THE GENERALIZED TUTOR-STUDENT LEARNING ALGORITHM FOR AUTONOMOUS MOBILE ROBOTS

by

Kevin Brammer

BSc., University of Northern British Columbia, 2004

THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

MATHEMATICAL, COMPUTER AND PHYSICAL SCIENCES

(COMPUTER SCIENCE)

THE UNIVERSITY OF NORTHERN BRITISH COLUMBIA

October 2005

Your file Votre référence
ISBN: 978-0-494-28385-1
Our file Notre référence
ISBN: 978-0-494-28385-1

Abstract

This thesis introduces the generalized Tutor-Student learning algorithm, which is designed to extend an existing robot controller in order to improve its performance and adaptability. The prototype 2 controller was developed for the tasks of obstacle avoidance and attractor path planning with predator avoidance, in partially observable static and dynamic environments. A version of this learning algorithm has been implemented to extend the prototype 2 robot controller, a behavior based robot controller. This extended controller is called the prototype 2A robot controller. The performance of this learning algorithm was evaluated by deploying the prototype 2 robot controller and the prototype 2A robot controller in the same environments and comparing their results. It has been shown that the prototype 2A robot controller has superior performance over the prototype 2 robot controller due to its usage of a Tutor-Student learning algorithm. This thesis will provide a complete description of the generalized Tutor-Student learning algorithm and an application of this algorithm.

TABLE OF CONTENTS

List of Tables

List of Figures

Chapter 1

Introduction

This thesis introduces the generalized Tutor-Student learning algorithm for mobile robot controllers. The learning algorithm is used to extend an existing robot controller in order to give the controller learning capabilities. This chapter will explain some of the problems with conventional robot controllers, as well as explain how the Tutor-Student learning algorithm can help overcome some of these problems.

1.1 Problem Statement

A *robot controller* is, in essence, the brain of the robot. The job of the robot controller is to map the robot's perceptions to actions. In this paper the terms agent and robot will be used interchangeably. The term *agent* applies to any entity that is self contained, independent and can interact with its environment. A mobile robot is an independent agent that has the ability to make decisions based upon a given situation without the influence or interference of a human operator.

Since the 1960s, research has been conducted in furthering the development of robot controllers. Three primary robot paradigms have been developed. The *deliberative* robot controller paradigm was developed in the 1960s. These types of controllers follow a sense $\longrightarrow$ plan $\longrightarrow$ act control cycle. The *reactive* robot controller paradigm was developed in the mid 1980s. Reactive controllers follow a sense $\longrightarrow$ act control cycle. The *hybrid* robot controllers are a combination of the deliberative controllers and the reactive controllers. In all of the above paradigms, actions are produced by the robot controller through either reaction or planning or both. As a consequence learning does not happen in any of the paradigms. These robot controllers are incapable of learning from past experiences, which results in the performance of the robot controller being solely dictated by the robot designer.

As a result the field of *robot learning* was founded. Robot learning is a branch of *machine learning*, which is concerned with developing techniques that allow machines to learn. The field of robot learning has focused on two areas of learning in robot controllers:

Competence Acquisition: The robot controller tries to learn a new competence. For example, a robot controller may not know how to make the robot walk, but it is able to learn how through negative and positive feedback.

Competence Adaptation: The robot controller tries to positively modify a competence through interaction with its environment. For example, if the robot controller knows how to fetch frisbees, can the same robot controller learn to gather more frisbees faster?

This thesis looks at the problem of competence adaptation. We ask the question, how can we convert an existing robot controller into a learning robot controller?

1.2 Our Contribution

In answering the question of how to convert an existing robot controller into a learning robot controller, we developed the generalized Tutor-Student learning algorithm. The generalized learning algorithm is a method that specifies how to add adaptability to the behaviors of a pre-existing robot controller. As well we have constructed a possible implementation of the learning algorithm and have evaluated its performance in four different simulated environments.

All four of the simulated environments test a robot controller's ability to avoid obstacles and to determine paths to attractors while avoiding predators. A robot controller's performance in all of these environments is judged by the number of times the robot controller was able to successfully complete its tasks (i.e. how many times it avoided obstacles, as well as how many times it found attractors). The experiments that we have conducted in these environments have shown that a robot controller using the Tutor-Student learning algorithm can achieve between almost two to over five times better performance (depending on the environment) in the task of attractor path planning with predator avoidance, than the same controller not using this learning algorithm. Therefore, we can claim that the Tutor-Student learning algorithm is successful since it allows a robot controller to automatically tune its actions according to its environment.

1.3 Overview

There are four remaining chapters in this thesis. Chapter 2 is the literature survey, which will focus on conventional robot controllers and robot learning with several case examples. Chapter 3 explains the principles of the Tutor-Student learning algorithm and offers some guidelines on its implementation. Chapter 4 describes the experimental procedures and results. Finally, Chapter 5 will present our conclusions and open questions regarding future research with the Tutor-Student learning algorithm.

Chapter 2

Review of Control Architectures

This literature survey will discuss the evolution of robot controllers and how robot learning integrates with robot controllers. In this literature survey, we will examine the three primary robot controller paradigms. As well, we will briefly look at the field of robot learning along with three learning robot controller case studies.

2.1 Robot Controllers

A robot controller maps how perceptions from sensors are translated into actions to be executed by actuators. Robot controllers can be implemented as either software or hardware, or as a combination of both. The study of robot controllers is quite diverse, consequently this paper will only be covering three of the primary paradigms in robot control architectures: deliberative, reactive and hybrid controllers.

2.1.1 Autonomous Agents

The term agent is an important concept in robotics. The definition of an autonomous agent can be defined iteratively. Maes does this [Mae94] as follows:

> "An *agent* is a system that tries to fulfill a set of goals in a complex, dynamic environment."
>
> "An agent is called *autonomous* if it operates completely autonomously, i.e. if it decides itself how to relate its sensor data to motor commands in such a way that its goals are attended to successfully."

The term agent in this paper is used to describe any entity with the capacity to act intelligently and purposefully in its environment in order to achieve its goals, specifically the term agent will refer to robotic agents.

2.1.2 Deliberative Controller Paradigm

Deliberative controllers were first developed in the 60s, and dominated the the field of robot controllers until about the mid 80s. In deliberative controllers the flow of control is unidirectional in the general case [Gat98]. Deliberative controllers decompose an agent in one decomposition. The decomposition in deliberative controllers is commonly called a *horizontal* decomposition due to the unidirectional flow of information through its modules.

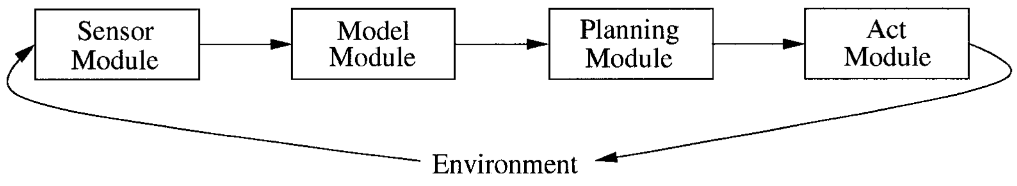

Fig. 1: Deliberative Controller Model 1

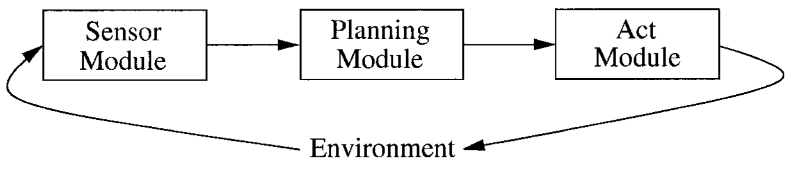

Fig. 2: Deliberative Controller Model 2

Figure 1 and Figure 2 present two types of deliberative controllers. Deliberative controllers usually are decomposed into the following modules:

Sensor Module: This module's job is to globally collect all the sensor signals from the agent's sensors. These signals are usually, but not necessarily, preprocessed into a knowledge representation (e.g., symbolic representation or fuzzy logic representation). This information is then passed onto either the model module or the planning module.

Model Module: This module translates the inputs from the sensor module into a representation of the agent's environment. This model representation of the environment, however, is only as accurate as the sensors sensing the actual environment [Bro91]. This information is then made accessible to the planning module.

Planning Module: This module is also called the planner. Its job is to formulate plans based on the agent's internal world model and current goal. Once a plan has been formulated, the plan is then given to the act module to execute. It should be noted that the model module and the planning module are sometimes combined into one module and referred to as the planning module.

Act Module: This module is also called the executive. The job of this module is to execute the plan given to it by the planning module. This plan is executed as a computer program is executed [Gat98], by carrying out a sequence of instructions or actions.

From the description of the above modules the deliberative controllers can be seen as modified general problem solvers in artificial intelligence [Bro91]. The limitations of deliberative controllers are discussed in section 2.1.5.1.

2.1.3 Reactive Controller Paradigm

Reactive controllers are characterized by their tight coupling of perception and action [Ark98, Bro86, Bro89]. In reactive controllers, the flow of control is explicitly or implicitly parallel [Bro86, NHS89], which can be seen in Figure 3. The parallel nature of a reactive controller is done for modularity, multiple goals and robustness [Ark98, Bro86]. The reactive controller paradigm decomposes an agent in several decompositions. Reactive controllers typically follow an incremental design and test philosophy [Ark98, Bro86]. An incrementally designed agent is designed and tested on a small set of behaviors. Once the agent has been tested successfully, more behaviors can then be added.

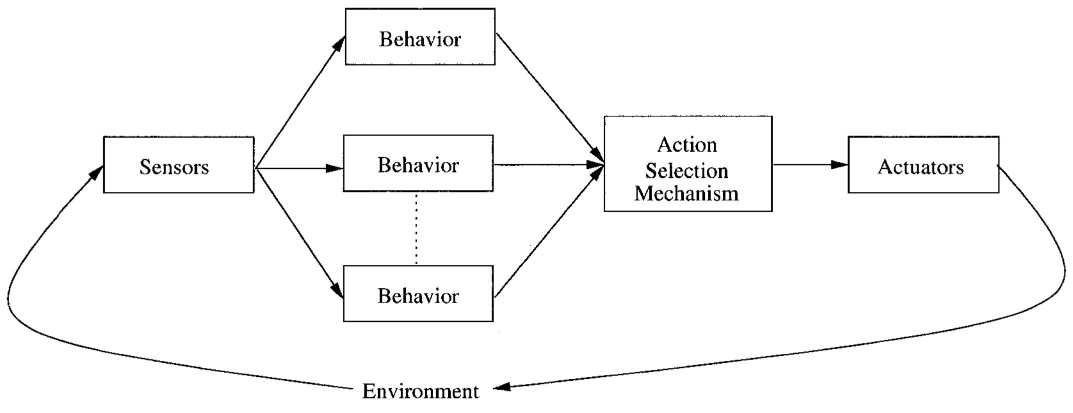

Fig. 3: Generalized Reactive Architecture

A reactive controller is usually decomposed into a set of behaviors and an action selection mechanism [Ark98, Bro86]. The job of a behavior module is to map perceptions into an action. This mapping is typically done so that there is very little or no planning involved in generating the action. Each behavior module runs concurrently with other behavior modules. A behavior module can be decomposed in the same manner as a deliberative controller [Bro86]; however, planning is typically kept to a minimum as emphasis is put on the speed of the behavior module. Behavior modules can receive perceptual information in several ways [Ark98, Bro86, Bro89]. According to Arkin there are three ways in which perceptual information can be received by a behavior module [Ark98]:

Sensor Fission: Sensor fission occurs when a behavior specific stimulus is used to produce a response, and thus a dedicated perceptual module (sensor) is used to channel its output directly into the behavior module [Ark98].

Action-Oriented Sensor Fusion: Action-oriented sensor fusion occurs when transitory representations local to a behavior are constructed from multiple perceptual modules (sensors) [Ark98]. These transitory representations restrict the sensor input to the requirements of a behavior modules [Ark98].

Perceptual Sequencing: Perceptual sequencing occurs when the behavior module allows for the coordination of multiple perceptual algorithms over time in support of a single behavioral activity [Ark98]. These perceptual algorithms are sequenced in and out, based on the agent's needs and the environmental context in which the agent is situated [Ark98].

