposed into a set of achievement goals, the modular system can learn a good nearest neighbor approximation of the optimal solution. When the goal structure is more complex however, as in

SID, it is not obvious what aspects of the learned policy will be sub-optimal. It is clear that to even attempt to apply the modular approach in a domain other than SID, the sub-goals must be clearly identifiable, with reward functions and inputs easily separated according to their relevance to each sub-goal. The other important aspect of the SID domain seems to be the definition of partially satisfiable sub-goals. Each sub-goal has several components, for example small and large obstacles in obstacle avoidance, some of which are more important than others. The experiments in the SID domain indicate that less important components of sub-goals will be sacrificed in favor of those components defined by large reward magnitudes. The agent designer must therefore be aware that small reward sources may be over shadowed by large ones, and must be prepared to accept solutions where such errors are made. Other than the rich reward structure however, there does not seem to be any other defining characteristics of the SID domain making it particularly amenable to the modular approach. It seems therefore, that other domains with similarly information rich reward functions, and where partial satisfaction of sub-goals is acceptable could also be learned using the modular approach.

Perhaps the most difficult part of learning multiple goals is designing a set of reward functions that work together correctly. Ideally, you would be able to encode in the reward function the degree to which any given goal must be satisfied. However, defining a reward function to represent rules like "the frequency of this error should be below 20%", can be difficult. Furthermore, it may be difficult to *a priori* identify all possible errors or alternative solutions, and thus hard to design a reward function that rewards them properly. However, the modular approach at least forces the agent designer to separate the reward according to sub-tasks, ensuring that rewards for accomplishing parts of the task remain consistent throughout the domain. This limits the flexibility of the reward function's design, in favor of simplicity. The loss of flexibility probably limits the modular approach less than it might appear, since a reward function taking advantage of the extra flexibility would be complex enough that unexpected interactions would lead to undesirable results.

The limited need for domain knowledge is typically touted as an advantage of RL. However, if we design the sensors and reward functions of the agent, a large amount of domain knowledge must typically be available to us. We know how to decompose the task to some extent, allowing us to create a good reward function. We know what sensory information is necessary for the tasks, and may even design sensors specific to certain sub-tasks (eg. "coke can detector"). Thus apart from the actual sensors and rewards themselves, we as agent designers often have a good idea of how to decompose the task, what inputs are relevant to what parts of the task, and maybe even under what circumstances a sub-task is "active" or not. The modular approach allows us to put this domain knowledge to good use. It would be very difficult to impart all the above domain knowledge to a standard, monolithic agent. It would probably necessitate a complex reward function, extra state-variables as inputs, and maybe even some clever initialization of the Q-values. With the modular approach however, we can simply create modules that naturally correspond to a given sub-task. Looking at each module it is clear how the domain knowledge has been applied to select the input sensors and create the reward functions. Thus, the modular approach allows us to easily take advantage of domain knowledge that is often available, while gaining a clear advantage in terms of learning time.

While the modular approach can be used to learn a good approximate solution in a short amount of time, the use of a hard-coded activation function shows that the learned policy can be improved. However, the experiments using a modular policy to initialize a monolithic system illustrate the importance of not achieving an improvement in policy at the expense of learning time. Ideally therefore, improvements should not necessitate learning in the global state-space, but rather rely only on the local information if possible.

One possibility is to try and learn the activation function. We defined each module's activation to depend only on the inputs available to that module, and are therefore not immediately forced to learn in the global state-space. A simple approach would be to simply add an "activation bit" to each module's input, and add another action, such as `toggle-activation`. However, it is not clear whether this would suffice, since the effect of changing the activation of a module is not an increase in the reward for that module, but rather an increase in the reward for other modules not being interfered with. Thus, a given module might not detect any utility in deactivating itself, and would learn to remain active all the time. It may be possible however to separate the learning of activations from the rest of the learning problem. A second reinforcement learning system, dedicated to activation, could be added to each module, using its perceptual inputs, but learning from the global reward function. Unfortunately, since the system learning activations would not see the global state information, the reward signal would appear to be very inconsistent. If on average however, there was a clear increase in utility in changing the activation of a module, no matter what the state of the other modules, the learning might still succeed.

