A Quantitative Approach to the Analysis of Memory Requirements for Autonomous Agent Behaviours using Evolutionary Computation

DaeEun Kim

Doctor of Philosophy

Institute of Perception, Action and Behaviour
Division of Informatics
University of Edinburgh

2002

Abstract

When building controllers for autonomous agents, such as real or simulated robots, there is very little theoretical guidance on how complex the controller must be in terms of internal state to be able to achieve good performance in a particular task. The underlying problem is that it is hard to quantify the amount of internal state — memory — the controller should have based on knowledge of the task to be tackled. This problem manifests itself in two related questions: how complex is a given task in terms of the amount of memory a controller needs to enable a particular agent to complete it? and how does this complexity vary depending on the sensorimotor abilities of the agent?

In this thesis I address these questions empirically using evolutionary computation techniques, by attempting to evolve controllers with good performance for a task while simultaneously minimizing the amount of internal memory used. Boolean logic networks with memory elements (registers) allow a natural progression from purely reactive (memoryless) controllers while being straightforward to use with evolutionary computations. Controllers are evolved using three kinds of representational structure — finite state machines, rule-based state machines and tree state machines — equivalent to Boolean logic networks with memory, for which the internal state can easily be quantified. Various evolutionary computational techniques are adapted to support analysis of memory, including evolutionary multiobjective optimisation, a sample selection method, the design of suitable genetic operators and an elitism strategy. Performance evaluation and evolutionary algorithm issues for noisy environments are also considered.

The evolutionary computational approaches are applied to a variety of standard grid world and simple robotic tasks to quantify the memory needed for adequate performance. The results demonstrate, as expected, that internal state plays a significant role in improving performance of agent behaviours when sensors or motor actions are restricted, and is related to the amount of perceptual aliasing present in the interaction between agent and environment. New results concerning the "difficulty" of the standard problems are obtained. The investigation of these problems also demonstrates the utility of the tools developed for quantifying and analysing memory use.

Acknowledgements

There are many people who have helped me, in one way or another, my dissertation work for the doctoral degree. I would like to give my special thanks to my supervisor, John Hallam. He suggested a challenging topic related to memory when I started studying in Edinburgh. He has shown insightful research directions and affectionate consideration. The meetings with him have led to intriguing research topics and they have been happy moments. I am also grateful to my second supervisor, Gillian Hayes, who has shown thoughtful concerns in Forrest Hill.

My colleague Jose Carmena has encouraged me to survive Edinburgh life. The cooperative work with Jose was one of exciting occasions in Edinburgh. I appreciate his help and concern. I would like to thank the mobile group members who shared much pleasant time with me in Forrest Hill - Yuval Marom, George Maistros, Ignasi Cos Aguilera, Paul A. Crook, Alzira Leite, Peter Ottery, Jay Bradley, Alex Champandard, Roger Said, Nuno C. Chagas, Richard Reeve, John Demiris, Mike Spratling, Arturo Espinosa-Romero and Joanna J. Bryson. I give my millions of thanks to Tim Tayler who kindly read and commented all chapters. I am also greatly thankful to Bridget Hallam and Bruno Lara who commented some chapters. I feel grateful to John Berry who provided me with technical support and advice for computing. Also I give my thanks to Sandy Colquhoun and Douglas Howie who helped me robotic experiments as technicians.

I would like to dedicate my thesis to my late grandparents who wished to see my doctoral degree. They showed much love, interest and support for my overseas study.

I give great thanks to my parents. I have been always indebted to them with great benefits, and their constant support for my study cannot be forgotten. Most particularly, I thank my beloved wife, Sun-A Back, for much invaluable encouragement to my study and for her deep love and support.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*DaeEun Kim*)

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis examines effects of memory on behaviors for autonomous mobile agents. The approach is mainly divided into two topics. One is to define the complexity of tasks or behaviors in terms of memory elements. The other is to investigate the relationship between sensor processing and memory. The approach involves searching for the minimalisation of control structures for various tasks.

Behavior-based robotics has been an feasible and attractive approach in mobile robot control systems. It has proved that a reactive coupling between sensor readings and motor actions can be successful in physical world environments. Brooks, who proposed the concept of behavior-based approach, built multiple primitive behaviors to handle many difficult situations and put the mechanism of action selection over those primitives [Brooks, 1986]. Nowadays the design of intelligent systems, especially in robotics, emphasizes the physical interaction with the environment, that is, more direct coupling of perception to action and dynamic interaction with the environment.

However, a general question can be raised, "Will memoryless reactive control systems be superior to memory-based control systems or inferior in mobile robots?" Behavior-based robotics skips planning process often seen in classical AI (Artificial Intelligence) and instead it emphasizes low-level reactive modules for complex tasks. Various complex tasks can be decomposed into subtasks or sequentialized with several subtasks. Sequential states in particular may require memory for control systems and memory elements can serve as connectives of subtasks.

It will be an interesting and novel approach to discriminate robot behaviors from the viewpoint of memory. It has not been studied so far to explain how complex robotic tasks are or how many

subcomponents exist. For example, one binary memory element can be used as a connective of two subtasks. Some researchers have tackled complex tasks with recurrent neural networks or Turing-computable machines [Floreano and Mondada, 1996; Yamauchi and Beer, 1994b; Nordin and Banzhaf, 1997]. It indicates indirectly that memory-based controllers should be applied to complex tasks rather than purely reactive controllers.

One of the objectives of this thesis is to provide quantitative comparisons among memory-based controllers and purely reactive controllers for a variety of tasks. Thus, it will investigate limitations of purely reactive systems or restricted memory systems for some tasks. It will enable us to establish a hierarchy of complex tasks in terms of memory elements.

It seems that memory plays a role in escaping difficult situations where the information of sensor readings is not sufficient to direct motor actions. Memory can logically expand the information of its sensor space by accumulating a history of perceptions. Situations with the same sensory pattern but requiring different motor actions often occur when sensor readings are restricted. It will be a challenging topic to investigate the relationship between sensor space and memory.

1.1 Content and Motivations

The design of control systems is not a trivial problem. Evolutionary algorithms have been a popular approach to build controllers in mobile robots. Inspired by the process of biological evolution, they automatically develops desirable control systems which can be adapted to complex environments [Nolfi and Floreano, 2000; Meyer et al., 1998; Gomi and Griffith, 1996; Harvey et al., 1992]. This kind of evolutionary method in mobile robots normally belongs to the field of evolutionary robotics.

An initial population of random configurations on robot control systems, described in data structures called chromosomes, is tested in the environment. Encoded information on each chromosome says how to process sensor readings and how to move or manipulate motor actions. The performance is measured for each chromosome representation. It corresponds to how well the robot can survive the environment (i.e. its fitness). The fitter robots are given more probability to reproduce themselves and genetic operators such as crossover and mutation are applied to chromosomes selected for breeding. As the evolutionary process continues, the fitness of the robots in the population generally improves. The process continues for a specified number of generations, or alternatively until a robot is obtained with fitness greater

than a predefined threshold level.

In many agent problems, behavior complexity is an important issue and it should be carefully investigated. In some aspects, behavior or task complexity may be represented by the complexity of its control system for the task. However, Simon [1981] suggested the following hypothesis about behavior complexity.

"An ant, viewed as a behaving system, is quite simple. The apparent complexity of its behavior over time is largely a reflection of the complexity of the environment in which it finds itself." [Simon, 1981]

From this viewpoint, the complexity is the result of interaction between agent and environment. The behavior complexity is not a simple measure, but it should be studied with many aspects and criteria together. Memory may be one such complexity criterion. It can play a role to decompose complex behaviors into small primitive behaviors. It is assumed that the number of memory elements required for a given behavior or task will determine the complexity level. Thus, a quantitative approach to memory components will be introduced to describe complexity in the thesis.

In the thesis, Boolean memory elements will be used to discriminate the complexity of agent behaviors, since they are of simple forms for quantification. A different number of register bits[1], flip-flops, will be embedded in memory control structures. Evolutionary computation is applied to search for the best performance with a limited number of memory elements. In the experiments of evolutionary algorithms, the results depend on random components of chromosome selection, crossover and mutation. A single experimental result cannot guarantee the solution and thus many runs of independent experiments are required to have a fitness distribution for a given memory structure. Such probabilistic distributions depending on memory structure will determine what is the best control structure.

In many agent behaviors, experiments have shown that more than the necessary memory elements are spurious. Agent behaviors are not progressed in terms of evolutionary fitness values if the amount of memory is over the essential limit. It is meaningful to find what is the basic memory limit to produce agent behaviors. An efficient evolutionary process with multi-objectives is applied to obtain the minimal memory expressions of control structures. The multi-objectives consist of behavior fitness and the number of memory states. It will easily display what is the optimal number of memory elements required for a given task. The method

[1] More exactly speaking, the number of memory elements is counted.

can relatively decrease the computation time required to take multi-experiments of independent runs for each size of memory elements.

Evolving robot controllers in simulation and then transferring them to real robots are one of the most efficient methods in evolutionary robotics. Simulation program for a small-sized Khepera robot is developed. The simulation is to imitate real sensor readings and motor actions on a Khepera robot by sampling the values. It is an efficient way to map the real situation by sampling senor readings at each different location and angle, and forming a function to interpolate the values. Also noise will lead to different trace of movement. The sensor and motor mapping table will be different even for the same style of Khepera robots.