The job of the action selection mechanism is to determine which action to select from the set of actions generated by the set of behavior modules [Bro86, Ark98, NHS89]. Once an action has been selected by the action selection mechanism, the action is then executed. Action selection mechanisms can be divided into two branches: arbitration and command fusion [Pir99]. Arbitration is a competitive approach where behaviors compete for control of the robot's actuators [Pir99], whereas, command fusion is where each behavior contributes to the final action that is selected to be executed by the robot's actuators [Pir99].

The most notable and popular reactive architecture is Brook's subsumption architecture [Bro86]. It was developed in response to the difficulties and limitations (see page 12) associated with the deliberative controller paradigm [Bro86]. The limitations of reactive controllers are discussed in section 2.1.5.2.

2.1.4 The Hybrid Controller Paradigm

Hybrid controllers attempt to capture the reactivity of reactive controllers and the deliberative planning of deliberative controllers [Ark98]. Unlike reactive and deliberative controllers, there is no clear breakdown of modules used in hybrid controllers [Ark98]. Hybrid controllers are usually described in terms of layers, where each layer is really a controller in its own right. At the very least, an agent implementing the hybrid controller paradigm requires a deliberative layer and a reactive layer [Ark98]. In fact these are called two level architectures [SRK93a]. In two level architectures, a deliberative planner is used to generate guidelines for the reactive controller [SRK93a]. The two level architectures follow Agre and Chapman's [AC90] philosophy of using plans as being a resource for the agent as opposed to explicit execution. Some notable two level architectures are the Theo-agent architecture[Mit90], the DAMN architecture [Pir99] and the fuzzy controller architecture in [SRK93b, SRK93a].

The other popular class of hybrid architectures, that have been relatively well defined are the three layer architectures. Figure 4, shows a diagram representation of this

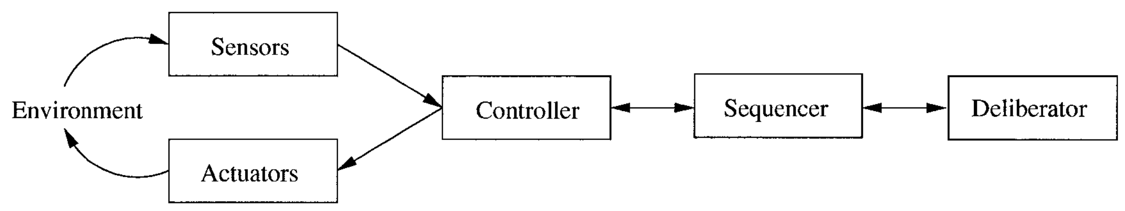

Fig. 4: Three Layer Architecture

architecture, which consists of the following three layers:

- A reactive feedback control mechanism called the *controller*. Gat [Gat98] describes the controller as follows:

  > "The controller consists of one or more threads of computation that implement one or more feedback loops, tightly coupling sensors to actuators. The transfer function(s) computed by the controller can be changed at run time. Usually the controller contains a library of hand-crafted transfer functions (called primitive behaviors or skills). Which ones are active at any given time is determined by an external input to the controller."

  The controller is in essence a reactive controller that implements a set of behaviors.

- A slow deliberate planner, which is called the *deliberator*. Gat [Gat98] describes it as follows:

  > "The deliberator is the locus of time-consuming computations. Usually this means such things as planning and other exponential search-based algorithms, but as noted before, it could include polynomial-time algorithms with large constants such as certain vision processing algorithms

> in the face of limited computational resources. The key architectural feature of the deliberator is that several Behavior transitions can occur between the time a deliberative algorithm is invoked and the time it produces a result. The deliberator runs as one or more separate threads of control. There are no architectural constraints on algorithms in the deliberator, which are invariably written using standard programming languages."

The deliberator is basically a deliberative controller.

- A sequencing mechanism called the *sequencer* is used to connect the controller and the deliberator. Gat [Gat98] describes it as follows:

  > "The sequencer's job is to select which primitive Behavior (i.e. which transfer function) the controller should use at a given time, and to supply parameters for the Behavior. By changing primitive Behaviors at strategic moments the robot can be coaxed into performing useful tasks. The problem, of course, is that the outcome of selecting a particular primitive in a particular situation might not be the intended one, and so a simple linear sequence of primitives is unreliable. The sequencer must be able to respond conditionally to the current situation, whatever it might be."

  The sequencer is more of an action selection mechanism than a robot controller. Its primary purpose is to interface the controller and the deliberator.

Some notable three layer architectures are Connell's SSS architecture [Con92], Gat's Atlantis architecture [Gat98] and 3T architecture [BK96]. The limitations of hybrid controllers are discussed in section 2.1.5.3.

2.1.5 Discussions of Different Robot Controller Paradigms

This section will discuss the advantages and disadvantages associated with the three primary robot controller paradigms.

2.1.5.1 Deliberative Controllers

The major advantage of deliberative controllers over reactive controller (Section 2.1.3 on page 7) is their ability to globally plan a course of actions for the agent to perform. The reactive controller's designer typically focuses on carefully designing the behaviors of the agent in order to achieve a task. Planning in deliberative controllers provides a higher level of control and task execution as compared to the reactive controllers.

While planning is a major advantage to the deliberative controllers, it is also the source of most of the deliberative controller's disadvantages. Planners in deliberative controllers rely heavily on an accurate world model to formulate plans. However, it is impossible to take any sensor data and create an accurate representation of the world [Bro91]. All sensors have margins of error. As well, the environment may not allow sensors to perform optimally. Not to mention, actuators are not accurate enough to precisely carry out actions dictated by the planning module. This kind of error in the actuators can cause an invalidation of the plan and the need for re-planning. Planners in deliberative controllers prevent the agent from being a part of its environment. The executive will execute a plan regardless of whether it is still relevant. Some executives are programmed to stop executing the plan if the plan can not succeed. Then control is given back to the planner so that it can formulate a new plan [AC90]. In dynamic environments, the formulation of a plan is time consuming and costly of the robot's resources. If the agent has to rethink plans continually due to a changing environment, then the agent will be slow to react to situations that call for immediate action. Hybrid controllers (Section 2.1.4 on page 9) attempt to alleviate this problem by combining the deliberative controller paradigm and the reactive controller paradigm together. Agre and Chapman [AC90], alternatively, propose that a plan be used more as a resource than as explicit instructions. This appears to be one of the primary ideas in hybrid controller design. Another disadvantage of deliberative controllers is that they are not robust. A single failure in any module will cause either an error in all proceeding modules, or the complete system failure of the agent.

2.1.5.2 Reactive Controllers

The reactive controllers have several advantages. They are able to react almost immediately to situations. However, immediate reaction may be a disadvantage, in that the robot may blindly act and miss opportunities presented to it by its environment. Reactive controllers have the ability to contend with multiple (possibly conflicting) goals due to the parallel nature of their behaviors and action selection mechanisms. Reactive controllers are robust in that a single failure will not cause the whole system to fail. This robustness is achieved through the parallelization of behaviors in reactive controllers.

The biggest difficulty in reactive controllers is their blind reaction to different situations. The robot's reaction to stimuli is mapped out by the robot designer who assumes what will be important stimuli for the robot and how the robot will react to that stimuli [Neh92]. The problem is that the way humans perceive things is not necessarily the same way a robot should perceive things [Neh92]. A solution to the problem of robot perception is to have the robot learn a behavior and determine the stimuli that are important for that behavior (see section 2.2.2 on page 21). Since explicit planning is discouraged in reactive controllers, implicit planning usually takes place through the controller's action selection mechanism. The vast majority of these implicit plans are hand designed by the designer of the robot, according to the designer's perceptions as opposed to the robot's. Another disadvantage in reactive controllers is that they do not scale up well to complex tasks [Gat98]. As well reactive controllers typically are unable to adapt from one environment to another, as a result of the specificity of the behaviors in reactive controllers for particular environmental cues.

2.1.5.3 Hybrid Controllers

The major advantage of hybrid controllers is that they provide the best features of both deliberative controllers and reactive controllers. They are able to react when necessary and to formulate plans when they have to.

The major difficulty with hybrid systems is that there is no definitive way of interfacing a deliberative planner with a reactive controller. Under the two level architectures, there are a variety of ways of interfacing the deliberative planner with the reactive controller, each of which has its own advantages and disadvantages. The three layer architectures are a bit more defined as to what layers are needed. However, there is not a lot of literature on the sequencer layer and more specifically how it interfaces with the deliberator and the controller.

2.2 Learning in Robot Controllers

The field of robot learning tries to tackle the question of how to make a robot acquire knowledge in order to perform certain tasks within an environment successfully [NSM93] and how to adapt existing behaviors in order to gain improved task performance from the robot [Ark98]. In this section, we present learning algorithms as well as case examples of learning in robot controllers.

2.2.1 Learning Algorithms

This section will present a selection of learning algorithms from cybernetics, neural networks and reinforcement learning. All of the presented learning algorithms learn incrementally through trial and error.

2.2.1.1 Cybernetics

Robotic controllers are designed to map perception to action. The first field to study this mapping from perceptions to action is cybernetics, which can be described in the following manner:

> "Cybernetics is the application of control theory to complex systems." [Neh99].
>
> "A marriage of control theory, information science, and biology that seeks to explain the common principles of control and communication in both animals and machines." [Ark98]

Both of the above descriptions include the field of control theory, which is the mathematical study of how to manipulate the parameters affecting the behavior of a system in order to produce an optimal outcome.

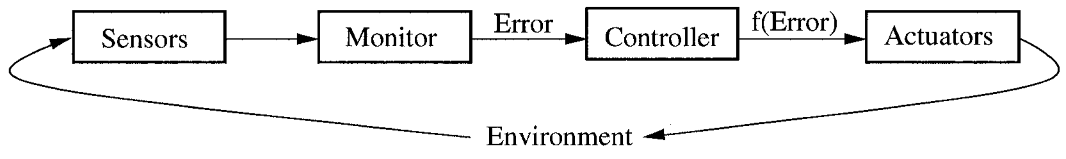

Fig. 5: Cybernetic Controller Model

A cybernetic control model is shown in Figure 5. It uses a function called the *monitor* to compare the actual status of a system x_t at time t with a desired system status τ_t [Neh99]. The monitor returns the error between the actual system state and the desired system state which is defined as $\epsilon_t = x_t$ - τ_t [Neh99]. The error ϵ_t is then used as an input signal to a *controller* module [Neh99]. The job of the controller is to minimize the error between the actual system state and the desired system state [Neh99]. Therefore, the controller will choose an action y_{t+k} such that $y_{t+k} = f(\epsilon_t) = f(x_t - \tau_t)$ at the next time step *t+k* [Neh99]. Thus, the next action selected by the controller is based on a function of the error ϵ_t. The function f in cybernetics is defined as the *control function* [Neh99], where learning takes place. The main focus in cybernetics is to define the control function f and minimize its parameter ϵ_t [Neh99].

2.2.1.2 Neural Networks

The field of neural networks attempts to emulate the neural networks found in biological creatures. Neural network learning fall into two categories: supervised and non-supervised learning. In supervised learning, we already know the inputs and outputs that are used to train the neural network. In non-supervised learning, the neural network attempts to cluster inputs into categories.

The neural network learning algorithm that we will examine is a supervised learning algorithm called the *incremental backpropagation learning algorithm.* This learning algorithm works on a neural network with a multilayer feed forward network topology, whose hidden layer and output layer employ sigmoid transfer functions [Mit97]. The learning algorithm employed by a backpropagation neural network allows it to learn non linearly separable functions [Fau94]. The algorithm for the incremental backpropagation learning algorithm is given in Appendix C (page 80).

2.2.1.3 Reinforcement Learning

Reinforcement learning occurs when an agent learns through trial and error interactions with its environment [SB98]. In order to adequately describe how reinforcement learning is connected with competence acquisition and competence adaption, four main elements to reinforcement learning need to be defined. A description of these four elements is given below:

Policy:

> "A *policy* defines the learning agent's way of behaving at a given time. Roughly speaking, a policy is a mapping from perceived states of the environment to actions to be taken when in those states." [SB98]

A policy is a way for choosing an action based upon numerical values. A typical policy used in reinforcement learning is if an agent is in state i, all the values of actions that can be performed in state i are recalled and the action with the maximum value is performed [SB98]. This process is known as exploitation in that the agent is exploiting previous knowledge [SB98]. Exploration occurs when a random action in state i is chosen. The policy of an agent usually chooses between actions that exploit the agent's knowledge or expand the agent's knowledge [SB98]. This is done so that the agent can have a balance between exploiting already know good actions and exploring actions that have the potential of being good.