Another approach would be to attempt to improve the extension that uses a modular policy to initialize a monolithic system. The extension failed because the modular approximation greatly over estimates the Q-values, causing the agent to have to relearn the policy in order to bring down the magnitude of the Q-values. It might be possible to avoid much of this relearning by scaling down the Q-values after initializing the monolithic system. The scaling mechanisms could range from a simple scaling by some guessed at (or determined through previous experimentation) factor, to some adaptive mechanism that spent some time after the initialization trying only to learn some appropriate scaling factor. In both cases care must also be taken to not *under-estimate* the Q-values, since that would also lead to poor performance. It may also be possible to use an internal simulation of the world, similar to that used by the DYNA system[Sutton, 1990], to arrive at Q-values of an appropriate scale.

The ideal extension to the modular approach would be an algorithm that could detect the global states in which the modular approximation failed, and then learned in the monolithic state-space for those states only. The modular policy would then be used in most situations, until a failure state was reached and would then revert to the monolithic system to learn the appropriate action. Unfortunately there is no clear criterion for discovering which states lead the modular approximation to fail. Immediate reward is in general not a good indicator of performance, since the approximation error may have been one in a state encountered several time steps earlier. However, the SID experiments indicate that most of the mistakes occur immediately preceding the receipt of negative reward (eg. changing to the left lane without cause). It may be worthwhile to explore the use of immediate negative reward as an indication of approximation failure therefore. It is possible that SID, and similar domains, have characteristics where this heuristic would perform well.

Though the modular approach provides a good way for the agent designer to incorporate domain knowledge about the structure of the task into the system, it would still be desirable to have a mechanism that could alter the decomposition, if not learn it from scratch. A possible avenue of exploration towards this goal is to use a memory-based approach, where the agent's history of states visited and rewards received is stored as the agent attempts to learn its task. Given a history of states and associated actions and rewards, different decompositions could be designed, and then tested on the previously gathered data. Some of the criteria for preferring one decomposition over another would then be how evenly it decomposes the state-space, how much hidden state is introduced in each module, and some measure of the expected utility of the policy. There also needs to be some good way to generate candidate decompositions, or one is left with the combinatorial problem of attempting all possible combinations of inputs.

Bibliography

[Agre, 1988] Agre, Philip E. 1988. *The Dynamic Structure of Everyday Life.* Ph.D. Dissertation, MIT Artificial Intelligence Lab. (Tech Report No. 1085).

[Arbib and House, 1987] Arbib, Michael A. and House, Donald H. 1987. Depth and detours:an essay on visually guided behavior. In Arbib, M. and Hanson, A., editors 1987, *Vision, Brain, and Cooperative Computation.* MIT Press, Cambridge, MA.

[Arkin, 1990] Arkin, Ronald C. 1990. Integrating behavioral, perceptual, and world knowledge in reactive navigation. *Robotics and Autonomous Systems* 6:105–122.

[Barto *et al.*, 1983] Barto, Andrew G.; Sutton, Richard S.; and Anderson, Charles W. 1983. Neuron-like elements that can solve difficult learning control problems. *IEEE Trans. on Systems, Man, and Cybernetics* SMC-13(5):834–846.

[Brooks and Connell, 1986] Brooks, Rodney A. and Connell, Jonathan H. 1986. Asynchronous distributed control system for a mobile robot. *SPIE* 727:77–84.

[Brooks, 1985] Brooks, Rodney A. 1985. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation* RA-2(1).

[Dayan, 1992] Dayan, Peter 1992. The convergenece of TD(λ) for general λ. *Machine Learning* 8.