The relationship between sensor space and memory is explored in the experiments by creating a various number of sensor states. A sensor state is defined as a state of sensory pattern that is mapped to the same motor actions, when a robot recognizes its environment through sensor readings at an instantaneous moment. A set of sensor states can be varied by dropping some of the sensors or allowing multi-thresholds for one sensor. In some cases each sensor value is transformed into a binary value by comparing with one middle point of sensor value. The process turns sensori-motor mapping into Boolean logic circuits. As a result, the entire control system can be represented as Boolean logic circuits with flip-flops.

1.2 Questions and Goals

This thesis presents an experimental study for the complexity issues of autonomous agent behaviors in terms of memory elements. The goal of this research is to explore the possibility and limitation of controllers in purely reactive systems, and study the importance of dynamic elements in reactive controllers. The dynamic component will be represented as a memory element in Boolean logic networks in mobile robots. Thus, the work will start with showing the difference between a memoryless pure reactive controller and a memory-based reactive controller, and explaining the memory effect in robot behaviors.

The following research questions will be answered in the thesis :

1. Is it possible to achieve many mobile robot behaviors by using Boolean logic networks with/without dynamic elements? What are the limitations of Boolean logic networks as robot controllers?

2. What kind of behaviors will it be hard to achieve without memory elements? If memory

is required, how many memory elements should be added to the controller?

3. What is the relationship between complex sensor readings and memory? Will robots need more memories if they have simple sensors of binary values or a reduced number of sensors?

4. What is the efficient method to develop robust controllers in the noisy world? How do we compare the performance of controllers?

The above questions are ultimately related to how to design minimal desirable controllers for mobile robots or agent tasks, more precisely, to handling and analyzing each behavior based on memory elements, which will determine what sort of behaviors need memory as well as how much amount of memory is required.

In the thesis, Boolean logic networks with dynamic elements are represented as state machines. Boolean logic networks as purely reactive controllers are applied to agent tasks and then compared with state machines. In some robotic tasks such as obstacle avoidance, exploration and box pushing bebavior, Boolean logic networks with discretized sensors are sufficient to evolve desirable behaviors. For more complex agent behaviors, a small amount of memory significantly improves the performance of behaviors. Especially the perceptual aliasing problem [Whitehead and Ballad, 1991] requires internal memory for agents.

The second issue is related to the quantification of memory amount required to achieve a desired behavior. Three kinds of state machines to quantify internal states are suggested: finite state machines, rule-based state machines and tree state machines. Behavior complexity is estimated with the method of evolutionary multi-objective optimization where the best behavior performance is obtained for each level of memory amount. Thus, it can quantify memory amount for the expected behavior, based on state machines. It will be demonstrated in the grid world problems and robotic tasks. Yet there still exist complex behaviors, e.g. homing navigation, which are difficult to achieve with state machines. It indirectly indicates there is a hierarchy of behavior complexity beyond state machines.

The third issue handles the relationship between memory and sensors. More sensor space, more action space and more memory have a potential of better performance and the three criteria build a lattice structure for performance. When sensors or motor actions are restricted, agents may require more memory to achieve desirable behaviors, since the necessary memory amount is influenced by perceptual aliasing situations. It will be demonstrated with the woods problems and robotic tasks.

The last issue involves an evolutionary technique and performance evaluation. A variation of subset selection method [Gathercole, 1998; Lee, 1998] is used to design robust controllers efficiently in noisy environments, especially emphasizing elitism. This method is useful to the problems requiring many starting positions or configurations in robotic tasks. The effort test [Lee, 1998] is extended to compare the performance of evolved controllers in terms of probabilistic distribution.

The experiments in the thesis consist of two kinds of problem sets, grid world problems and robotic tasks. Grid world problems are provided with simplified sensors and motor actions in a restricted environment, but imitate real-world agent tasks. Robotic tasks are simulating robotic behaviors in the real world and we will assume a Khepera-style robot [Mondada et al., 1993].

1.2.1 Grid Problems

Grid problems have been studied for agent behaviors in the field of artificial life [Koza, 1992; Jefferson and Collins, 1991; Teller, 1994; Ashlock, 1998; Wilson, 1994]. They have one or more agents located on a grid environment and the range of sensors is discretized. The environment is different from the real world, but it is often used as a simple model of reality. The restricted model of a robot agent imitates robotic tasks and also we can see how an agent develops various agent behaviors in many situations.

Three grid world problems are studied in the thesis; the "artificial ant" problem [Koza, 1992; Jefferson and Collins, 1991], the "Tartarus" problem [Teller, 1994; Ashlock, 1998] and the "woods" problem [Wilson, 1994; Lanzi, 2000]. Those problems are tested to see the effect of memory (Boolean memory) on agent behaviors.

The artificial ant problem is a problem imitating an ant's exploration and foraging behavior. In the problem, the task of an agent is to explore its environment and collect as much food as possible in the shortest time. The food is irregularly spread over the grid environment and a time limit is given. The agent has one sensor in front and four possible actions; move forward, turn left, turn right and think. Thus, how to design controllers of the ant agent is not an easy problem.

The Tartarus problem is a simple version of a box pushing task. An agent is supposed to push several boxes to the walls and, if possible, to the corners. The agent can detect boxes and walls with eight sensors around it. The agent is scored according to how many boxes are pushed to the walls. There are many possible board configurations in this problem. It is a difficult

problem to construct successful control systems to satisfy all board configurations.

The woods problem is an exploration problem where an agent searches for the food in the environment with woods. The task is to find the best path to the food position, starting at any random position.

The above three problems are requiring memory elements in order to obtain near-optimal performance. A memoryless approach in control design definitely has its limitation on these problems. Thus, they will be good examples to see what is the role of memory elements in controllers. The study of memory effects for these problems will be discussed in detail in the thesis.

1.2.2 Robotic Tasks

In this thesis, robotic experiments will handle various robotic tasks from primitive behaviors to moderately complex behaviors; obstacle avoidance, wall following, exploration, corridor following, box pushing, T-maze decision and so on.

In evolutionary robotics, simulation is an important tool for evolving robot controllers. It can save much time and efforts required to test the real world experiments. Instead of directly evolving controllers in the physical environment, experiments in the thesis will use simulation programs of a Khepera robot. For some tasks, the evolved controllers will be transferred to the real robot to test the real world environment. Robots experience much noise in sensor readings in the real world. A Khepera robot simulation program assumes random noise in sensor readings.

It is assumed that sensor states and memory are correlated. Simpler sensor space seems to lead to more memory states. Thus, a robotic task will be repeated with different sensor configurations and different environments. The simplest sensor states can be constructed by binarizing each sensor with a threshold.

Obstacle avoidance behavior is a primitive behavior for a robot to escape any obstacle or object without any collision, but not to stagger around a position. Exploration behavior is checking the ability of a robot to explore the whole environment. Wall following behavior is a behavior that a robot follows walls in the environments. Corridor following is similar to wall following, but a robot is placed in the environment with slightly more difficult situations of narrow corridors. Box pushing behavior is a behavior for a robot to push a box to a target position. The target can be marked with a light or walls. T-maze learning behavior is a behavior for a robot to

remember the state of lights while following a corridor and finally take an action of moving left or moving right at the end of corridor.

1.3 Summary of Methods

1.3.1 Boolean logic circuits with memory

In evolutionary robotics, feature extraction from sensory patterns has been emphasized as an important tool in reactive architectures to map input patterns to appropriate control outputs [Nolfi and Floreano, 2000]. Neural networks are one of the popular approaches to extract features from sensory patterns. Recurrent neural networks have advantages to keep internal memory as well as extract features. Recurrent neural networks have feedback loops among nodes and they are recording and accumulating previous node activations. When converting recurrent neural networks into finite state machines, recurrent neural networks include many memory states [Omlin and Giles, 1996]. The continuous activation values of nodes with feedback loops are clustered into memory states. However, it is hard to see evolving the structure with a small number of memory states or imposing restrictions to reduce the amount of memory on the recurrent neural networks.

An alternative control structure which can quantify memory is a combinational logic circuit, acting on sensor values converted to binary by thresholding or comparison. Lee [1998] evolved fairly complex reactive behaviors for a real robot using such a combinatorial logic. Such an approach offers the possibility of adding memory in an easily quantified way. In the thesis, three methods to construct Boolean logic memory are suggested.

One method to use a finite state machine is that the threshold of each sensor is pre-determined and so each sensor has a binary value. It starts with the logical result from threshold setting over each sensor so the characteristics of sensor readings are preserved in the threshold for perception or pattern recognition. After pre-processing sensor readings, Boolean logic circuits are evolved with a genetic algorithm. In the evolutionary approach, a model of finite state machine, which is equivalent to sequential logic circuits [Kohavi, 1970], is applied to count the number of memory states, instead of using the explicit form of logic circuits.