Reward Function:

> "A *reward function* defines the goal in a reinforcement learning problem. Roughly speaking, it maps each perceived state (or state-action pair) of the environment to a single number, a *reward*, indicating the intrinsic desirability of that state. A reinforcement learning agent's sole objective is to maximize the total reward it receives in the long run. The reward function defines what are the good and bad events for the agent." [SB98]

The reward function gives an immediate numerical reward to the agent for transitioning from one state to another state [Gos03]. The immediate reward received by an agent can be positive or negative. If the reward is negative, it is typically referred to as a cost. In dynamic programming (which is a precursor to reinforcement learning), each state is given a immediate reward [Gos03]. However, in reinforcement learning only a subset of all the states in the agent's world are given immediate rewards, whereas the rest of the states are assumed to have an immediate reward of zero [Gos03]. Thus, only states that are considered to be significant are given immediate rewards.

Value Function:

> "... a *value function* specifies what is good in the long run. Roughly speaking, the *value* of a state is the total amount of reward an agent can expect to accumulate over the future, starting from that state." [SB98]

The value function is used to compute the value of a state or state-action pair. It is a way of looking ahead and judging how valuable it is for an agent to perform an action or enter a certain state. Typically, a value function is defined in a tabular format that maps state-action pairs to numerical values. These numerical values provide the basis for selecting which action to perform. Usually a value function can also be defined in a neural network with an incremental style updating rule [Gos03]. An incrementally trained neural network is one that is updated in increments rather than in batches. A neural network has the property that it can generalize output values across different input values, even input values that it has not seen before. By incrementally teaching the neural network, it becomes a function approximator.

Model:

> "This is something that mimics the behavior of the environment." [SB98]

A model is a simulation, which allows the agent to learn quicker than if it had to learn in the actual environment[Gos03]. A simulation allows the agent to experience more states in less time than in its actual environment. In essence the model is a training world for the agent.

The reinforcement learning algorithm that we will examine in this section is the *Q-learning algorithm.* Q-learning has a learning component called the value function Q(state, action) and a decision component called the policy. The learning in Q-learning works by shaping a value function as the agent interacts with its environment [SB98]. Decision making in Q-learning is accomplished through the policy [SB98], which is a function that chooses an action based on the results of the value function for all combinations of the current state i and the set of possible actions that can be executed in state i [SB98]. If the chosen action causes the robot to be in an undesirable state, then that action's expected reward for the previous state is deceased. Otherwise, the action's expected reward for the previous state is increased. By repeating this process, the agent's policy will eventually converge to the actual optimal policy [SB98]. An algorithm for standard Q-learning is given in Appendix A (page 78).

Typically a value function of Q-learning is defined in a tabular format that maps state-action pairs to numerical values. These numerical values provide the basis of selecting which action to perform for a particular state. A value function can also be defined in a neural network with an incremental style updating rule [Gos03]. In a Q-learning algorithm that is coupled with neural networks, each action is associated with its own neural network. Each of these action neural networks takes a state as their input and outputs the value of taking that action from the input state. The algorithm for this form of Q-learning is given in Appendix B (page 79).

2.2.2 Case Studies

In this section, we will analyze some representative cases of learning algorithms applied to robotics. A competence acquisition learning algorithm can be defined as the process used for an agent to acquire a new behavior through interacting with its environment [Neh92]. And a competence adaptation learning algorithm can be defined as the process used by an agent to modify an existing behavior so that the behavior's performance improves with respect to the agent's interactions with its environment [Ark98]. In both competence acquisition and competence adaptation, the goal for the learning algorithm is to improve a behavior's performance [Ark98].

2.2.2.1 Example of Competence Acquisition

The *Really Useful Robot* (RUR) project conducted at the University of Edinburgh provides a good example of competence acquisition learning in a robot agent [NHS89]. In the classification of robot controllers discussed earlier, this architecture would be classified as a reactive controller since it just reacts to situations without any planning. The RUR control architecture was developed in order to explore the question of competence acquisition learning in a robot agent [NHS89]. This control architecture uses a set of *instinct rules* to determine when the architecture trains an association between an input vector and an action to its associative memory. Instinct rules act as conditions placed on the robot's sensor readings (e.g. a reading on a distance sensor). This usage of instinct rules shapes the perceptron network of the agent. Moves that violate the agent's instinct rules are not learned by the robot's associative memory, whereas moves that do not violate the agent's instinct rules are learned by the robot.

The process of learning in this architecture is accomplished by pushing the outputs of a feed forward perceptron neural network towards a desired output value. It should be pointed out that the output value generated by the neural network for a particular state may not be the desired output. The learning by the perceptron network in this architecture does not go on until the desired output has been achieved but rather for a certain number of iterations.

The learning algorithm [Neh95] in this architecture works by checking the current state of the robot against the robot's instinct rules. If any of the instinct rules are violated, an input vector of the current state of the robot is generated. This input vector is then sent into the robots associative memory. A vector of possible actions is generated by the associative memory for the architecture to execute. The choice of possible actions from the action vector proceeds by choosing the first action in the vector to execute first. The retrieved action is executed and if the resulting state satisfies the robot's violated instinct rule, then the association between the input vector and action is taught to the robot's associative memory. Otherwise, if the robot's violated instinct rule is still violated, the next action in the action vector is then executed for a longer period of time than the first unsatisfactory action. According to the authors [Neh95], the execution of an action for a longer period of time is done so that the robot can get itself out of "deep problems". The architecture repeats this process until an action that satisfies the robot's violated instinct rule is found. If a satisfactory action cannot be found within the action vector, then a random action is generated from the robot's set of possible actions. If the randomly generated action satisfies the robot's violated instinct rule, then the association between the input vector and the action is taught to the robot's associative memory. Otherwise, if the robot's violated instinct rule is still violated, another random action is generated and is executed for a slightly longer period of time than the first unsatisfactory action. This process of generating and testing random actions continues until an executed action satisfies the robot's violated instinct rule. Once the robot's instinct rule has been satisfied, the last executed actions continues to be performed until one of the robot's instinct rules becomes violated. When an instinct rule becomes violated, the architecture executes the above learning algorithm until the violated instinct rule becomes satisfied.

The RUR control architecture has several difficulties. In the RUR controller, the associative memory is where the robot's learning of competences is dynamically and incrementally shaped via the robot's interactions with its environment. Actions that are considered good cause the associative memory to be pushed towards choosing an action in a particular state. This architecture picks actions incrementally from the action vector [Neh92]. Thus it does not exploit the knowledge stored in the neural network as much as it should. Another problem with this architecture is that when it exhausts all of the suggested actions in the action vector, it will then choose actions at random for different durations until the robot goes into a non-violated instinct rule state. When the control architecture gets into the phase of the learning algorithm where solely random actions are chosen, the architecture is no longer exploiting prior knowledge that it has learned about a state. In this case, the architecture is just randomly exploring. Once it has found an action that satisfies its instinct rules, it then teaches this action to the associate memory for the first instinct rule violated state. However, this action could be inappropriate for the first instinct rule violated state.

Despite the above difficulties, the RUR controller was able to learn and perform the tasks of obstacle avoidance, wall following, corridor following, phototaxis and box pushing [Neh94]. These tasks were accomplished through modifying the RUR controller's set of instinct rules for each task. As stated above instinct rules are conditions that are put on the robots sensors. When an instinct rule is violated, actions are performed until the instinct rule is satisfied. For example, say we have a robot with a forward sensor and two front whisker sensors (one to the left and one to the right). A possible set of instinct rules for wall following could be:

1. Forward sensor is true.
2. Front left whisker sensor is true XOR front right whisker sensor is true.
3. Front left whisker is false or front right whisker is false.

It should be noted that the order of the instinct rules is important. With this set of instinct rules the robot will operate in the following manner. When the robot is turned on it will initially be stationary, thus instinct rule #1 is violated, which causes the RUR controller to try actions until the robot is moving forward. After satisfying instinct rule #1, instinct rule #2 is violated, because none of the robot's whisker sensors are in contact with any objects. This causes the robot to try actions until it hits an obstacle (in our case a wall). Instinct rule #2 becomes satisfied once the robot has hit an obstacle with one of its whisker sensors. Now the robot is violating instinct rule #3, because one of its whisker sensors is touching the wall. In order, to satisfy instinct rule #3 the robot must try actions until it has moved away from the wall. After instinct rule #3 is satisfied the first instinct rule is tried again.

The Edinburgh researchers that created the RUR controller [Neh94] primarily used simple push button and whisker sensors for all their robots that used the RUR controller. By using simple sensors the RUR creators made it easy to check if an instinct rule had been violated since the sensors had only two discrete states. Unfortunately, we were unable to find research on how the RUR controller would scale up to sensors that return continuous states (e.g. Sonar or IR sensors).

It appears that the RUR researchers in designing their architecture have taken some ideas from the fields of Cybernetics, Neural Networks and Reinforcement learning. The idea of using functions as couplings between sensors and motors in the RUR controller is a very common idea in Cybernetics [Bra86]. The RUR researchers used an associative memory (neural network) as a dynamic function that uses sensor values as inputs and outputs actions for the motors to execute. In this architecture the instinct rules act as a means of determining which actions to learn (reward) and which actions to throw away (punish). The RUR researchers use the controller's associate memory almost like a value function from reinforcement learning in that it outputs a set of actions for a particular sensor state. However, it differs from a reinforcement learning value function in that it does not determine the value of an action for a state. The RUR associate memory only creates a vector of possible actions for the robot to execute and not a particular value associated with those actions.

2.2.2.2 Examples of Competence Acquisition and Competence Adaptation

Carbot Architecture

A good example of a competence acquisition and competence adaptation learning algorithm is shown in the Carbot architecture that was designed by Meeden, McGraw and Blank [MMB93]. In this architecture, the robot has to learn the value function for the task of navigation in a simple environment [MMB93].

Like in the RUR project [Neh92], the Carbot architecture combines elements of connectionist methods and reinforcement learning into the architecture. In this architecture, the neural network is trained with a sophisticated learning algorithm called *Complementary Reinforcement Back-Propagation* (CRBP) [MMB93]. Meeden, McGraw, and Blank [MMB93] use conditions on the robots sensors in order to indicate when the robot should either be rewarded or punished. This use of conditions placed on the robot's sensors is similar to instinct rules in the RUR project [Neh92]. This architecture works by using its neural network to generate an action to be executed based on the robot's current state. If the action does not violate any of the conditions put on its sensors, then the prior state and successful action are updated in the neural network by using the CRBP learning algorithm with a step size set at 0.3. If an action is unsuccessful, then the prior state and unsuccessful action are updated to the neural network by using the CRBP learning algorithm with a step size of 0.1. Thus, if an action is successful (the action satisfied the robot's sensor conditions), the neural network is pushed towards that action, otherwise the neural network is pushed away from that action [MMB93]. This process of generating actions repeats for the entire run of the robot.

An interesting thing about this architecture is that it does not limit the robot's moves to a certain set of moves like the RUR architecture, because the Carbot architecture has neural networks directly connected to its motor inputs. The number of states for the

Carbot architecture to learn is limited by the number of sensors and types of sensors that are employed. The Carbot architecture employs touch sensors and light sensors. As well, the environment used to evaluate the Carbot architecture was a simplified partially observable static environment [MMB93]. The Carbot architecture does not randomly generate moves in the same sense as the RUR architecture. The neural network used by the Carbot architecture is initially untrained; thus its output values are going to be random values. The training of the architecture's neural network as stated above shapes the values that are outputted by the neural network. Thus, the Carbot architecture has an implicit random move component. The actions generated by the architecture for unknown states will be essentially random.

This architecture has proven to be successful in learning and adapting the competences of phototaxis and obstacle avoidance [MMB93]. However, the Carbot architecture has two significant drawbacks. The first difficulty with this architecture is that it requires a learning phase of 3000 decisions [MMB93] before it is able to perform its tasks competently within its environment. This leads to the second drawback; the robot's learning is tailored specifically to the environment in which it has been trained. Unfortunately, we have been unable to find any research involving this architecture where the controller has been trained in one environment and then moved into another environment; thus the adaptability of the architecture is questionable. Finally, the Carbot architecture has only been used in simplified static environments. We have been unable to find any research where the Carbot controller has been deployed in more complex environments. It is therefore questionable whether the architecture would operate well in more complex environments.