[Feldman, 1962] Feldman, D. 1962. Contributions to the two-armed bandit problem. *Annals of Math. Statist.* 33(2):847–856.

[Foulser *et al.*, 1990] Foulser, David E.; Li, Ming; and Yang, Qiang 1990. Theory and algorithms for plan merging. Technical report, University of Waterloo.

[Garey and Johnson, 1979] Garey, Michael R. and Johnson, David S. 1979. *Computers and Intractability.* W.H¿ Freeman and Co.

[Haddawy and Hanks, 1990] Haddawy, Peter and Hanks, Steve 1990. Issues in decision-theoretic planning: Symbolic goals and numeric utilities. In *DARPA workshop on Innovative approaches to planning, scheduling, and control.* Morgan Kaufmann.

[Humphrys, 1996] Humphrys, Mark 1996. Action selection methods using reinforcement learning. In *"Proceedings of the Fourth International Conference on the Simulation of Adaptive Behavior",*.

[Jacobs and Jordan, 1991] Jacobs, Robert A. and Jordan, Michael I. 1991. A competitive modular connectionist architecture. In Lippmann, R. P.; Moody, J.; and Touretzky, D. S., editors 1991, *Advances in Neural information processing systems 3.* Morgan Kaufmann publishers, Inc.

[Johnson, 1990] Johnson, David S. 1990. Local optimization and the traveling salesman problem. In Paterson, M. S., editor 1990, *Lecture notes in computer science: Automata, languages and programming.* Springer-Verlag.

[Jordan and Jacobs, 1993] Jordan, Michael I. and Jacobs, Robert A. 1993. Hierarchical mixtures of experts and the em algorithm. Technical Report 9301, MIT Computational Cognitive Science.

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

[Khatib, 1986] Khatib, O. 1986. Real-time obstacle avoidance for manipulators and mobile robots. *International Journal of Robotics Research* 5(1).

[Lin, 1993a] Lin, Long-Ji 1993a. *Reinforcement learning for Robots using neural networks.* Ph.D. Dissertation, Carneige Mellon.

[Lin, 1993b] Lin, Long-Ji 1993b. *Reinforcement Learning for Robots Using Neural Networks.* Ph.D. Dissertation, Carnegie Mellon, School of Computer Science.

[Maes and Brooks, 1990a] Maes, Pattie and Brooks, Rodney A. 1990a. Learning to coordinate behaviors. In *Proceedings of AAAI 90.*

[Maes and Brooks, 1990b] Maes, Pattie and Brooks, Rodney A. 1990b. Learning to coordinate behaviors. In *Proceedings of AAAI-90.* 796–802.

[Maes, 1989] Maes, Pattie 1989. The dynamics of action selection. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence.*

[Maes, 1992] Maes, Pattie 1992. Learning behavior networks form experience. In Varelda, Francisco J. and Bourgin, Paul, editors 1992, *Toward a Practise of Autonomous Systems: Proceedings of the First European conference on Artificial Life*. MIT Press.

[Mahadevan and Connell, 1990] Mahadevan, Sridhar and Connell, Jonathan 1990. Automatic programming of behavior-based robots using reinforcement learning. Research Report RC 16359, IBM T.J. Watson Research Center.

[Mataric, 1994] Mataric, Maja 1994. Reward functions for accelerated learning. In *The Proceedings of the Eleventh International Machine Learning Conference*. Morgan Kaufmann Publishers, Inc.

[McCallum, 1992] McCallum, R. Andrew 1992. First results with utile distinction memory for reinforcement learning. Technical Report 446, University of Rochester Computer Science Dept.

[McCallum, 1996] McCallum, Andrew Kachites 1996. *Reinforcement Learning with Selective Perception and Hidden State*. Ph.D. Dissertation, University of Rochester.

[Minsky, 1954] Minsky, Marvin L. 1954. *Theory of Neural-Analog Reinforcement Systems and Its Application to The Brain-Model Problem*. Ph.D. Dissertation, Princeton University.