Another method to encode finite states is a rule-based state machine. Each chromosome has a fixed number of rules with preconditions, actions and the next state to be triggered. It assumes that sensor thresholds are pre-defined and sensor values are Boolean logic values. Precondi-

tions of each rule are of binary format, and the best matching rule to satisfy the preconditions will be fired to generate motor actions. Such rule-based state machines will be useful to handle robots with many sensors.

The third method is a tree state machine with decision trees. The decision trees have a sensor number and its threshold in the internal tree node and defines motor actions and the next memory state in the leaf node. Evolving together motor actions and thresholds for perception will help reflect significant thresholds of the real sensor readings. Each memory state has its own decision tree. The decision tree is evolved with thresholds of sensor readings, motor actions and the next memory state to be triggered.

1.3.2 Evolutionary algorithms with memory-based controllers

Many kinds of memory-based systems including recurrent neural networks have been developed in evolutionary robotics. To order to quantify memory, the method encoding finite states is introduced. After pre-processing sensor readings, motor actions are assigned to a memory state and a sensor state. In case that sensor readings are binarized, the control system becomes a finite state machine.

When a complex task requires several memory bits or memory states, it is meaningful to see what is the minimal number of memory states to achieve the complex task. Normally many states over the essential memory states are redundant in control systems, since they do not help improve the performance but increase the size of the control system. The minimal number of memory states required to achieve a complex task can determine the complexity of complex tasks. Independent evolutionary experiments are tested with various number of states and compared with the use of performance evaluation. Also an efficient evolutionary algorithm with multi-objectives is suggested to test concurrently controllers with various number of states.

1.3.3 Performance measure

In evolutionary computation, there exist a variety of evolutionary parameters such as crossover and mutate rate, selection mechanism, population size and so on. In most evolutionary experiments, the choice of parameters tends to depend on tasks. They are determined case by case. It is suggested that fitness distribution and effort test [Lee, 1998] should be used to determine desirable evolutionary parameters with minimal computational cost. The fitness distribution

test is looking at the distribution of fitness curves with the same number of generations [Goldberg, 1989]. It will check what conditions will be helpful in leading to an efficient performance within the same computation time. The effort test is analyzing how early an evolution mechanism can reach a given performance and how much effort is needed to reach a pre-defined performance [Lee, 1998]. These two tests will help decide desirable control structures or strategies as well as evolutionary parameters. In the thesis, both fitness distribution test and an extended version of effort test will also be applied to memory-based systems and purely reactive control systems.

1.4 Summary of thesis

The approach applied in this thesis follows the scheme of evolutionary robotics.

Chapter 2 provides an overview of evolutionary robotics. It first introduces evolutionary algorithms including genetic algorithm and genetic programming. It shows how researchers in the evolutionary robotics have developed various control mechanisms and as well what kind of complex robot behaviors have been handled.

Chapter 3 describes the methodology of the main approach in this thesis. It covers various memory control structures and the corresponding evolutionary algorithms. An efficient Pareto-optimization method is suggested to find minimal memory expressions of control structures with desirable fitness. Performance evaluations over fitness distributions are also explained to determine parameters and strategies.

Chapter 4 handles artificial ant problems that require memory-based systems. Experiments first search for desirable evolutionary parameters that need the least computation time. They are determined by two performance evaluation methods, fitness distribution test and effort test. Experiments also demonstrate how to determine the minimal number of memory elements to solve each grid world problem. The experiments use two schemes for statistical evaluation of fitness distribution. One is to apply a fixed number of memory states for a single experimental run and repeat independent runs of experiments. The other is to apply the multi-objective concept to evolutionary computation and test various memory states in a single run.

Chapter 5 discusses grid world problems including the Tartarus problem and woods problem. In the Tartarus problem, experiments involve how to handle random environmental configurations and again how to choose desirable evolutionary parameters. Finite state machines and tree state machines are applied to the problem. Experiments use an evolutionary multiobjec-

tive optimization scheme for memory analysis. Using the method, several woods problems are investigated to see the required memory amount with a given agent configuration. It shows the relationship between memory and the number of sensors.

Chapter 6 shows behaviors of mobile robots with simple robot models. It uses whisker-like binary sensors and robot kinematics equations. Experiments compare purely reactive controllers and memory-based controllers. Several robotic tasks are tested with fitness distribution, including obstacle avoidance, wall following, exploration, corridor following and box pushing. The control structures for encoding memory are rule-based state machines. The complexity of robot behaviors will be analyzed with memory concept.

Chapter 7 handles the experimental issues with realistic robot simulation. It first describes realistic sensor models for a Khepera robot and then discusses how to select sample positions and evaluate them in noisy environments. Several schemes are introduced to support robustness and elitism in evolutionary computation. Wall following behavior is tested with those evolutionary methods in noisy worlds.

Chapter 8 shows the simulation results of Khepera robots. They investigate several robotic tasks, wall following, corridor following, box pushing and T-maze decision. The complexity of robot behaviors is evaluated in terms of memory amount. The control structures for encoding memory are state machines including finite state machines and rule-based machines. Purely reactive control systems with Boolean circuits, feedforward neural networks and the suggested state machines will be compared and discussed. The relationship between sensor space, action space and memory amount will be discussed together.

Chapter 9 has a conclusion and discussion.

Chapter 2

Literature Review

Evolutionary robotics is the application field of artificial life to robotics. It handles various topics from robot control design to robot morphology. The field has a view of designing life-like robots and autonomous agents instead of passive robots requiring human hand-coding. As a result, it supports the idea of automatic design of robot controllers. In this chapter, an overview of evolutionary robotics is presented. It will generally cover what issues have been handled by researchers and what kind of control structures have been used.

The behavior-based approach is closely related to the progress of evolutionary robotics. The development and concept of behavior-based robotics is presented as an introduction (section 2.1). Then evolutionary algorithms, one of the main tools in evolutionary robotics, are explained (section 2.2). Various evolutionary approaches in robotics have been studied and many research groups have investigated the design problems and control structures. The research and their methodologies are explained (section 2.3). Evolutionary robotics often confronts problems with figuring out how to design controllers in simulation and reality. The next section presents the investigation of how to evolve control systems in simulation and reality (section 2.4). The main topic that is addressed in this thesis is related to memory-based control structures. The applications of memory-based control systems are described in the last section (section 2.5).

Section 2.5 will be the key section related to the work in the thesis, but the earlier sections are provided to give a solid background in the underlying technique.

2.1 Behavior-based Robotics

The classical AI approach [Haugeland, 1985; Pfeifer and Scheier, 1999] has several important characteristics, such as the use of knowledge to involve explicit symbolic representation about the world environment and the ability to represent a hierarchical structure by abstraction and symbolic processing. The idea means that knowledge representation and knowledge processing are central to intelligence. Behavior-based robotics systems started against these traditions. Classical robotics focused on representation and planning issues while assuming that the low-level details of reliable interaction with a realistic environment would be straightforwardly resolvable in due course. This assumption proved to be difficult, and the realisation of that fact led to the widespread switch to behavior-based robotics. The notion of sensing and acting within the environment began to be a main topic in AI-related robotics research rather than the previous focus on knowledge representation and planning. Since then, robotics has been more concerned with perception and action than with classical artificial intelligence methods [Pfeifer and Scheier, 1999].

Nowadays the design of intelligent systems, especially in robotics, emphasizes the physical interaction with the environment, that is, more direct coupling of perception to action and dynamic interaction with the environment. A behavior is often viewed as a reaction to a stimulus in behavior-based robotics. Similarly, reactive control in robotics is a technique for tightly coupling perception and action, typically in terms of sensori-motor behaviors, to produce timely robotic responses in dynamic and unstructured environments. This concept avoids the use of explicit abstract representational knowledge in the generation of response. Brooks' subsumption architecture [Brooks, 1986] is the origin of behavior-based robotics, where a robot control system is built layer by layer. Each layer is responsible for a robot behavior with the method linking senors to motors while interactions between layers are restricted to inhibition, suppression or message passing.

Brooks argued that the sense-plan-act paradigm used in some of the first autonomous robots such as Shakey [Nilsson, 1969] was detrimental to the construction of real working robots. He also asserted that building world models and reasoning using explicit symbolic representational knowledge prevented timely robotic response and actually led robotics researchers in the wrong direction.

Mobile robots need to be able to generate efficient behavior under complex environments. A main idea is to draw an emergent behavior over the interaction of the robot with its dynamic environment rather than to design robot controllers depending on human knowledge and skills.

The design of even reactive primitive modules or sensori-motor mapping may not be simply planned or constructed in many cases.

Evolutionary robotics is an artificial life application of behavior-based robots. The development of a control structure using evolutionary computation can provide emergent behavioral control strategies in dynamically changing environments [Cliff et al., 1992; Mataric and Cliff, 1995; Gomi and Griffith, 1996; Meyer et al., 1998; Nolfi and Floreano, 2000]. In evolutionary robotics, evolutionary algorithms modeled after biological evolution are used to develop robot control systems. They start by encoding a robot control system as genotypes and generating a random population of controllers. Then some evaluation task or behavior is tested over controllers in their environment, and only controllers adaptable to a given environment will survive during the evolutionary process.