Subsumption-based Architecture

Another good example of learning algorithms for competence acquisition and competence adaptation is found in the architecture designed by Mahadevan and Connell [MC92]. In this architecture, the robot has to learn a value function for box pushing in a simple environment. Their architecture was based on the subsumption architecture [Bro86], so that the box pushing task could be split into the following behaviors for the robot to acquire and adapt [MC92]:

Finding a Box: This behavior is used by the agent in order to find a box.

Pushing a Box: This behavior is used by the agent in order to push a box.

Getting unwedged: This behavior is used by the agent in order to get unwedged from an immovable obstacle.

These behaviors are prioritized in the subsumption architecture such that the getting unwedged behavior has the highest priority, the pushing a box behavior has the second highest priority and finding a box has the lowest priority [MC92]. By splitting up the box pushing task into the above behaviors, Mahadevan and Connell have effectively parallelized the reinforcement learning algorithm used in this architecture. In this architecture, sensors are used to determine the state of the agent and the applicability of the agent's behaviors. As well, the sensors are used by the reward function of the agent [MC92]. Q-learning is used by this architecture in order to train the agent in the competence of box pushing [MC92].

The designers of this architecture point out that it is a non-trivial task to decompose the learning task into subtasks and reward functions [MC92], in that, there are no guidelines on how to setup a reward function for each learning task[MC92]. Another problem that the designers of this architecture point out is that the number of states and actions can easily make the reinforcement learning problem with Q-learning intractable [MC92]. The reason for this is that it is not possible to store the value of every state-action pair when there are a large number of states and actions. The designers of this architecture also point out that the generalization of states the agent can significantly reduce the number of states that need to be considered [MC92].

2.3 Summary

In this chapter, we have reviewed the three primary robot controller paradigms. As well we have briefly examined the field of robot learning. We have also reviewed learning algorithms used in cybernetics, neural networks and reinforcement learning. Finally we studied three representative robot learning controllers.

All three of the example robot learning architectures have common drawbacks. The first drawback is that these architectures are unable to function with any degree of competence when they are first deployed in an environment. A second difficulty with regard to the Carbot architecture [MMB93] and the subsumption-based architecture [MC92] is that learning in these architectures takes a great deal of time. The RUR architecture, however, does possess the ability to learn quickly due to its use of a perceptron network for its associative memory. The RUR architecture suffers a drawback due to its use of a perceptron network in that it acts erratically since the perceptron network can not approximate non-linearly separable functions. As well, all of these architectures have been used only in partially observable static environments. We have been unable to find any research that would indicate if any of these architectures would scale up to more dynamic environments.

The generalized Tutor-Student learning algorithm that we have proposed in this thesis allows us to deploy a robot, which is able to function competently in an environment as soon as it is deployed. The Tutor-Student learning algorithm learns quickly and does not make the robot act erratically. As well, our learning algorithm works in both partially observable static and dynamic environments. More importantly this learning algorithm can be used to extend any of the three primary robot controller paradigms, where as, the three example architectures have only been able to add learning capabilities to reactive controllers.

Chapter 3

Methodology

This chapter presents the methodology of the Tutor-Student learning algorithm, which attempts to imitate the interactions between a tutor and a student. The job of the tutor is to teach and guide the student, which also includes correcting the student when the student is wrong. The job of the student is to absorb the teachings of the tutor and try to improve upon those teachings. Through this learning algorithm, an existing robot controller can become more attuned to its environment, as well as becoming more adaptable to changes in its environment. In this chapter, we will start with a review of some prior research that has inspired this approach. We will also present the generalized Tutor-Student learning algorithm as well as an application based upon this method.

3.1 Inspirations

The Tutor-Student learning algorithm was inspired by prior experiments with our earlier prototype 1 robot controller, as well as some ideas from the fields of reinforcement learning and cybernetics. In the following sections, we will briefly present these inspirations.

3.1.1 The Prototype 1 Robot Controller

3.1.1.1 Overview

The prototype 1 controller was designed to acquire behaviors based upon a set of evaluation conditions. This controller is basically a modified subsumption controller, where each behavior is given a rule system, a learning system and a set of conditions. The goal of this robot controller is to teach each behavior's learning system how to act through a combination of the behavior's rule system and through interaction with the robot's environment.

The rule system in each behavior is a non-deterministic rule system, in that it randomly chooses from a set of situation applicable rules. A non-deterministic rule system was chosen in order to allow equally applicable rules a chance to be activated. For example, if there is a fork in the path and no other knowledge is known about which path is better, then taking the left fork is as good as taking the right fork.

The learning system for each behavior uses a modified Q-learning algorithm, which is different from the standard Q-learning approach. The main difference between them is in our use of the function approximator, which is a method used to generalize a desired function from only a subset of known inputs and outputs. For each possible action, we used several incremental backpropagation neural networks to approximate the action's value function. By using multiple neural networks for each action and averaging their outputs, the modified Q-learning can estimate the value function more accurately.

The set of conditions for each behavior is used to indicate when the behavior is applicable, as well as when to reward or punish the learning system.

3.1.1.2 Limitations

In experiments, the prototype 1 robot controller failed to show any long term competence acquisition. The experimental results for the controller were in fact quite erratic for experiments with long decision runs. There are several possible reasons for this. However, in this section we will only focus on the limitations that became apparent with this controller.

One of the main limitations is its inability to construct a sequence of actions for getting out of bad situations. The primary reason is that both the Q-learning algorithm and the rule system choose actions based on the current situation. Consequently, the rule system has no proper sequencing of actions that the Q-learner could learn from. As well, it takes a lot of time and experiences for the Q-learner to build up a sequence of actions for its value function.

Another limitation that the prototype 1 controller has is that the rule system and the Q-learner often disagree with each other. A common example of such a situation occurs when the robot hits an obstacle and rule system causes the robot to back up and then the Q-learner has the robot go forward again. The reason why this disagreement occurs can be explained as follows. In each behavior, the Q-learner makes decisions until a behavioral condition is violated. Then on the behavior's next decision, the rule system chooses an action. However, when the modified Q-learner punishes an action, it does not punish it enough so that even though the action's value is decreased, it is not less than any of the other actions' values. In this case, insufficient punishment by the Q-learner leads to a repeat of the punished action for that state, (i.e., the Q-learner would undo the action previously chosen by the rule system).

3.1.2 Inspirations from Reinforcement Learning

In the field of reinforcement learning, the value function is used to store the values of state-action pairs [SB98]. Typically a function approximator is used instead of an explicit table to represent the value function [Gos03]. The reason for this is that a function approximator allows for the generalization of the value for state-action pairs and uses less memory to store those values [Gos03]. This means that the learner does not necessarily have to experience every state in its environment, because the value of state-action pairs can be generalized by the function approximator [Gos03].

3.1.3 Inspirations from Cybernetics

The field of cybernetics has shown that the behavior of a machine can be modified through the amplification of its actions [Ash64]. This means that it is possible to change the behaviors of an existing machine without changing the rules governing that machine.

3.2 Generalized Tutor-Student Learning Algorithm

3.2.1 Overview

In the Tutor-Student learning algorithm, the tutor is an existing proven robot controller and the student is a function approximator, which is charged with learning state-action pairs. This learning algorithm attempts to improve the actions of either the tutor or the student by amplifying either of those actions.

A generalized version of this algorithm works in the following way. The robot's *sensors* retrieve the *current state* of the robot and its environment. The current state is then supplied to both the *tutor* and *student* modules. Each of these modules then generates an action based upon the current state. The generated actions, which we call the *tutor action* and *student action* respectively, are then outputted to the *action selector* module, which chooses an action. This *selected action* is then given to the *amplifier* component for possible amplification. The amplifier produces an action called the *resulting action*, which is sent to the robot's *actuators* for execution. This entire process is called the *propagation phase* of the algorithm. Note that the student's actions are based upon previously evaluated actions. The evaluation of an action for a state happens during the *pre-evaluation phase* and *evaluation phase* of the algorithm. The pre-evaluation phase is used to gather information needed for the evaluation phase and happens during the same time period as the propagation phase. During the evaluation phase the resulting action is evaluated by the *evaluator* module based on the *resulting state* of the robot and its environment. The resulting state is produced by executing the resulting action from the propagation phase of the algorithm.

The following sections will discuss the components used by the Tutor-Student learning algorithm as well as the three phases of the algorithm.

3.2.2 Components

The Tutor-Student learning algorithm utilizes the following modules: tutor, student, action selector, amplifier, and evaluator. The robot's sensors and actuators are not included in this discussion since they are components that are provided by the robot.

3.2.2.1 Tutor

The tutor is the name given to the existing robot controller. This component is responsible for providing basic behaviors in order, to guide the learning of the student. As well the tutor provides behaviors that allow the robot to operate in critical situations.

3.2.2.2 Student

The student is the learning component for the Tutor-Student learning algorithm. It can be any type of function approximator. The job of the student is to learn actions for given situations.

3.2.2.3 Action Selector

The action selector module is used to select either an action generated by the tutor or the student. This module works by comparing the types of actions generated by the tutor and the student. If both actions have the same type, then the student action is selected, otherwise the tutor action is selected. In this way the selected action is always of the same type as the tutor's action in order to guarantee the basic behavior of the robot.

3.2.2.4 Amplifier

The amplifier module's job is to decide if the components of an action are to be modified through amplification by a preset amount. Amplification is used to modify the actions of either the tutor or the student. By using amplification as a quantity for learning, we are able to modify the behaviors of the tutor without changing the rules governing the tutor.

3.2.2.5 Evaluator

The evaluator module is used to determine if the previous action performed by the agent was appropriate for the previous state. The appropriateness of an action is decided by checking if the state resulting from the execution of the action has violated any of the agent's operational conditions. An operational condition is an assertion based on the current state of the agent. Operational conditions are used to determine which action the student is taught for the previous state. An example of such condition is "If the robot hits an obstacle". The evaluator module evaluates an action using the procedure outlined in the evaluation phase of the learning algorithm (see page 39 for details).

3.2.3 Algorithm

The Tutor-Student learning algorithm is split into three different phases: the propagation phase, the pre-evaluation phase and the evaluation phase. The propagation phase and the pre-evaluation phase happen in parallel during the same time period. These two phases are separately described in two diagrams and in each diagram only modules relating to the corresponding phase are illustrated. This is done in order to facilitate the understanding of the algorithm. The evaluation phase happens one time period after the propagation and pre-evaluation phases.

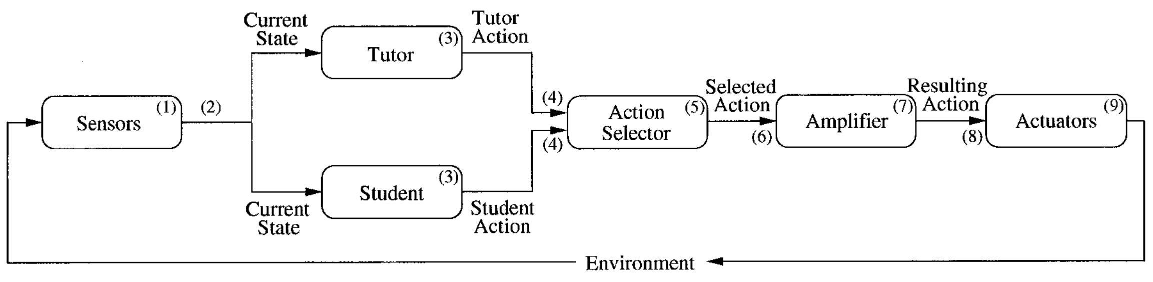

Fig. 6: Tutor-Student Propagation Phase

Figure 6, shows the propagation phase of the Tutor-Student learning algorithm. The numerical labels in Figure 6 refer to the steps in the following procedure for the propagation phase.