[Peng and Williams, 1992] Peng, Jing and Williams, Ronald J. 1992. Efficient learning and planning within the dyna framework. In *From Animals to Animats:Proceedings of the Second International Conference on Simulation of Adaptive Behavior*. MIT Press.

[Sacerdoti, 1974] Sacerdoti, E. D. 1974. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence* 5(2):115–135.

[Sacerdoti, 1975] Sacerdoti, E. D. 1975. The non-linear nature of plans. In *IJCAI-4*. 206–214.

[Samuel, 1963] Samuel, A. L. 1963. Some studies in machine learning using the game of checkers. In Feigenbaum, E. and Feldman, J., editors 1963, *Computers and Thought*. Krieger, Malabar, FL. 71–105.

[Seidenfeld, 1985] Seidenfeld, Teddy 1985. Calibration, coherence, and scoring rules. *Philosophy of Science* (52):274–294.

[Singh, 1992] Singh, Satinder 1992. Transfer of learning by composing solutions of elemental sequential tasks. *Machine Learning* 8:323–339.

[Spector and Hendler, 1990] Spector, Lee and Hendler, James 1990. Knowledge strata: Reactive planning with a multi-level architecture. Technical Report 2564, Institute for Advanced Computer Studies, Dept of Computer Science and Systems Research Center, University of Maryland.

[Sutton, 1988] Sutton, Richard S. 1988. Learning to predict by the method of temporal differences. *Machine Learning* 3(1):9–44.

[Sutton, 1990] Sutton, Richard S. 1990. Integrated architectures for learning, planning, and reactingbased on approximating dynamic programming. In *Proceedings of the Seventh International Conference on Machine Learning*. Morgan Kaufman Publishers.

[Tenenberg *et al.*, 1992] Tenenberg, Josh; Karlsson, Jonas; and Whitehead, Steven 1992. Learning via task decomposition. In *Proceedings of the Second International Conference on Simulation of Adaptive Behavior: From Animals to Animats*. Morgan Kaufmann Publishers, Inc.

[Tenenberg, 1988] Tenenberg, Josh D. 1988. *Abstraction in planning*. Ph.D. Dissertation, University of Rochester.

[Tesauro, 1992] Tesauro, G. 1992. Practical issues in temporal difference learning. *Machine Learning* 8:257–277.

[Thrun, 1992] Thrun, Sebastian 1992. Efficient exploration in reinforcement learning. Technical Report CMU-CS-92-102, School of Computer Science, Carnegie Mellon University.

[Watkins and Dayan, 1992] Watkins, Christopher and Dayan, Peter 1992. Q-learning. *Machine Learning* 8.

[Watkins, 1989] Watkins, Chris 1989. *Learning from delayed rewards*. Ph.D. Dissertation, Cambridge University.

[Whitehead and Ballard, 1989] Whitehead, Steven D. and Ballard, Dana H. 1989. A role for anticipation in reactive systems that learn. In *Proceedings of the Sixth International Workshop on Machine Learning*, Ithaca, NY. Morgan Kaufmann.

[Whitehead *et al.*, 1992] Whitehead, Steven D.; Karlsson, Jonas; and Tenenberg, Josh 1992. Learning multiple goal behavior via task decomposition and dynamic policy merging. In *Robot Learning*. MIT Press, Cambridge, MA.

[Whitehead, 1991] Whitehead, Steven D. 1991. *Reinforcement Learning for the Adaptive Control of Perception and Action*. Ph.D. Dissertation, Department of Computer Science, University of Rochester, Rochester, NY.

[Wixson, 1991] Wixson, Lambert E. 1991. Scaling reinforcement learning techniques via modularity. In Birnbaum, Lawrence E. and Collins, Gregg C., editors 1991, *Machine Learning:Proceedings of the eighth International workshop.* Morgan Kaufman.