2.2 Evolutionary Computation

There are several different styles of evolutionary computation approaches such as genetic algorithms [Holland, 1975], genetic programming [Koza, 1992], evolutionary strategies [Back and Schwefel, 1993], and evolutionary programming [Fogel et al., 1966]. All of them gain inspiration from biological evolution.

Popular methods among evolutionary computations recently applied in mobile robots are genetic algorithms and genetic programming. Evolutionary programming is known as the first method to follow an evolution model; evolutionary strategies are a variation of genetic algorithm. We will first see what is the difference in their operators and their computation styles.

2.2.1 Evolutionary Programming

Fogel developed evolutionary programming in his studies [Fogel et al., 1966; Fogel, 1995]. Intelligent behavior was described as prediction of the environment and a proper sequence of action-responses to a given goal. The environment was modeled as a sequence of alphabet symbols. He used finite state machines with internal states to predict the next symbol from a given observed symbol, and used the input-output symbol pairs and state transitions for each input symbol, over every state, to specify the behavior of any finite state machine given a starting state. As a result, a finite state machine produces a sequence of output symbols for a sequence of input symbols.

Offspring are generated by randomly mutating each parent finite state machine. Each parent produces a single offspring. Fogel used five types of random mutation: changing an output symbol, changing a state transition, adding a state, deleting a state, and changing the initial state. Mutations are selected according to a probability distribution.

In his experiment, the finite state machine structures within the evolutionary program were tested for simple two-symbol cyclic sequences, more complex eight symbol cyclic sequences degraded by random noise, sequences of symbols generated by other finite state machines, and nonstationary sequences [Fogel et al., 1966]. Fogel's approach showed the possibility of using finite state machines to encode evolutionary structures, but they can also be seen as a kind of time series prediction. It is not closely related to robotics but the underlying idea is quite relevant.

2.2.2 Genetic Algorithm

An evolutionary approach called genetic algorithms (GAs) has been popular among engineers and scientists, and much research related to genetic algorithms has been done [Mitchell, 1996; Goldberg, 1989]. Holland, an inventor of genetic algorithms, showed that genetic algorithms can play the role of an adaptive system through reproduction and evolution [Holland, 1975]. GAs maintain a fixed number of populations with its cost or fitness evaluation and apply genetic operators such as crossover and mutation to the members of the population to produce a new generation. Each member in the population is represented as a bit string called a chromosome, and a chromosome consists of a string of genes, each of which is a genetic feature of the individual chromosome. The chromosomes for reproduction are chosen based on their fitness in the environment. Thus, the chromosomes with higher fitness have a higher probability of producing new offspring while the chromosomes of low fitness may disappear. Genetic operators are of great help in preserving and broadcasting the good characteristics of high fitness genes.

The theoretical foundation as a stochastic approach is based on the schema theorem [Holland, 1975; Goldberg, 1989]. The dynamics of the chromosome in a genetic algorithm may now be given by the following sequence:

1. Initialize the population, typically with a random set of chromosomes.
2. Calculate the fitness(cost) of each chromosome/member in the population using the fitness function.

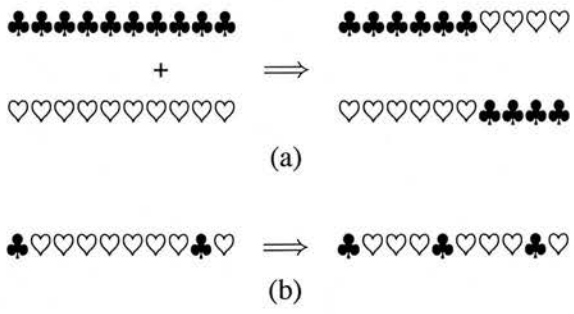


Figure 2.1: An example of crossover and mutation operators in genetic algorithm (a) crossover at position 6 (b) mutation at position 5

3. Order the population by fitness.

4. Generate the next population by the process of selecting high-fitness chromosomes of the current population and applying genetic operators, crossover and mutation.

5. A generation is completed. Go to step 2 and repeat the above process until the desired performance is obtained or the time limit is exceeded.

The crossover operator as one of genetic operators is a process of recombination of genes by exchanging segments between pairs of chromosomes with crossover point selected randomly as in Figure 2.1(a). This operator plays a role in collecting the characteristics of good genes and guiding the search for areas of our concern. The mutation operator randomly replaces one allele of a gene with another to yield a new type of gene as shown in Figure 2.1(b). This compensates for the weak points of the crossover operator which only handles already existing genes and cannot produce a new type of gene.

The hope of the GAs is that reproduction using genetic operators may allow the characteristics of good solutions, i.e. good genes, to combine in new and better ways. A chromosome can be not only represented with bit strings, but also with a string of real numbers. Especially neural networks have real number encodings for chromosomes [Ijspeert, 1998; Beer and Gallagher, 1992]. Also there is a structured chromosome GAs such as the SAGA (Species Adaptation Genetic Algorithms) [Harvey, 1992a,b] which use variable length chromosomes.

Usually a genetic algorithm maintains a chromosome representation structure and manipulates a population of gene strings. Its genetic search may be viewed often and processed through the space of strings. As a result, a solution to a specific problem can be handled easily in the search space of the string expression. We can see the changing population of gene strings as

a search mechanism to find a solution in multi-modal, nonlinear spaces through reproduction, crossover, and mutation operators. Thus, in a genetic algorithm, a high-quality solution is found by applying a set of biologically inspired operators to individual points within a search space. This yields better generations of solutions over time. The fitness for each member of the population of robot controllers measures how well each individual is adaptable or executable with respect to the chosen tasks.

2.2.3 Genetic Programming

Another evolutionary approach is genetic programming (GP) designed by Koza [1992]. Genetic programming is a variation of genetic algorithms with the difference in representations and genetic operators. It uses tree structures of dynamic size rather than fixed length strings. The chromosomes are represented as a tree. Unlike genetic algorithms which use bit strings, genetic programming manipulates higher-level primitive constructs such as S-expressions in Lisp programs. The property of abstract representation can reduce the problem search space, thus increasing the performance of the evolutionary process. The tree expression can be seen as a computer program.

The procedure of running genetic programming is similar to that of genetic algorithms, but genetic operators are applied to tree expressions, and functions and terminals should be defined to solve problems. The function and terminal sets in a tree are important components of genetic programming. The function set can be composed of various functions including mathematical functions, such as addition, subtraction, multiplication, division, and other user-defined functions. The terminal set consists of variables and constants of a program. The terminal set is located at the leaves of trees in genetic programming, while the functions are positioned at the internal nodes of trees to execute an appropriate function over subtrees.

The crossover operator in GP swaps subtrees of two parent chromosomes as in Figure 2.2(a) where the crossover point can be specified at an arbitrary branch. Mutation deletes a subtree and creates a new random subtree as in Figure 2.2(b). These genetic operators have the effect of generating new S-expressions to be tested for fitness while keeping parts of good expressions.

One advantage of genetic programming is that it can allow variable-length coding and utilize computer programs. It can easily encode the characteristics of computer programs, such as conditional execution, computation on variables and constants, and further, iterations and recursions. It has more representation power than bit strings of fixed length in a conventional genetic algorithm.

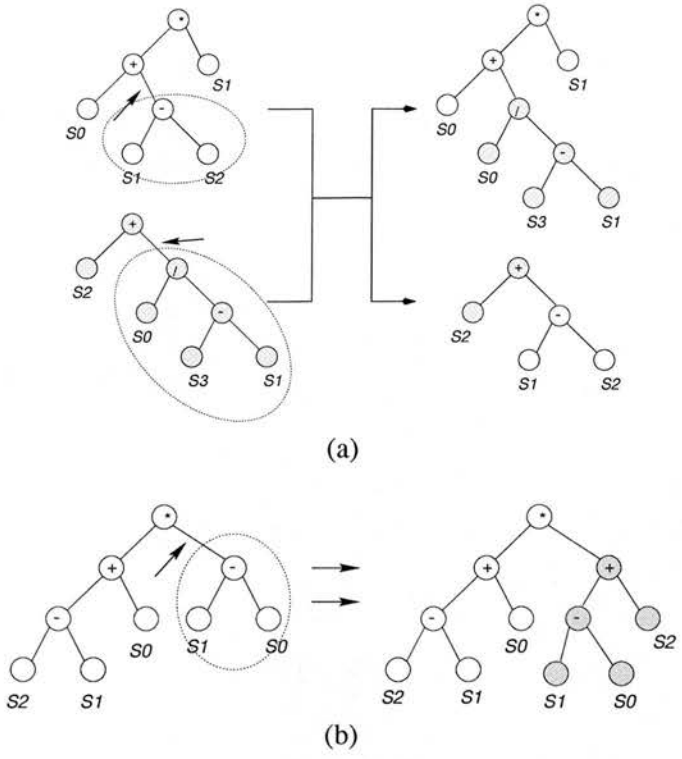


Figure 2.2: An example of crossover and mutation operators in genetic programming (a) crossover operator (b) mutation operator

2.3 Evolutionary Robotics

One of the ultimate goals in evolutionary robotics is to provide a methodology to create complex behaviors automatically rather than by hand design and to find more efficient and emergent control structures for such behaviors.