1. Get the current state from the agent's sensors.

2. The agent's sensors send the current state to the tutor and student modules.

3. The tutor and student modules each generate an action based on the current state.

4. The tutor and student modules send their actions to the action selector module.

5. The action selector module selects an action from its inputs.

6. The action selector module outputs the selected action to the amplifier module.

7. The amplifier module possibly amplifies the selected action.

8. The amplifier module sends its resulting action to the agent's actuators.

9. The agent's actuators execute the resulting action.

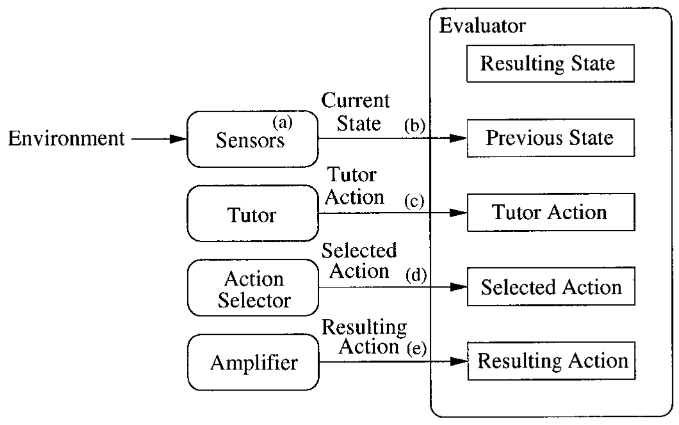

Fig. 7: Tutor-Student Pre-Evaluation Phase

Figure 7, demonstrates the pre-evaluation phase, which is a preparation process for the evaluation phase. The information stored in the evaluator can be seen as the evaluator's model of the robot's current situation. The letter labels in Figure 7, correspond to the steps for the following pre-evaluation procedure.

a. Get the current state from the agent's sensors.

b. The agent's sensors send the current state to the evaluator module and the evaluator module stores the current state as the previous state.

c. The tutor sends its action to the evaluator module and the evaluator module stores the tutor action.

d. The action selector sends its selected action to the evaluator module and the evaluator module stores the selected action.

e. The amplifier sends its resulting action to the evaluator module and the evaluator module stores the resulting action.

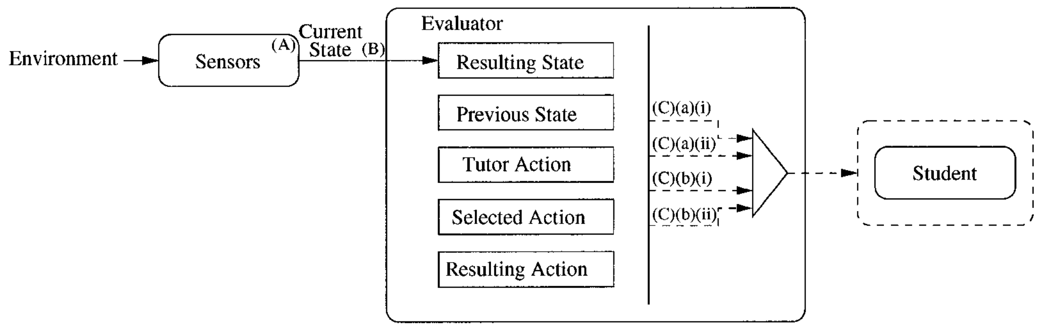

Fig. 8: Tutor-Student Evaluation Phase

The evaluation phase happens one time period after the propagation and pre-evaluation phases. Its process is shown in Figure 8, whose labels correspond to the steps for the following evaluation procedure.

A. Get the current state from the agent's sensors.

B. The agent's sensors send the current state to the evaluator module and the evaluator module stores the current state as the resulting state.

C. The evaluator module checks if the stored resulting action was amplified.

   (a) If it was amplified, then the evaluator tests the agent's operational conditions with the stored resulting state.

      i. If any of the agent's operational conditions are true, then the evaluator module teaches the student module the stored previous state and the stored selected action.

      ii. Otherwise, the evaluator module teaches the student module the stored previous state and the stored resulting action.

   (b) Otherwise, the evaluator module tests the agent's operational conditions with the stored resulting state.

      i. If any of the agent's operational conditions are true, then the evaluator module teaches the student module the stored previous state and the stored tutor action.

      ii. Otherwise, the evaluator module teaches the student module the stored previous state and the stored selected action.

The essence of the evaluation phase is to choose an action stored during the pre-evaluation phase of the learning algorithm, which is to be taught to the student with the previous state. The choice of action taught to the student is based upon whether the stored resulting action was amplified during the previous propagation phase and whether the resulting action was successful. The criterion of a successful action is that none of the robot's operational conditions has been violated. In this algorithm if any of the operational conditions is true, then a violation has occurred.

If the resulting action was amplified, then the evaluator checks the robot's operational conditions. If any operational conditions are true, then the resulting action should not be taught to the student. Instead the selected action should be taught to the student. However, if none of the operational conditions are true, then the resulting action is taught to the student.

If the resulting action was not amplified, then the evaluator checks the robot's operational conditions. If any of these conditions are true then the resulting action should not be taught to student. The tutor action is then taught to the student because the tutor action is assumed to always be a successful action. Otherwise, if none of the operational conditions is true then the resulting action is taught to the student.

3.3 Application

This section presents the details of an implemented version of the generalized Tutor-Student learning algorithm. This version uses a behavior based robot controller called *prototype 2* as the tutor and an incremental backpropagation neural network as the student. The robot controller resulting from applying the Tutor-Student learning algorithm to the prototype 2 robot controller is called *prototype 2A*.

3.3.1 Tutor

The prototype 2 robot controller is designed for the tasks of obstacle avoidance and attractor path planning with predator avoidance in partially observable static and dynamic environments.

The task of obstacle avoidance is for the agent to wander around its environment without hitting any obstacles. The task of attractor path planning with predator avoidance is one in which the agent has three subtasks: The first subtask is to avoid hitting obstacles in its environment; The second subtask is for the agent to move towards a prey (also called attractors) and to move away from the prey after it gets within a certain range of the prey; the third subtask is for the agent to detect predators (also called repulsers) and flee from them when they get within a certain range.

A partially observable environment is an environment where the agent does not have full knowledge about its environment. A static environment is an environment that does not change while the agent is deciding what to do next and a dynamic environment is an environment that has the ability to change as the agent is making decisions.

The heart of the prototype 2 robot controller is a state planner that acts as the sequencing rule system for the controller. A state planner consists of a set of states where each state in the state planner possesses its own set of rules. All the rules in the state planner have three components: condition, action and transition. The condition component of a rule is used to indicate if the state rule is applicable. If it is applicable, then the action component gives an action for the robot to execute; as well the transition component gives the next state transition for the state planner. The state rules within each state are tested for applicability from the first rule to the last in the state. Therefore there is a priority ordering of state rules in each state. The state planner used by the prototype 2 robot controller consists of the following states:

Subsumption State: This state consists of a priority ordering of the following types of state rules:

Obstacle Avoidance Rules: These rules govern how the robot controller avoids hitting obstacles in its environment.

Predator Avoidance Rules: These rules govern how the robot avoids predators. A predators for the prototype 2 robot controller is any blue object found in the controller's environment. These blue objects come in two varieties: a static blue box and a moving blue robot.

Fleeing Rules: These rules govern how the robot flees after it gets too close to prey. A prey for the prototype 2 robot controller is any red object found in the controller's environment. These red objects come in two varieties: a static red box and a moving red robot.

Seeking Rules: These rules govern how the robot locates and tracks prey in its environment.

Wander Rules: These rules govern how the robot wanders about its environment. These rules consider open areas in the robot's environment good areas for the robot to wander to.

Note that the subsumption state rules are not necessarily grouped together by type in the state's list of rules.

Forward State: This state provides the forward behavior of the robot.

Avoid Spin State: This state provides a spin behavior for obstacle avoidance. This spin behavior happens when the robot is close to obstacles on its front side, left side and right side.

Prey Spin State: This state provides a spin behavior for fleeing from prey. This behavior is triggered when the robot gets too close to its prey, where upon the robot will spin until it no longer sees its prey.

Predator Spin State: This state provides a spin behavior for fleeing from predators. This behavior is triggered when the robot gets within a certain range of a predator. Once the robot is within that range, the robot will spin until it no longer sees the predator.

The actions generated by the state planner consist of three components: translation, rotation and duration. Translation is a continuous floating point number between -1.0 and 1.0, which indicates whether the robot is to go forwards or backwards. Rotation is a continuous floating point number ranging from -1.0 to 1.0, which indicates whether the robot is to turn left or right. Finally, duration is a continuous floating point value greater than or equal to 0.0, which describes how long the robot should perform the action. The prototype 2 robot controller has five types of actions: stop, forward, backward, left and right.

The prototype 2 robot controller has been tested throughly in various environments and has performed its tasks successfully. The success of the prototype 2 controller made it a good candidate for being the tutor module for our implemented Tutor-Student learning algorithm.

3.3.2 Student

The student as mentioned earlier is a function approximator. We chose to use an incremental backpropagation neural network as a function approximator, because of its well known ability to approximate a multivariate function [Gos03]. The student's sensor inputs were converted to a binary representation in order to improve the performance of the neural network. As well, the student's outputs are in a binary representation and are interpreted to other values which are action description, amplification level and ampbit. These output values are taught to the student during the evaluation phase of the learning algorithm.

The action description is used to retrieve the component form of an action used by the tutor. The amplification level is used to compute the amplification value, which is then added or subtracted to the appropriate component of the retrieved action. The amplification value is equal to an amplification constant times the amplification level. For this implementation the amplification constant is equal to 0.1. The ampbit is used to determine if further amplification should occur on the action generated by the student. If the ampbit is 1 then the student's action is a candidate for further amplification, else if the ampbit is 0 then the student's action is not a candidate for further amplification.

3.3.3 Action Selector

In the prototype 2A robot controller, the action selector module compares string descriptions of actions generated by the tutor and the student. If the tutor action description is the same as the student action description, then the student action becomes the selected action along with its amplification level and ampbit, otherwise the tutor action becomes the selected action along with an amplification level of zero and ampbit set to one.

3.3.4 Amplifier

The amplifier module takes the selected action in its component form, (Translation, Rotation, Duration), along with a string description of the action, and the selected ampbit. Based on the string description of the action, the amplifier then amplifies the appropriate component of the action depending on the value of the ampbit or whether the controller is still in its learning phase.

A learning phase is employed in the amplifier module because we are using an incremental backpropagation neural network as a function approximator for the student. If the ampbit is zero at the start of the learning algorithm then amplification may not be applied to the selected action.

The algorithm for the amplifier checks to see whether the ampbit is true or the robot controller is still in learning phase. If either of these conditions is true then the action is amplified, otherwise the action will not be amplified.

The amplifier for this applied Tutor-Student learning algorithm only amplifies the translation or rotation components of an action and not the duration component. The effectiveness of amplifying the duration component will be investigated in our future work (see section 5.2.2 on page 70).

3.3.5 Evaluator

The evaluator module works as described above in the generalized Tutor-Student learning algorithm (Section 3.2.2.5 on page 36), except that the evaluator teaches the student an action description, an amplification level and an ampbit value with the previous state. The prototype 2A robot controller has only one operational condition, which is "If the robot hits an obstacle". For successful actions, the student is taught with an ampbit equal to 1, whereas for unsuccessful actions the student is taught with an ampbit equal to 0.

When the evaluator teaches a state-action pair to the student, it is done for a fixed number of iterations, called the teaching rate. The teaching rate for the incremental backpropagation neural network is how many times the state-action pair is taught to the neural network at one time.

Chapter 4

Experiments

This chapter focuses on presenting the experiments that have been conducted using the prototype 2 and prototype 2A robot controllers that we introduced in Chapter 3. The parameters of the experiments will be explained, as well as the evaluation procedures used. The experimental results for each experiment will also be presented and discussed.

4.1 Environments

The prototype 2 and prototype 2A robot controllers will operate in two types of environments: partially observable static simulated environment and partially observable dynamic simulated environment. A partially observable environment is an environment where the robot's sensors only have partial access to the state of the environment. For example a robot in an office-like environment would be unable to know what is happening on the other side of a wall (unless the robot has x-ray vision). A static environment is an environment that does not change while the robot is deciding what to do next. A dynamic environment is an environment that can change while the robot is deliberating what to do next.

The controller's environment is a simulated environment generated by the Player/Stage simulator[1], which is controlled through the Pyro[2] programming environment. Pyro is a programming environment designed for constructing robot controllers and artificial in-

[1]Player and Stage are freely available under the GNU General Public License from HTTP://playerstage.sourceforge.net

[2]Pyro is freely available under the GNU General Public License from HTTP://pyrorobotics.org

telligence programs [BKMY03].