The behavior-based approach is a popular approach in robotics. It emphasizes reactive modules to handle sensori-motor mapping for a biological stimulus-response relationship. However, behavior-based robotics still have the problem of how to decompose a complex behavior into primitives and also how to design an arbitration over those primitives. According to conventional behavior-based robotics approaches, simple behaviors are wired into a robot and debugged before adding the next behavior. Then, each new behavior is again wired into the robot by hand design [Brooks, 1986; Harvey et al., 1996; Arkin, 1998]. Evolutionary robotics has been suggested as an alternative to develop control structures automatically in behavior-based robotics [Lee et al., 1997; Lee, 1998]. Evolutionary robotics incrementally and emergently evolve complex robot control systems instead of attempting to figure out each behavior step by hand design.

Many research groups have worked on evolutionary robotics recently and each group has shown its own special characteristics in its approach and experiments. We will discuss those different robotics experiments and methods in this section, and it will help us understand what behaviors can be tested and obtained with restricted sensors in a given environment.

2.3.1 Classifier Systems

The evolutionary approach for general control systems was first tested by Goldberg in a gas pipeline application. [Goldberg, 1983, 1989]. He used classifier systems which consist of binary rule-based systems without a variable state. The success of the genetic algorithm was first demonstrated with classifier systems in control problems.

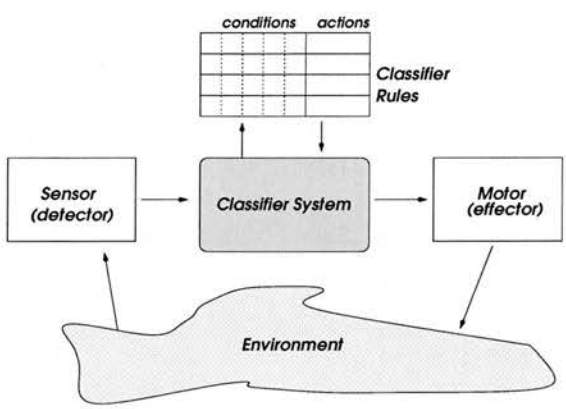


Figure 2.3: Diagram of classifier systems

In classifier systems shown in Figure 2.3, genetic operators are used on a set of rules encoded by bit strings. The basic structure of a classifier system is a rule-based system where preconditions for a set of rules are examined to see their applicability given the current situation. The preconditions have fixed-length bit encoding with values of { 0, 1, #(don't care) }. The action part of the rule is also a fixed-length encoding with the values of { 0, 1 }.

Classifier systems apply reinforcement learning [Holland, 1975; Sutton and Barto, 1998] as well as evolutionary learning. The credit assignment problem, as one of main problems in reinforcement learning, is the problem of apportioning credit or penalty for overall outcomes of the system to each of the internal components or subsystems. It normally depends on their contributions to overall outcomes. We have difficulty in determining precisely which component has changed the overall performance in complex environments and also how much it has contributed.

contributed.

Classifier systems employ a bucket-brigade algorithm for the credit assignment problem in reinforcement learning [Holland, 1975; Goldberg, 1989]. To implement the algorithm, each rule called a classifier is assigned a quantity called its strength. The strength value is adjusted to reflect the usefulness of the rule in the environment. The algorithm gives rewards to all active rules at the time when the reward is received. Rewards are also given to the classifier rules that contributed to the activation of the rules. Thus, the bucket-brigade algorithm handles the credit assignment problem for each active rule, evaluating the results of the chosen actions, while a genetic algorithm creates new rules using genetic operators.

Colombetti and Dorigo [1993] has used a classifier system on a real robot to demonstrate phototaxis, learning to approach both stationary and moving light sources, to escape a predator and find a food resource, and to follow a moving light hidden behind wall [Colombetti and Dorigo, 1993]. A hierarchical architecture of classifiers was built for desired behavior sets and coordinating classifiers. After generating control results in simulations, they transferred them into a real robot. They developed behavior analysis and training (BAT) methodology to design and build learning autonomous robots [Dorigo and Colombetti, 1998].

Another approach combining the power of production rules and genetic algorithms is the SAMUEL system developed at Naval Research Lab, USA [Grefenstette and Schultz, 1994]. It was tested on a Nomad 200 robot. The task for the robot was to learn how to avoid obstacles and to safely navigate to a specific goal region in a cluttered environment within s predetermined time using 52 sensory inputs, tactile, sonar and infrared sensors. When transferred to the physical robot it performed with an 86 % success rate.

As defined above, classifier systems are based on production rules of binary strings and have difficulty in managing continuous values. Another popular controller is a neural network controller which can, by contrast, map continuous input spaces into continuous output spaces.

2.3.2 Sussex University Group

Recently evolutionary robotics research has concentrated on using proximal sensing devices such as infra-red sensors, mechanosensory whiskers and bumpers, or active ranging devices including ultrasound sensors. In fact, sophisticated navigation requires visual sensors, but conventional vision processing needs extensive computation power. It is difficult to incorporate vision techniques into evolutionary approaches using current technology [Mataric and Cliff,

1995; Meyer et al., 1998; Harvey et al., 1994].

Only a few groups have tested evolutionary robotics including vision. Sussex group first studied the evolution of a visually guided robot, using a neural network architecture and simulation model involving ray-tracing computer graphics before it was tested on a real robot [Cliff et al., 1992]. They used two photoreceptors, each of which had a 4×4 square cross-section of its receptive field. The robot was evolved to find the center of a circular room without explicit reference to visual processing. The photoreceptors did not handle the variation and distribution of each pixel value and the sensor input to the network was an averaged intensity, so it is hard to say it is evolved for visual processing.

Harvey et al. [1994] developed a gantry robot allowing for 50 Hz frame rates from a 64×64 monochrome CCD camera to off-board computers. They used the evolutionary approach over neural network controllers and visual morphology of receptive fields to produce visually guided behaviors in the real world. The sensory input was taken as an estimate of the average image intensity of the pixels within a receptive field.

Their evaluation task was to make controllers capable of an elementary visual discrimination between a rectangle-shaped target and triangle-shaped target. They first evolved the robot controller to move towards a large white rectangular target in a dark visual environment. Then towards a much smaller white rectangular target, and then finally to move towards a small white triangle instead of a small white rectangle. When they observed the active part of the evolved neural network, the visual morphology of the network using two receptive fields proved sufficient for shape discrimination between the oriented edges of the triangular target and the vertical edges of the rectangular target.

As an extension of the above study, Smith [1997] also has used an evolutionary approach for a vision robot. His football playing Khepera robot controller with a one dimensional CCD camera was evolved in simulation to find the ball and push it into the goal and then transferred into the real world successively. The controller was a recurrent neural network with 16 visual inputs and 16 hidden units.

2.3.3 Mondada, Floreano and Nolfi Approach

Floreano and Mondada [1994] applied genetic algorithms to a real robot in real time instead of using simulation. Evolutionary computation determined weights and thresholds in a recurrent network for obstacle avoidance. They used an evaluation fitness function with three

components: the forward speed of the robot, the speed of rotation during a given time slot, and sensor activation. This fitness function has been often used among many evolutionary robotics experiments for obstacle avoidance.

They also tested some complex behaviors using a neural controller scheme, including a homing behavior where a Khepera robot locates a battery charger and periodically returns to it [Floreano and Mondada, 1996].

Nolfi evolved a garbage collecting behavior as well as target approach behavior [Nolfi and Parisi, 1995; Nolfi, 1997a,b; Nolfi and Floreano, 2000]. The robot has a grip sensor to detect if it is holding an object. It also has infrared sensors on the left and right sides, and motor outputs for two wheels and a trigger output for object pickup and release. His target approach behavior involves classifying objects by distinguishing small cylindrical objects, large cylindrical objects and walls. The activation level for each of the eight IR sensors was recorded for many different orientations and different distances to reflect a real environment model in simulation. The size and shape of Khepera, the physical environment, and the robot position were accurately modeled in the simulation. Nolfi first succeeded in classifying objects with a simple feedforward neural network from input node to output node without any hidden nodes. Then he tested a garbage collecting behavior with emergent modular architectures. Garbage collecting is a non-trivial behavior consisting of several primitive behaviors: exploring the environment avoiding the walls, recognizing the target object, placing the robot body to pick up the target, picking up the target, moving toward the wall, placing the body near the wall, and releasing the object. Even in this case, he has succeeded with his special modular architectures without any memory neurons.

2.3.4 Case Western Reserve University

Beer and Gallagher [1992] also used neural networks in an evolutionary approach. The evolution of recurrent neural networks was investigated for adaptive agent control with chemotaxis and locomotion control for a six-legged robot where the neurons are based on the continuous Hopfield network model [Hopfield, 1982] with full connections between neurons. The network consisted of sensory neurons for chemical signals, motor outputs, and interneurons. Later they showed experiments with a neural network controller for locomotion in a real 6-legged robot where each leg is controlled with a 5-unit Hopfield-style network [Gallagher et al., 1996].

Yamauchi and Beer [1994a] demonstrated the successful evolution of continuous-time recurrent neural networks for landmark recognition and navigation. They first tested the landmark

a = a3+4;	b = d – d;	e = d + a4;	d = d * 10;	c = d | 7;
d= a2 >> a1;	a = c – 3;	c= b & c;	b = e >> 6;	motor1 = c * 9;
b=a1 – d;	a = c4 << a;	d = d + d;	d = b & 0 ;	c = e & e;
d=a2 + 2;	d = a* 4;	c=d | 8 ;	e = c >> b;	
c= d >> 1;	d = a– 5;	c = d | 8 ;	motor2 = a | e;	

Table 2.1: Assembly instructions developed for obstacle avoidance by Nordin and Banzhaf [1995]

recognition using temporal sequences of sonar sensor readings on a Nomad 200 robot and then explored navigation behavior to reach a goal within an environment of landmarks. After simulation, they transferred the evolved controllers to a real robot. The results showed that dynamic neural networks could be evolved to integrate temporal sensory perceptions in decision making while the robot turned around the landmark object, and then extended to take control actions to reach a goal based on the decision. However, their landmark navigation was only tested with a simple environment; goal, agent, and two landmarks positioned in serial.

Before the previous example of sequence learning, Yamauchi [1993] applied the same neural network structure to a Nomad 200 robot for both predator escape behavior and obstacle avoidance behavior together.

2.3.5 Genetic Programming Approach

Koza [1992] used genetic programming over simulation of robots for wall following and box pushing behaviors. The robot was assumed to have 12 sonar sensors, a bump sensor, and a set of several sensor primitives and action primitives, but the experiment was not representative of a noisy reality in many respects including sensor positions, even though they made a good controller for the given environment.

Reynolds [1994] has also tested genetic programming for corridor following. His experimental environment was simplified in sensor and motor properties, so we cannot assert that the developed controller can work well in real environment.

Nordin and Banzhaf [1996] have used genetic programming in a real Khepera robot for the task of obstacle avoidance and object following. They applied genetic operators to assembly language level instructions handling register operations. They used real-time learning with a physical robot using genetic programming. To overcome the long learning time needed they devised a second method, involving learning from past experiences stored in memory, which made a speed-up to the algorithm [Nordin and Banzhaf, 1997]

Lee and his group evolved controllers with circuit networks based on genetic programming for complex behaviors, and used a hybrid genetic algorithm/genetic programming approach to co-evolve controllers and morphologies [Lee et al., 1996, 1997]. They have also shown that a complex behavior can be decomposed and evolved in its composite parts with genetic programming where its primitives and arbitrators can form a complex behavior without any memory element [Lee et al., 1996]. The controllers evolved in simulation were transferred into a real robot. They succeeded in pushing a box to a specific goal marked with light using this method.

2.3.6 Legged Locomotion Robots

Most evolutionary robotics experiments have worked mainly with two-wheeled robots, such as the Khepera and Nomad 200. In the future, legged robots have much potential to be explored and studied in a complex environment.

Lewis et al. [1994] used genetic algorithm methods to evolve the weights for a neural controller for a robotic hexapod rather than using a traditional neural learning method such as back-propagation. Fitness functions were defined for first learning oscillatory controls for each of the legs and then coordinating the oscillations to produce effective gaits.

Galt et al. [1997] evolved the optimal gait parameters for a 8-legged walking robot. To control gaits, the desirable phase and duty cycles for leg movements were searched. Gruau and Quartramaran used a cellular encoding method for an 8-legged OCT-1 robot [Gruau, 1995; Gruau and Quartramaran, 1997]. The method encodes a grammar tree to control the division of cells which grow into recurrent neural networks based on a developmental model.

2.3.7 Other Evolutionary Robotics Experiments

Ram et al. [1994] applied an evolutionary approach to find a desirable set of parameters for reactive behaviors such as obstacle avoidance and moving to a goal for navigation. Miglino and his group succeeded in evolving a robot controller in a Khepera robot for obstacle avoidance using a two-layer feedforward neural network without a hidden layer [Miglino et al., 1995]. They first used a simulation and transferred the best controller to a real robot. Mayley's work was to evolve a feedforward neural controller for wall following in a Khepera [Mayley, 1996]. Some weights were defined in gene encodings as plastic and updated by Hebbian learning.

Some research has been done on evolving neural controllers with developmental models to grow a neural structure. Development is an integral part of biological evolution. Genetic changes are not directly reflected in phenotype changes. A complex developmental process is involved between genetic representation and phenotype information. The developmental model follows the natural mechanism of evolutionary systems and provides a compact chromosome representation. It has the effect of building a complex structure incrementally and encouraging modular design of control systems.

Gruau [1995] used a cellular developmental process for modular neural networks. Instead of explicit representations for the construction of the robot's controller morphology, implicit specifications of the robot architecture was considered. The development of neural networks from genotype to phenotype has the form of genetic programming in the approach. Eggenberger [1996] evolved a neural controller with the developmental method to avoid obstacles, and seek or avoid light. Michel [1996] used a developmental method for wall following and obstacle avoidance.

Meeden [1996] applied a recurrent neural network to four-wheeled mobile robots to control a forward/backward movement and steering. The task was to keep moving, avoid wall obstacles, and seek or avoid light. The results showed the evolutionary weight update gives better performance than the backpropagation learning method in this situation.

Baluja [1996] evolved a neural network controller for an autonomous land vehicle. He used a 15×16 pixel array for vision input and a 3 layer feedforward neural network for desirable steering. His experimental results showed the evolutionary computation outperforming backpropagation method on average.

Salomon [1996] suggested an efficient evolutionary strategy which encoded each parameter as floating-point numbers and applied mutation to all parameters at the same time by moving a self-adapted step size for each parameter. He developed a Braitenberg-style neural network which can produce obstacle avoidance behavior on a Khepera robot. In his experiment, it took a few hours to evolve control networks in the physical robot.

2.3.8 Other Behavior-based Robots

Most evolutionary robotics experiments so far have used simple sensors. This is due to the fact that evolutionary computation requires much time in finding desirable controllers. A good utilization of sensor information is often seen in associative learning or self-organizing mapping

in behavior-based robotics. In that case, it can be seen as many memory elements are involved. For example, each neuron unit of a self-organizing network plays the role of a memory unit. Some applications of self-organizing networks have given good hints on how to achieve difficult task processing.

A research group at the University of Edinburgh has studied ways to increase a robot's behavioral repertoire through a connectionist associative memory [Nehmzow and Smithers, 1991; Nehmzow and McGonigle, 1994]. They built an internal state space representation to develop navigational competence in a mobile robot using a self-organizing network. Distinct physical locations in the real world are memorized and recalled with feature recognition techniques. This method successfully provides robust and flexible behavior

Lambrinos [1995] has tested homing navigation behavior similar to that seen in many insects which use a light compass. His Khepera mobile robot was equipped with ambient light sensors and infrared sensors. The robot was able to learn to associate light sensor inputs with proper motor actions. For his obstacle avoidance behavior, he used a Braitenberg-style network with infrared proximity sensors. A Kohonen self-organizing network was used to form an appropriate connectivity pattern between sensor inputs and motor outputs and thus create an adaptive light compass. Scheier and Lambrinos [1995] applied the self-organizing map to a garbage collecting behavior which has components for avoiding obstacles, classifying objects, grasping objects and taking them to the nest.

2.4 Methodology

The above research reviews are presented with research groups or the evolutionary approaches and applications. In evolutionary robotics, an important issue is *how* to evolve controllers. Currently the most popular method is to evolve neural controllers in simulation and transfer them to the physical environment. A precise simulation of a real world environment is a very difficult problem. Generally, when the controllers are transferred to the real world, their performance degrades.

Simulated computation can be executed at even faster speeds than a real-time evolutionary computation method. Simulation transfer controllers can save much time, but have a potential to be significantly degraded in a physical environment if the simulation environment does not reflect the reality very well.

We can divide experiments in evolutionary robotics into three cases: evolving the controller

in simulation only; evolving the controller in simulation and testing in the real world; and evolving the controller in a physical robot. Research in each of the three cases is described below. For some special cases, we can see evolution of the hardware for robot controller and of the morphology and controller together.

2.4.1 Evolving Neural Network Controller in Simulation Only

In this case, the robots have been studied in simulation, but people are still exploring the designing of a real robot.

Beer and Gallagher [1992] tested robot morphology and control actions in simulation before their real hexapod robot was built. Cliff et al. [1992] tested visually guided robots in simulation before designing their gantry robot. Many robotics experiments are in progress using simulation. Simulation can save much of the time needed to find desirable controllers and can be used to test the plausibility of control structures and robot morphology to prepare for real design.

Ijspeert, while still at Edinburgh University, tested a neural network model for motor control of the lamprey and salamander [Ijspeert et al., 1997]. He used a genetic algorithm to find desirable parameters over a continuous-time dynamic recurrent neural network model with 3 state neurons. He developed a good animal robot model using a central pattern generator.

2.4.2 Evolving in Simulation and Testing in a Physical Robot

When a simulation shows a reasonable degree of accuracy to the real robot and environment, the best control structures developed over many generations can be transferred to the actual robot and expected to work.

Jakobi et al. [1995] studied the effects of simulation noise level in his experiments. According to his results, when the noise levels in simulation are significantly different from the real noise in the real robot, the transfer from the simulated controller to real world is not effective. He also found that the simulation will not produce a good performance if the complexity and interactions of robots with the environment increase. Jakobi [1998] provided some guidelines for constructing simulations that would probably allow successful transfer of evolved behavior to reality.