The robot controller's environment consists of two parts. The first part is the robot's internal environment, which is a simulated Pioneer 2 DX robot[3]. The Pioneer 2 DX has a differential drive train and 16 sonar sensors positioned around its 50cm $\times$ 50cm body. A subset of these sonar sensors will be used in the robot controller experiments. The second part of the controller's environment is its external environment, which is its immediate surroundings.

For the experiments involving a static environment, the prototype 2 and prototype 2A robot controllers will operate alone in their environments, whereas, in the experiments involving dynamic environments both controllers will operate in their environments along side predator and prey robots. A predator robot uses a subsumption-like controller to seek and chase the prototype 2 and prototype 2A robot controllers around its environment. A prey robot uses a subsumption-like controller to wander around its environment.

Generally speaking, there are several concerns about simulated environments. The first one is that simulators provide an agent with accurate sensors and actuators, which real robots do not possess. The second concern is that simulators usually make strong assumptions about the robots and environments.

With regard to the first concern, the Stage simulator does provide relatively accurate sensors and actuators. However, since we are in early developing stages, we preferred to take the approach of starting with accurate sensors and actuators to make sure that the algorithm would function properly under ideal situations. The question of how the

[3]The simulated Pioneer 2 DX is provided with the Player/Stage simulator software. The Pioneer 2 DX itself is an actual robot realized in hardware.

learning algorithm cooperates with real robots is left for future work.

As for the second concern, we have not been able to find any papers that discuss the assumptions made by the Stage simulator. However, we have found several papers [JS02, MWBDW02, VSSM02] where researchers have developed clients using the Stage simulator that work with little or no modification on real robots.

4.2 Evaluation

The prototype 2 and prototype 2A robot controllers are evaluated based upon how well they perform the tasks of obstacle avoidance and attractor path planning with predator avoidance in four different environments. The task of obstacle avoidance will be evaluated by recording how many times the robot controller experiences a collision. The task of attractor path planning with predator avoidance will be evaluated by recording the number of times the robot controller locates attractors.

The robots will be started at the same locations and in the same order. The individual performances of the controllers in the tasks of obstacle avoidance and attractor path planner with predator avoidance will then be averaged over all trial runs. In each environment, the prototype 2 robot controller and each of three different learning phases of the prototype 2A robot controller will be given 10 trial runs, each of which will consist of 10,000 decisions. Three different learning phases of the prototype 2A robot controller were tested: they are 2500, 5000 and 7500 decisions.

4.3 Experiments

The prototype 2 and prototype 2A robot controllers are tested in four different environments: the *Chase-Avoid environment*, the *Cave-Box environment*, the *Cave-Predator-Prey environment*, and the *Cave-Predator-Prey-Box environment.* All of the these environments are modified versions of environments that come with the Player/Stage software.

4.3.1 Prototype 2A Controller: Parameters

In each of the following experiments, the prototype 2A robot controller is set with the following parameters as shown in Table 1.

Parameters	Values
Number of Neural Networks	1
Inputs Units	14
Hidden Units	30
Output Units	7
Learning Rate	0.3
Momentum Rate	0.9
Teaching Rate	3
Amplify Amount	0.1

Table 1: Prototype 2A Parameters

4.3.2 The Chase-Avoid Environment

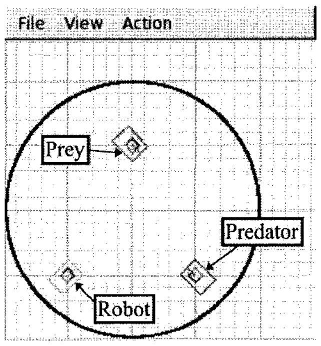

Fig. 9: The Chase-Avoid Environment

4.3.2.1 Description

The Chase-Avoid environment shown in Figure 9 is a partially observable dynamic environment. This environment is designed to examine how well both controllers, 2 and 2A, are able to avoid obstacles and locate a prey robot in a small enclosed environment, while avoiding a predator robot. Due to the small area of the environment, there is a strong possibility of collisions with other robots.

4.3.2.2 Results and Discussion

The results of the prototype 2 and prototype 2A robot controllers for their sets of trial runs are given in Table 2.

Controller		Obstacle Avoidance		Attractor Path Planning with Predator Avoidance	
		Avg. Number of Collisions	Standard Deviation	Avg. Number of Attractors Found	Standard Deviation
Prototype 2		31.2	57.62	84.6	18.11
Prototype 2A with different learning phases	2500	20.6	6.82	130.1	12.93
	5000	19.4	5.85	125.4	16.59
	7500	31.5	17.10	125.5	18.42

Table 2: Controllers Results for the Chase-Avoid Environment

Table 3 presents the average improvement of the prototype 2A robot controller over the prototype 2 robot controller for the tasks of obstacle avoidance and attractor path planning with predator avoidance.

Learning Phase (Number of Decisions)	Obstacle Avoidance Improvement	Attractor Path Planning with Predator Avoidance Improvement
2500	34.0%	53.8%
5000	37.8%	48.2%
7500	-0.96%	48.3%

Table 3: Average Improvement by the Prototype 2A Controller for the Chase-Avoid Environment

From the results shown in Table 2 and Table 3, it is demonstrated that for the task of obstacle avoidance, the prototype 2A robot controller performed significantly better or almost as well as the prototype 2 robot controller. As well, the results also demonstrate, for the task of attractor path planning with predator avoidance, that the prototype 2A robot controller performed significantly better than the prototype 2 robot controller.

For the Chase-Avoid environment, we can say that the Tutor-Student learning algorithm has greatly improved the performance of the prototype 2 robot controller for this environment.

4.3.3 The Cave-Box Environment

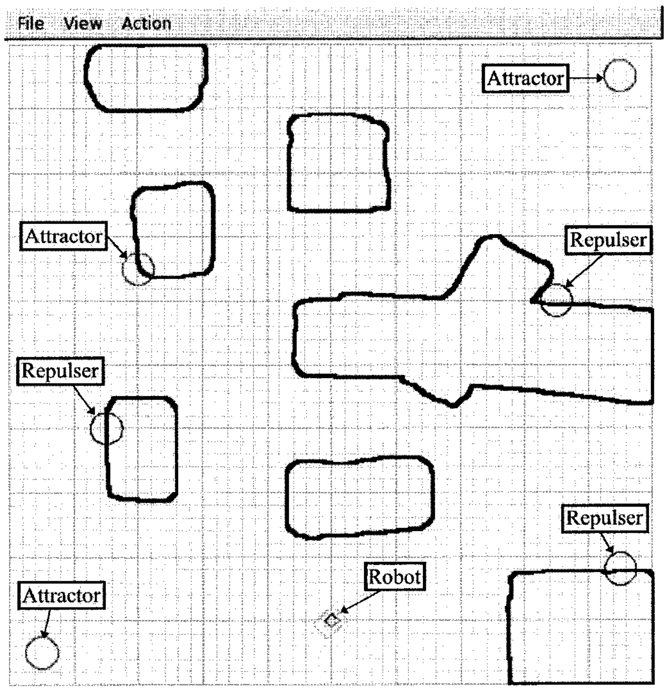

Fig. 10: The Cave-Box Environment

4.3.3.1 Description

The Cave-Box environment shown in Figure 10, is a partially observable static environment. It is designed to examine how well both controllers, 2 and 2A, are able to perform the tasks of obstacle avoidance and attractor path planning with predator avoidance, in a large complex environment with three static attractors and three static repulsers. This environment presents a challenge for both controllers in that they have to explore a complex environment in order to locate attractors.

4.3.3.2 Results and Discussion

The results of the prototype 2 and prototype 2A robot controllers for their sets of trial runs are given in Table 4.

Controller		Obstacle Avoidance		Attractor Path Planning with Predator Avoidance	
		Avg. Number of Collisions	Standard Deviation	Avg. Number of Attractors Found	Standard Deviation
Prototype 2		0.0	0.0	8.1	2.02
Prototype 2A with different learning phases	2500	4.4	4.14	41.6	4.22
	5000	7.1	9.80	39.1	7.20
	7500	2.0	3.37	40.9	5.86

Table 4: Controllers Results for the Cave-Box Environment

From the results in Table 4, it is clear that for the task of obstacle avoidance, the prototype 2A robot controller performed slightly worse than prototype 2 robot controller.

Despite this, the prototype 2A robot controller performed very well considering that in the worst case scenario, the prototype 2A controller's average collision is 7.1 out of the 10,000 decisions made.

Table 5, presents the average improvement of the prototype 2A robot controller over the prototype 2 robot controller for the task of attractor path planning with predator avoidance.

Learning Phase (Number of Decisions)	Attractor Path Planning with Predator Avoidance Improvement
2500	413.6%
5000	382.7%
7500	404.9%

Table 5: Average Improvement by the Prototype 2A Controller for the Cave-Box Environment

The results shown in Table 4 and Table 5 demonstrate that for the task of attractor path planning with predator avoidance, the prototype 2A robot controller performed significantly better than the prototype 2 robot controller.

For the Cave-Box environment, we can say that the Tutor-Student learning algorithm did not improve the performance of obstacle avoidance in the prototype 2 robot controller. However, the learning algorithm did significantly improve the prototype 2A robot controller at the task of attractor path planning with predator avoidance. It is within our expectation that due to the nature of this learning algorithm, (i.e. to conduct modification over tutor's actions) more collisions may occur because of this process.

4.3.4 The Cave-Predator-Prey Environment

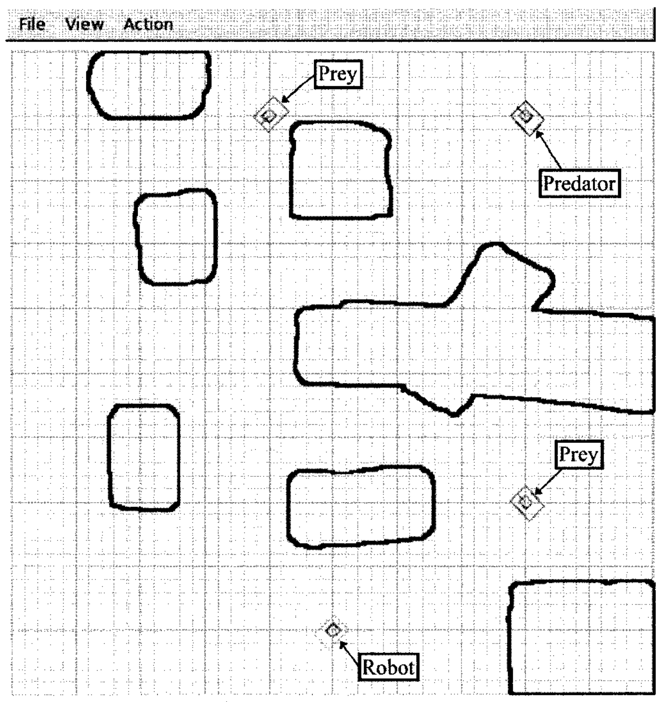

Fig. 11: The Cave-Predator-Prey Environment

4.3.4.1 Description

The Cave-Predator-Prey environment shown in Figure 11, is a partially observable dynamic environment. This environment is designed to examine how well both controllers, 2 and 2A, are able to avoid obstacles and locate prey (attractors), in a large complex environment with two prey robots and one predator robot. The task of attractor path planning with predator avoidance in this environment is a difficult task in that there are only two attractors and they are moving around a large complex environment.

4.3.4.2 Results and Discussion

The results of the prototype 2 and the prototype 2A robot controllers for their sets of trial runs are given in Table 6.

Controller		Obstacle Avoidance		Attractor Path Planning with Predator Avoidance	
		Avg. Number of Collisions	Standard Deviation	Avg. Number of Attractors Found	Standard Deviation
Prototype 2		1.6	1.84	29.2	22.00
Prototype 2A with different learning phases	2500	7.2	4.21	68.8	29.94
	5000	9.1	4.82	42.1	12.42
	7500	5.7	4.85	49.7	15.09

Table 6: Controllers Results for the Cave-Predator-Prey Environment

From the results shown in Table 6, it is clear that for the task of obstacle avoidance the prototype 2A robot controller performed slightly worse than prototype 2 robot controller. Despite this, the prototype 2A robot controller performed very well considering that the most average collisions of the prototype 2A controller is 9.1 out of the 10,000 decisions made.

Table 7, presents the average improvement of the prototype 2A robot controller over the prototype 2 robot controller, for the task of attractor path planning with predator avoidance.

Learning Phase (Number of Decisions)	Attractor Path Planning with Predator Avoidance Improvement
2500	135.6%
5000	44.2%
7500	70.2%

Table 7: Average Improvement by the Prototype 2A Controller for the Cave-Predator-Prey Environment

The results in Table 6 and Table 7 demonstrate that for the task of attractor path planning with predator avoidance, the prototype 2A robot controller performed significantly better than the prototype 2 robot controller.

For the Cave-Predator-Prey environment, we can say that the Tutor-Student learning algorithm did not improve the performance of obstacle avoidance in the prototype 2 robot controller, however, the learning algorithm did significantly improve the prototype 2A robot controller at the task of attractor path planning with predator avoidance.

4.3.5 The Cave-Predator-Prey-Box Environment

4.3.5.1 Description

The Cave-Predator-Prey-Box environment shown in Figure 12 is a partially observable dynamic environment. This environment is designed to examine how well both controllers, 2 and 2A, are able to avoid obstacles and locate attractors in a large complex environment. In this environment, the robot will be accompanied by two prey robots

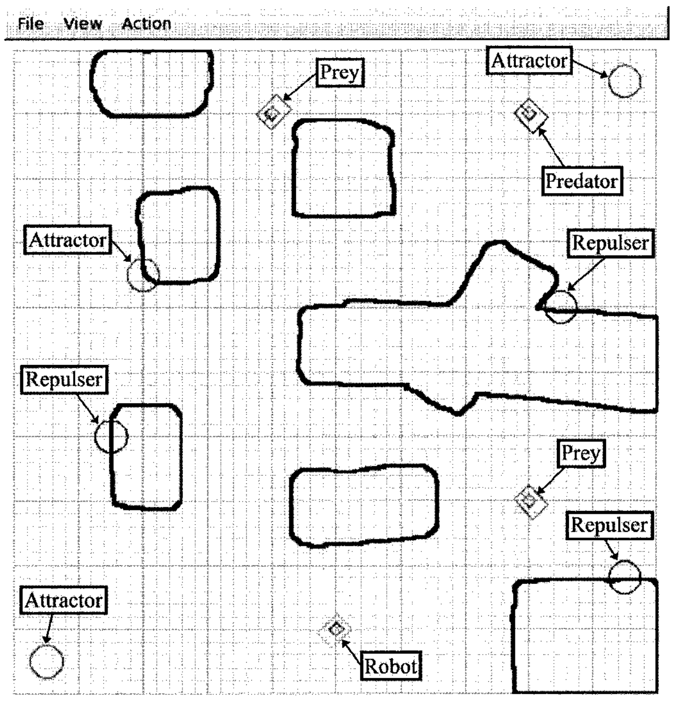

Fig. 12: The Cave-Predator-Prey-Box Environment

and one predator robot, which adds a dynamic element to this environment. As well, there will be three static attractors and three static repulsers located throughout the environment.

4.3.5.2 Results and Discussion

The results of the prototype 2 and prototype 2A robot controller, for their sets of trial runs are given in Table 8.

The results in Table 8, demonstrate that for the task of obstacle avoidance, the prototype 2A robot controller performed slightly worse than prototype 2 robot controller. Despite this point, the prototype 2A robot controller performed very well considering that the highest number of average collisions with prototype 2A controller is 9.2 out of the

Controller		Obstacle Avoidance		Attractor Path Planning with Predator Avoidance	
		Avg. Number of Collisions	Standard Deviation	Avg. Number of Attractors Found	Standard Deviation
Prototype 2		0.3	0.67	30.3	15.60
Prototype 2A with different learning phases	2500	9.2	5.47	85.1	18.13
	5000	8.5	4.57	76.2	13.69
	7500	4.4	1.78	78.5	9.50

Table 8: Controllers Results for the Cave-Predator-Prey-Box Environment

10,000 decisions made.

Table 9, presents the average improvement in the prototype 2A robot controller over the prototype 2 robot controller for the task of attractor path planning with predator avoidance.

Learning Phase (Number of Decisions)	Attractor Path Planning with Predator Avoidance Improvement
2500	180.9%
5000	151.5%
7500	159.1%

Table 9: Average Improvement by the Prototype 2A Controller for the Cave-Predator-Prey-Box Environment

The results shown in Table 8 and 9 clearly demonstrate that for the task of attractor path planning with predator avoidance the prototype 2A robot controller performed significantly better than prototype 2 robot controller.

For the Cave-Predator-Prey-Box environment, we can say that the Tutor-Student learning algorithm did not improve the performance of obstacle avoidance in the prototype 2 robot controller, however, the learning algorithm did significantly improve the prototype 2A robot controller at the task of attractor path planning with predator avoidance.

4.4 Conclusions

The experiments conducted on the prototype 2 and prototype 2A robot controllers have presented the effectiveness of the Tutor-Student learning algorithm. For the task of obstacle avoidance, the learning algorithm was able to modify the actions of the prototype 2 robot controller so that the controller performed either better or slightly worse. As stated above, the prototype 2A robot controller having slightly worse results than the prototype 2 robot controller for the task of obstacle avoidance is expected. The Tutor-Student learning algorithm learns through trial and error interactions with its environment. It is expected that during this learning process the robot will make mistakes. However, for all of these experiments, the prototype 2A robot controller was involved in collisions for less then 0.1% of the total decisions made by the controller, which is a significant statistic, since as far as we know, there is no other trail and error learning method that can perform as well.

In all of our experiments, the prototype 2A robot controller obtained the best results when the controller's learning phase was set to 2500 decisions. As stated previously

(Section 3.3.4 on page 46), a learning phase is used to force the learning algorithm to amplify the controller's actions, in order to ensure that the function approximator does not stop the amplification of the controller's actions too early, due to an initial spurious output value for the ampbit. By forcing amplification of the controller's actions for all of the experimental environments past 2500 decisions, it appears to have caused the controller to decrease its performance. This decrease in performance could be because that the student has reached a stable point where the amplification of the controller's actions is no longer required and further amplification past this point may cause the controller to decrease in its performance.

From our experiments using the implemented Tutor-Student learning algorithm we are unable to say if the vast majority of learning occurs during the learning phases. There are two reasons that make this difficult to determine. First, it is possible that one function approximator may require less training than another function approximator (e.g. the initial weights in one neural network can be more conducive towards a particular task than the initial weights in another neural network of the same type). As well, the Tutor-Student learning algorithm is designed to allow the robot to learn throughout its entire set of decisions, therefore learning maybe occurring all through the trial run of the robot. To determining whether there is a decision interval or set of decision intervals where the majority of learning takes place is left for future work (see section 5.2.5 on page 71 for more details).

In all of our experiments, the Tutor-Student learning algorithm considerably improved the performance of the prototype 2 robot controller in the task of attractor path planning with predator avoidance. This learning algorithm improved the average performance of the prototype 2 robot controller's ability to locate attractors by almost 100% to over 400% (depending on the environment).

Chapter 5

Conclusion

This chapter will present a summary of this thesis, as well as open questions for future research with the Tutor-Student learning algorithm.

5.1 Summary

This thesis has proposed the generalized Tutor-Student learning algorithm, which is an innovative method that is designed to convert an existing robot controller into a learning robot controller. This learning algorithm mimics the interactions between a human tutor and student. A sample application of the Tutor-Student learning algorithm has been presented.

In contrast to the learning robot controllers explored in Section 2.2.2 on page 21, the generalized Tutor-Student learning algorithm allows us to deploy a robot that is able to function competently in an environment. The robot controller is able to learn quickly and does not make the robot act erratically. As well our learning algorithm works in both partially observable static and dynamic environments. The Tutor-Student learning algorithm is able to add learning capabilities to any of the three primary robot controller paradigms, whereas other methods in the literature have only been able to add learning capabilities to reactive controllers.

Experiments have been conducted employing the generalized Tutor-Student learning algorithm on a behavior-based robot controller in four different test environments. The experimental results have shown the effectiveness of the generalized Tutor-Student learning algorithm upon a robot controller. For the task of obstacle avoidance, the learning algorithm was able to modify the actions of the prototype 2 robot controller so that the controller performed either better or slightly worse. For all of these experiments, the prototype 2A robot controller was involved in collisions for less than 0.1% of the total decisions made by the controller, which is a significant statistic, since as far as we know, there is no other trail and error learning method that can perform as well. The Tutor-Student learning algorithm also considerably improved the performance of the prototype 2 robot controller in the task of attractor path planning with predator avoidance. This learning algorithm improved the average performance of the prototype 2 robot controller's ability to locate attractors by almost 100% to over 400% (depending on the environment).

5.2 Open Questions

This section presents the open questions for future research with the Tutor-Student learning algorithm.

5.2.1 Streamlining the Tutor-Student Learning Algorithm

Can the Tutor-Student learning algorithm be streamlined? Since amplification is the only necessary learning quantity used by this algorithm, is it possible for the student to just learn states-amplification level pairs instead of state-action pairs? This modification will effect all three phases of the learning algorithm. This modified Tutor-Student learning algorithm is briefly described below.

Figure 13, shows this modified propagation phase.

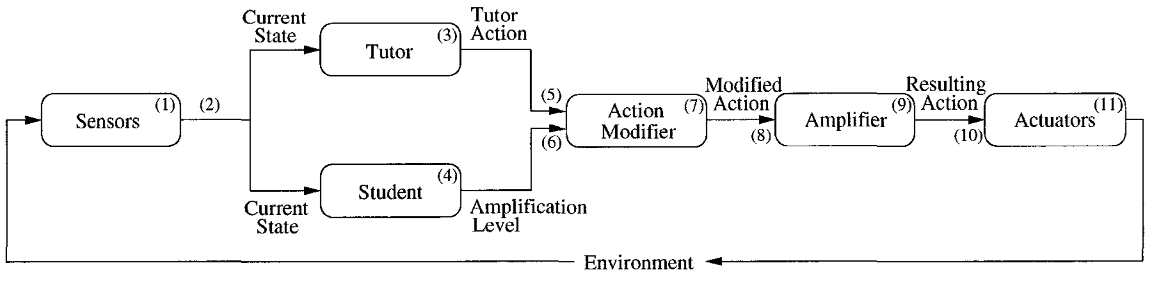

Fig. 13: Modified Propagation Phase

The numerical labels in Figure 13, refer to the steps in the following procedure for the propagation phase.

1. Get the current state from the agent's sensors.
2. The agent's sensors send the current state to the tutor and student modules.
3. The tutor module generates an action based on the current state.
4. The student module generates an amplification level based on the current state.
5. The tutor sends its action to the action modifier module.
6. The student sends its amplification level to the action modifier module
7. The action modifier module modifies the tutor's action with the student's amplification level.
8. The action modifier then sends the modified action to the amplifier module.
9. The amplifier module possibly amplifies the modified action.
10. The amplifier module sends its resulting action to the agent's actuators.

11. The agent's actuators execute the resulting action.

As can be seen from above, the student only outputs an amplification level for a given state, which is then applied to the tutor's action. Due to this change, the action selector module is no longer needed. Figure 14, shows the modified pre-evaluation phase.

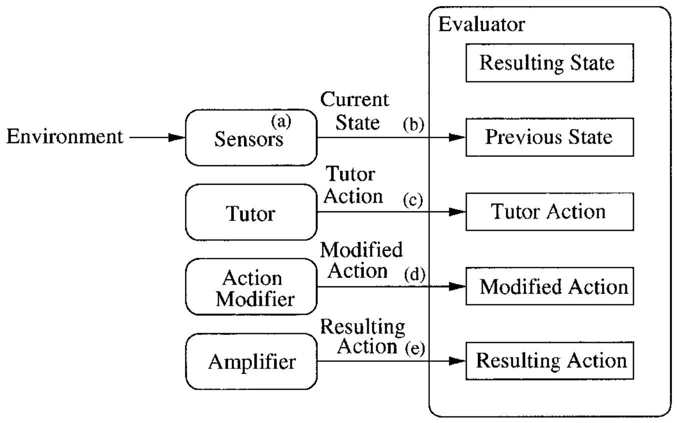

Fig. 14: Modified Pre-Evaluation Phase

The letter labels in Figure 14, correspond to the steps for the following pre-evaluation procedure.

a. Get the current state from the agent's sensors.

b. The agent's sensors send the current state to the evaluator module and the evaluator module stores the current state as the previous state.

c. The tutor sends its action to the evaluator module and the evaluator module stores the tutor action.

d. The action modifier sends its modified action to the evaluator module and the evaluator module stores the modified action.

e. The amplifier sends its resulting action to the evaluator module and the evaluator module stores the resulting action.