Miglino et al. [1995] used the method transferring the best simulation controller to the physical world. They applied statistical analysis of real sensor noise to the simulation and saw that more

realistic noise levels can provide more feasible solutions, reducing the gap between simulation and reality. They also used the technique of capturing a model of the actual sensor and motor behavior of the particular robot in the simulator, thereby making transfer easier.

There have been many other applications to transfer the best controller in a simulation to real robots [Cliff et al., 1992; Nolfi, 1997b; Lee, 1998]. This method is currently the most feasible approach in evolutionary robotics, while considering computation time and real solutions together.

2.4.3 Evolving Controllers in a Physical Robot

Some robotics research groups have attempted to design robust controllers by evolving control structures in the real world [Floreano and Mondada, 1994, 1996; Nordin and Banzhaf, 1996]. The real-robot evolutionary robotics experiments take a long time to find a stable desirable controller, but their controllers reflect reality well. There is a tradeoff between time and control effects. Thus, the method of evolving controllers directly in physical robots has been tested only with simple behaviors such as obstacle avoidance and object following.

2.4.4 Evolving Hardware for a Robot Controller

Evolvable hardware is currently a hot research topic. Direct coupling between hardware and environment not only gives a fast response time to events in the environment but also allows direct evolution of robot control hardware without a software program. The following is one example of evolvable hardware experiments.

Thompson [1995] experimented with evolving reconfigurable electronic circuits to produce control hardware for autonomous mobile robots. For control hardware, he used Field Programmable Gate Arrays (FPGA) which can encode a variety of digital logic functions. He first demonstrated by evolving a millisecond oscillator circuit from nanosecond logic gates, which generated output spikes at about 4 kHz. Then he showed wall avoidance behavior using an electronic circuit controller and two sonar inputs on the left and right sides. His physical semiconductor circuits were based on dynamic state machines whose temporal dynamics followed the physical characteristics of the hardware circuit elements. Thus, the behavior of the dynamic state machine involved a stochastic component, resulting from the asynchronous sonar inputs and asynchronous state transitions, even though it directly drove the motor outputs according to the raw echo signal input and the current state.

2.4.5 Evolving Morphology and Controller in Simulation

A group at Sussex University developed visually guided robots using a recurrent neural network for the task of room centering [Cliff et al., 1992]. They used a vision simulation model based on a real robot, and explored the possibility of evolving together visual sensors and control neural networks for navigation. The genotypes of variable-length chromosome described position and acceptance angles of each of the two photoreceptors and also the weights and connectivity of the control networks.

Sims [1994] developed a novel approach to evolving robotic creatures using genetic algorithms. Sims' work has been tested only in simulation. The simulations allows for the evolution of not only the agent's controller, but also its morphology. Genotypes encoded as directed graphs are used to produce phenotype structures for three-dimensional kinematic systems. He succeeded in developing several evolved creatures for swimming and walking.

Lee et al. [1996] showed that it was possible to co-evolve robot controllers and robot body plans. For instance, the robot body plan in a mobile robot may include sensor types and positions, body size, wheel radius, and motor time constants. The robot body as well as the robot controller can influence the behaviors of the robot, and the evolutionary approach is developed for the evolution of both controllers and bodies together for specific tasks.

2.5 Memory-based Controllers

This section is directly related to the main approach of this thesis. There have been much research done to investigate the importance of memory in complex problems. The approaches have emphasized memory encoding, but have neglected quantitative comparisons among memory-based systems and reactive systems. Also there have been no methods so far to measure the amount of memory needed to achieve a task.

Reactive systems in mobile robots are often seen as a direct coupling between perception and action without any intermediate processing. But in fact many real robot systems following a behavior-based approach actually use several memory elements to handle the control problem correctly and efficiently. Even Brooks worked on subsumption architecture with memory elements [Brooks, 1986]. Memory is a medium for temporal information processing in reactive systems. However, there has been little research about the effect of memory in behavior-based robotics.

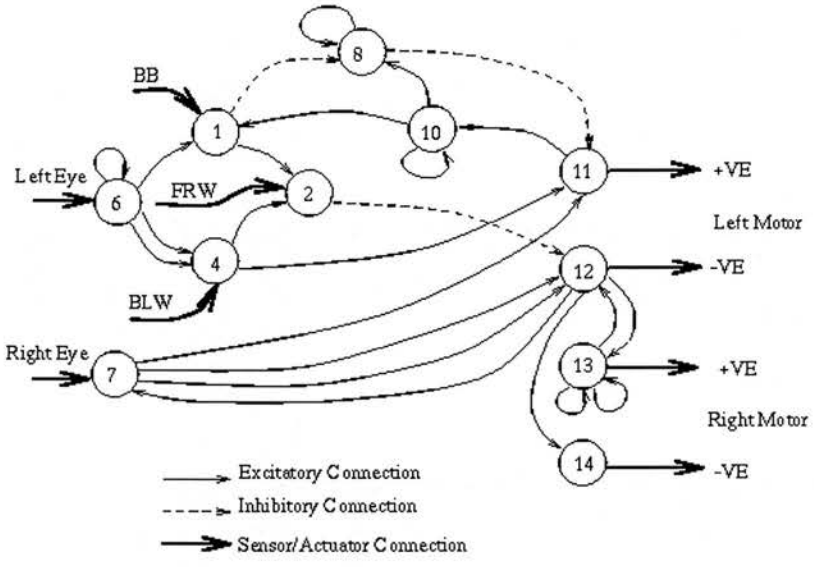


Figure 2.4: An example of a recurrent neural network (Reprinted from [Cliff et al., 1992])

2.5.1 Neural Networks in Evolutionary Robotics

Many evolutionary robotics approaches have been tested using recurrent neural networks to generate complex behaviors as mentioned in the above sections. A recurrent neural network has at least one feedback loop among its nodes unlike a feedforward neural network; a neuron feeds its output signal back to inputs of other neurons. The presence of feedback connections assumes a delay of the output activations of neurons. Also repeated looping of activation through the recurrent connections can accumulate activations and maintain memory. Figure 2.4 is an example developed for a visually guided robot by a group at Sussex University [Cliff et al., 1992].

Recurrent neural networks as robot controllers have been largely developed using genetic algorithms and proved to be very useful and effective, because the genetic algorithms provide a way of 'training' a recurrent network without assuming a pre-specified output trajectory. Their neural network encodings actually use continuous functions, but the interactions between neurons are unpredictable. They may often represent the unstable internal states of the continuous systems which makes it hard to analyze theoretically.

In some robotic tasks perceptual information alone is insufficient to uniquely determine the appropriate action, for example, when robots go to the fourth nominally identical door in a corridor. It may be necessary to integrate the partial perceptual cues with previous actions or per-

ceptual information to decide proper actions. Many tasks, such as landmark-based navigation and sequential behaviors, require that an agent also be able to integrate perceptual information over time in order to determine the appropriate course of action [Yamauchi and Beer, 1994a,b]. In evolutionary robotics, this ability is usually provided by evolving continuous-time recurrent neural networks which, for instance, can be evolved to generate a fixed sequence of outputs in response to an external trigger occurring at varying intervals of time [Yamauchi and Beer, 1994b].

2.5.2 Boolean Logic Networks in Evolutionary Robotics

Some evolutionary robotics experiments have shown that purely reactive systems without memory can implement various primitive behaviors: for example, object avoidance and exploration behaviors, and box-pushing to a target signalled by an environmental marker [Lee, 1998; Kim and Hallam, 2001]. These needed only a purely reactive controller with binary sensory inputs. More complex combinations of such behaviors can be realized using stateless switchers to arbitrate between the primitives. Rather than using neural networks, Lee et al. [1997] achieve these results by evolving a Boolean logic network using genetic programming (GP).

In Lee's approach for robotics using Boolean logic networks [Lee, 1998; Lee et al., 1997], each behavior is a purely reactive controller with no internal state. Its control structure is a combinational logic circuit, evolved using GP, acting on analog sensor values converted to binary by thresholding or comparison (the thresholds and comparison functions are also determined by GP evolution). Such an approach offers the possibility of adding memory in an easily quantified way.

Kube and Zhang [1993] showed that for a social behavior of box pushing, the arbitrator of action selection can be implemented for cooperative behavior by a Boolean logic network without any internal memory. The task was to locate and push a heavy box with a group of robots. The box pushing required the net force of at least two robots in the same direction and thus a successful group of robots had to cooperate to push a box.

Nilsson [1994]'s teleo-reactive (T-R) program suggests a circuitry of Boolean logic in agent control. The T-R program is an agent control program to direct the agent to a goal by considering changing environmental situations. It consists of production rules in priority which each rule has a condition and an action:

$$K_1 \rightarrow a_1$$

$$K_2 \rightarrow a_2$$
$$\ldots$$
$$K_m \rightarrow a_m$$

K_i is a condition for sensory inputs and a_i is the corresponding action. These condition-action rules assume continuous evaluation of sensory inputs. He argued that the *continuous* execution is distinguishable from the conventional production systems and the T-R sequence can be implemented by circuitry. Figure 2.5 shows a neural network implementation of a T-R sequence program [Nilsson, 1994] where K_i is a linear threshold function of sensory features. Thus, the conditions are represented as conjunctions of sensory features. The neural network diagram is similar to Brooks' subsumption architecture in which a high priority module can suppress lower level modules. However, the construction of circuitry shows the potential of Boolean logic networks in mobile robot control.