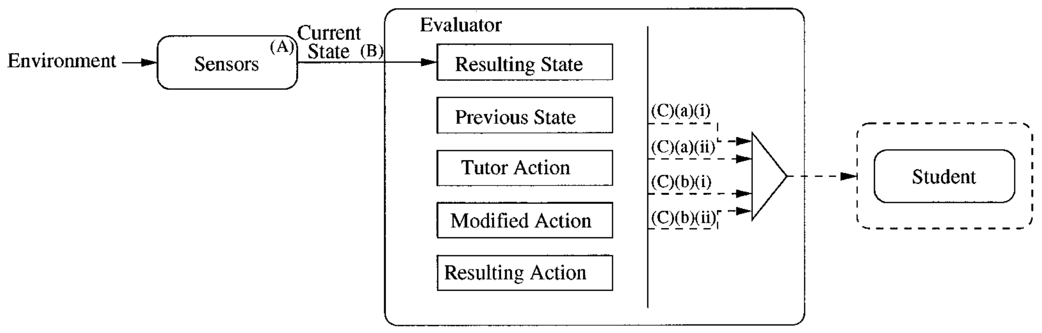

Fig. 15: Modified Evaluation Phase

The evaluation phase happens one time period after the propagation and pre-evaluation phases. Its process is shown in Figure 15, whose labels correspond to the steps for the following evaluation procedure.

A. Get the current state from the agent's sensors.

B. The agent's sensors send the current state to the evaluator module and the evaluator module stores the current state as the resulting state.

C. The evaluator module checks if the stored resulting action was amplified.

   (a) If it was amplified, then the evaluator tests the agent's operational conditions with the stored resulting state.

      i. If any of the agent's operational conditions are true, then the evaluator module teaches the student module the stored previous state and the stored modified action.

        ii. Otherwise, the evaluator module teaches the student module the stored previous state and the stored resulting action.

    (b) Otherwise, the evaluator module tests the agent's operational conditions with the stored resulting state.

        i. If any of the agent's operational conditions are true, then the evaluator module teaches the student module the stored previous state and the stored tutor action.

        ii. Otherwise, the evaluator module teaches the student module the stored previous state and the stored modified action.

5.2.2 Amplification Strategies

In this thesis, amplification is handled by adding a preset amplification amount to the appropriate component of an action. Are there any other amplification strategies? As well, is it possible to amplify more than one component of an action (i.e. either the translation or rotation component along with the duration component)?

5.2.3 Student Transplants

Is it possible to transplant a student that is trained for a task in a partially observable static environment into a partially observable dynamic environment? Can it adapt more quickly than if the student were just trained for the same task in the partially observable dynamic environment? That is, is it possible to give the student a "head-start" by training in a partially observable static environment? Would one expect this to be true in human performance?

5.2.4 Real Robots

Experiments with the Tutor-Student learning algorithm on real robots need to be conducted. Will this learning algorithm perform as well in real robots as in simulated robots?

5.2.5 Learning Intervals

An interesting experiment would be to try and determine if there is a decision interval or set of decision intervals where the majority of learning takes place.

Acknowledgment

I would like to thank my supervisor, Dr. Charles Brown, for his encouragement and guidance during my graduate studies. He gave me the freedom to pursue my own research and my graduate experience has been enhanced by his expertise, understanding and patience. As well, I would like to thank Professor Desa Polajnar and Dr. William Owen for their counsel and valuable suggestions on how to improve my research and this thesis.

I would like to give special thanks to my wife Jia Zeng for her support, help and advice. As well I would like to thank my parents for helping me to reach this point.

References

[AC90] P. E. Agre and D. Chapman. What are plans for? In P. Maes, editor, *Designing Autonomous Agents: Theory and Practice from Biology to Engineering and Back*, pages 17–34. The MIT Press: Cambridge, MA, USA, 1990.

[Ark98] Ronald C. Arkin. *Behavior-based Robotics.* MIT Press, 1998.

[Ash64] W. Ross Ashby. *An Introduction to Cybernetics.* Routledge Kegan and Paul, 1964.

[BK96] R. P. Bonasso and D. Kortenkamp. Using a layered control architecture to alleviate planning with incomplete information. In *In Proceedings of the AAAI Spring Symposium 'Planning with Incomplete Information for Robot Problems'*, pages 1–4. Menlo Park, CA: AAAI Press, Stanford, CA, 1996.

[BKMY03] D. S. Blank, D. Kumar, L. Meeden, and H. Yanco. Pyro: A python-based versatile programming environment for teaching robotics. *Journal of Educational Resources in Computing (JERIC)*, 3(4):1531–4278, 2003.

[Bra86] Valentino Braitenberg. *Vehicles: Experiments in Synthetic Psychology.* MIT Press, 1986.

[Bro86] R. A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, RA-2(1):14–23, 1986.

[Bro89] R. A. Brooks. A robot that walks: Emergent behaviors from a carefully evolved network. Technical Report AI MEMO 1091, MIT, 1989.

[Bro91] R. A. Brooks. Intelligence without reason. In John Myopoulos and Ray Reiter, editors, *Proceedings of the 12th International Joint Conference on Artificial Intelligence (IJCAI-91)*, pages 569–595, Sydney, Australia, 1991. Morgan Kaufmann publishers Inc.: San Mateo, CA, USA.

[Con92] J. Connell. Sss: A hybrid architecture applied to robot navigation. In *Proceedings of the 1992 IEEE International Conference on Robotics and Automation (ICRA-92)*, pages 2719–2724, 1992.

[Fau94] L. Fausett. *Fundamentals of Neural Networks: Architectures, Algorithms, and Applications.* Prentice Hall, 1994.

[Gat98] E. Gat. On three-layer architectures. In D. Kortenkamp, R. P. Bonasso, and R. Murphy, editors, *Artificial Intelligence and Mobile Robots: Case Studies of Successfid Robot Systems*, pages 195–210. MIT Press, Cambridge MA, 1998.

[Gos03] A. Gosavi. *Simulation-Based Optimization: Parametric Optimization Techniques and Reinforcement Learning.* Kluwer Academic Publishers, 2003.

[JS02] B. Jung and G. S. Sukhatme. Tracking targets using multiple robots: The effect of environment occlusion. *Autonomous Robots*, 13(3):191–205, 2002.

[Mae94] P. Maes. Modeling adaptive autonomous agents. *Artificial Life, I*, (1&2)(9), 1994.

[MC92] S. Mahadevan and J. Connell. Automatic programming of behavior-based robots using reinforcement learning. *Artificial Intelligence*, 55(2-3):311–365, 1992.

[Mit90] T. Mitchell. Becoming increasingly reactive. In *Proceedings of 8th National Conference on Artificial Intelligence (AAAI-90)*, pages 1051–1058. Morgan Kaufmann, 1990.

[Mit97] T. Mitchell. *Machine Learning.* WCB/McGraw-Hill, 1997.

[MMB93] L. Meeden, G. McGraw, and D. Blank. Emergent control and planning in an autonomous vehicle. In Touretsky D. S., editor, *Proceedings of the Fifteenth Annual Meeting of the Cognitive Science Society*, pages 735–740. Lawerence Erlbaum Associates, Hillsdale, NJ, 1993.

[MWBDW02] A. Makarenko, S. Williams, F. Bourgault, and H. F. Durrant-Whyte. An experiment in integrated exploration. In *In Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 534–539, 2002.

[Neh92] U. Nehmzow. *Experiments in Competence Acquisition for Autonomous Mobile Robots.* PhD thesis, University of Edinburgh, Department of Artificial Intelligence, 1992.

[Neh94] U. Nehmzow. Autonomous acquisition of sensor-motor couplings in robots. Technical Report UMCS-94-11-1, The Univerity of Manchester, Department of Computer Science, 1994.

[Neh95] U. Nehmzow. Flexible control of mobile robots through autonomous competence acquisition. In *Measurement and Control*, volume 28 of *2*, 1995.

[Neh99] U. Nehmzow. *Mobile Robotics: A Practical Introduction.* Springer Verlag, 1999.

[NHS89] U. Nehmzow, J. Hallam, and T. Smithers. Really useful robots. In *Proceedings of Intelligent Autonomous Systems*, volume 2, 1989.

[NSM93] U. Nehmzow, T. Smithers, and B. McGonigle. Increasing behavioural repertoire in a mobile robot. In *From Animals to Animats*, volume 2, pages 291–297. MIT Press, 1993.

[Pir99] P. Pirjanian. Behavior coordination mechanisms - state-of-the-art. Technical Report IRIS-99-375, University of Southern California, 1999.

[SB98] R. Sutton and A. Barto. *Reinforcement Learning: An Introduction.* MIT Press, 1998.

[SRK93a] A. Saffiotti, E. H. Ruspini, and K. Konolige. Blending reactivity and goal-directedness in a fuzzy controller. In *Proc of the 2nd IEEE Intl Conf on Fuzzy Systems*, pages 134–139, San Francisco, CA, 1993.

[SRK93b] A. Saffiotti, E. H. Ruspini, and K. Konolige. Robust execution of robot plans using fuzzy logic. In A. L. Ralescu, editor, *Fuzzy Logic in Artificial Intelligence — Procs. of the IJCAI'93 Workshop*, pages 24–37. LNAI 847, Springer, Berlin, Germany, 1993.

[VSSM02] R. T. Vaughan, K. Stoy, G. Sukhatme, and M. Mataric. Lost: Localization-space trails for robot teams. In *IEEE Transactions on Robotics and Automation*, pages 796–812, 2002.

Appendix A

Standard Q-learning

An algorithm for standard Q-learning is given below [SB98, Gos03].

α = step size

λ = discount value

1. Initialize $Q(state, action)$ to arbitrary values

2. Repeat:

   (a) Get current state i

   (b) Choose action a from i using a policy

   (c) Execute action a, observe received reward r and next state j

   (d) Update $Q(i, a) = (1 - \alpha)Q(i, a) + \alpha[r(i, a, j) + \lambda max_{b \epsilon A(j)} Q(j, b)]$

3. Until: End of the agent's trial run

Appendix B

Standard Q-learning with a Function Approximator

An algorithm for standard Q-learning with a function approximator is given below [Gos03].

α = step size

λ = discount value

```
1. Repeat:
    (a) Get current state $i$
    (b) Choose action $a$ from $i$ using a policy
    (c) Execute action $a$, observe received reward $r$ and next state $j$
    (d) Determine the output $q$ of the *action* neural network for action $a$ in state $i$
    (e) Set $q_{next}$ to be equal to the maximum value of the outputs for the *action* neural
        networks with state $j$ as input to these networks.
    (f) Update $q = (1 - \alpha)q + \alpha[r(i, a, j) + \lambda q_{next}]$
    (g) Update the *action* neural network associated with action $a$ with $q$
2. Until: End of the agent's trial run
```

Appendix C

Incremental Backpropagation Learning Algorithm

The algorithm for the incremental backpropagation learning algorithm is given below [Mit97].

input vector $= \mathbf{x}$

output vector $= \mathbf{o}$

target vector $= \mathbf{t}$

$\mathbf{w}$ = weight

η = learning rate

- The input from neuron i to neuron j is denoted by x_{ij}.
- The weight from neuron i to neuron j is denoted by w_{ij}.

```
For a specified number of iterations
    For all: ( $\mathbf{x}$, $\mathbf{t}$ ) in a set of training examples
        Input the $\mathbf{x}$ to the network. Compute the output of each neuron.
        For all: neurons $k$ in the output layer
            Calculate $output\_neuron_k$'s error $\delta_k = o_k(1 - o_k)(t_k - o_k)$
        For all: neurons $h$ in the hidden layer
            Calculate $hidden\_neuron_h$'s error $\delta_h = o_h(1 - o_k)\Sigma_{k \epsilon outputs} w_{kh} \delta_k$
        For all: Network Weights
        $w_{ji} = w_{ji} + \eta\delta_j x_{ji}$
End For
```