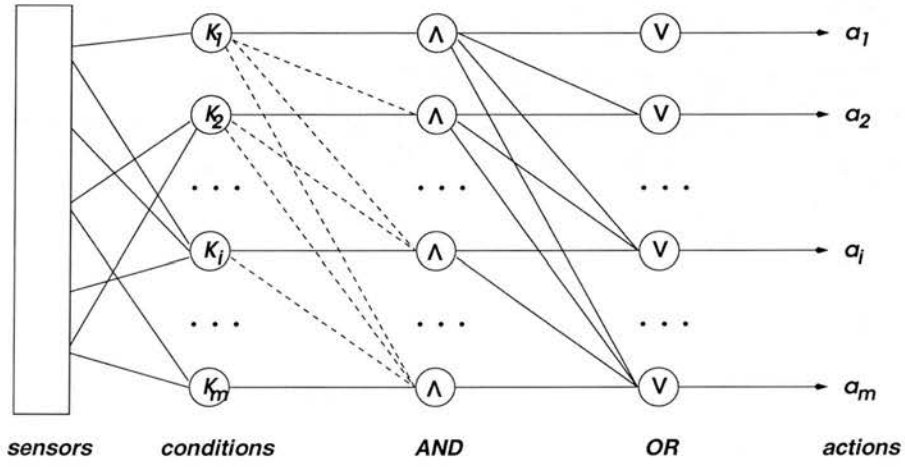


Figure 2.5: A teleo-reactive program to implement a neural network (dotted line: inhibition) [Nilsson, 1994]

Kauffman [1993] modeled a genetic regulatory system using a Boolean logic network to represent the coupling between the various active or inactive genes. He observed that Boolean networks are a very powerful idealization which we can use to consider a very broad class of continuous nonlinear systems. Switching systems are good idealizations of many nonlinear systems [Kauffman, 1993; Hopfield, 1982].

A Boolean logic network with internal memory is equivalent to finite state machines [Kohavi, 1970]. Finite state machines are very useful for describing aggregations and sequences of behaviors [Arkin, 1998]: they can represent the interaction of behaviors, the behaviors active at

any given time and the transitions between them. They have been used to accomplish various high-level goals with complex behavioral control systems consisting of a set of primitive behaviors [Arkin and MacKenzie, 1994; Gat and Dorais, 1994]. Finite state automata provide a good mechanism for expressing the relationships between various behavior sets and have been widely used within robotics to implement control systems [Arkin, 1998]. Brooks [1986] also used a variation of finite state automata—augmented finite state machines—to implement behaviors within his subsumption architecture.

2.5.3 Memory in Grid World Problems

Especially in grid world problems, the importance of memory has been studied by many researchers. Grid world problems have restricted environments and their agents have simple value sensors. They are normally noise-free worlds and the tasks demonstrate the limitations of memoryless policies.

Wilson used a zeroth-level classifier system (ZCS) for his animat experiments [Wilson, 1994]. The original formulation of ZCS has no memory mechanisms, because the input-output mappings from ZCS are purely reactive, but Wilson suggested how internal temporary memory registers could be added. Adding an internal memory register consisting of a few binary bits can increase the number of possible actions in the system. Following Wilson's proposal, one-bit and two-bit memory registers were added to ZCS in Woods7 environments by Cliff and Ross [Cliff and Ross, 1995]. They argued that ZCSM (ZCS with internal memory) would manipulate and exploit the internal states appropriately and efficiently in non-Markovian environments. Their experiments used simplified models and were quite different from real robotic experiments. But, they showed the potential and importance of internal states for robot behaviors. Bakker and de Jong [2000] proposed a means for counting the number of states required to perform a particular task in an environment. Such state counts extracted from finite state machines can be a measure of the complexity of agents and environments. They applied their methodology to a fixed simple simulated environment such as the Woods7 environment [Wilson, 1994] instead of real robotic environments.

There have been other approaches to encode state information in evolutionary computation. A representation for sequential actions is to use 'progn' function in genetic programming, which has been tested in various applications [Koza, 1992]. The 'progn' function executes a series of actions and then returns the result of the last statement. An example of 'progn' functions is shown in Figure 2.6.

```
(if-food-ahead  (move)
                (progn3  (left)
                         (progn2  (if-food-ahead (move)
                                                 (right))
                                  (progn2 (right)
                                          (progn2 (left) (right))))
                         (progn2  (if-food-ahead (move)
                                                 (left))
                                  (move))))
```

Figure 2.6: An example of progn function (this is developed for Santa Fe trail problem by Koza [1992])

```
(if-then-else  (Less (Read 3) (Read 17))
               (Write 3 (ADD (Read 3) (Read 1)))
               (Write 1 (Read 5)))
```

Figure 2.7: An example of indexed memory (if (memory[3] $<$ memory[17]), then add memory[1] to memory[3]; otherwise store memory[5] to memory[1]) [Teller, 1994]

Another representation for state information is to use indexed memory [Teller, 1994]. Indexed memory is an array of registers which can store and retrieve values. Genetic programming employs 'read' function to retrieve a register value and 'write' function to store a temporary result at a certain memory location. In this representation, it needs a set of programs to trigger a particular memory element to write a temporary result at the appropriate time, and also a set of programs to trigger and read a specified memory at the proper time. An example of the indexed memory program is displayed in Figure 2.7.

The concept of multiple interacting programs has been developed to represent state information [Angeline, 1998]. It evolves, instead of a single program, several symbolic expressions of dynamical systems to refer to the output of each symbolic equation. Thus, the output of an equation may be an input for all other equations. Angeline [1998] argued that his method is more efficient than Teller's indexed memory approach. A set of symbolic equations corresponds to a set of state programs. A program is assigned for a state, and each symbolic equation can have a term to refer to each of the others. This set of programs is similar to recurrent neural networks in that the output of one equation can be connected to the inputs of the other equations. Figure 2.8 is a program example for the Santa Fe trail problem which has one sensor and four output actions [Angeline, 1998].

The incorporation of state information permits the evolved control structure to behave better

A_0 = A_2 · NoFood · A_2
A_1 = A_2 + 1.434651 / NoFood
A_2 = 0.852099 / NoFood
A_3 = A_2 / (-0.198859 + A_2 + 0.200719) · (-0.696450(A_1 - A_3) / A_3)

Figure 2.8: An example of multiple interacting programs (NoFood is a sensor and $A_0, ..., A_3$ are the actions for no operation, right, left and move, respectively) [Angeline, 1998]

start: 2 → 8			
	if even	if odd	deciders
0	2 → 9	0 → 9	(ITE x_5 (Com x_3) ($\neg x_5$))
1	4 → 5	2 → 3	(ITE x_5 (Com x_3) ($\neg x_5$))
2	2 → 6	3 → 9	(ITE x_2 $x_3 x_7$)
3	2 → 2	1 → 2	(ITE x_6 x_3 (Odd x_4))
4	2 → 5	0 → 0	(ITE (Odd 0) x_7 ($\neg x_7$))
5	0 → 6	0 → 0	(< -1 (ITE 0 x_3 0))
6	2 → 0	3 → 8	(- (< x_3 x_3) ($\neg x_4$))
7	1 → 4	0 → 8	(Com (> ($\neg x_8$) x_5))
8	1 → 0	2 → 5	(ITE x_7 (<> x_5 x_7) (Odd x_1))
9	1 → 3	2 → 0	(ITE x_5 (Odd -1) ($\neg x_1$))

Figure 2.9: An example of GP automata [Ashlock, 1997]

using past information than a pure reaction to the current sensor inputs. Finite state machines have been used in evolutionary computation to incorporate state information [Fogel et al., 1966; Ashlock et al., 1995; Miller, 1989]. Finite state machines should have their transitions from discretized sensor readings. It is believed that clustering or feature extraction over the sensor space will improve the ability of finite state machines.

GP-automata was proposed to extract features with genetic programming as well as to maintain the structure of finite state machines for state information [Ashlock, 1997]. GP-automata is a set of states, each of which is associated with a transition function, an action, and a parse tree. The transition function has a mapping from the current state to the next state, and the response action will be the output of a controller. Both transition function and response action will be invoked by the parse tree result. This structure has the advantages of both finite state machines and genetic programming search. It, however, still has a limitation of sensor states, even though it can easily tackle many sensors using genetic programming. An example of a GP-automaton is given in Figure 2.9. The notation '2 → 8' means 'output 2 and go to state 8'. Each state has its own decider and environment information around an agent is processed with the decider. The decider result determines what will be the next action and the next state, depending on

whether the decider result is even or odd.

2.5.4 Hidden States in Reinforcement Learning

McCallum [1996] defined a hidden state as any world state information not provided by the current immediate perception of a mobile agent. The hidden state problem in robotics research is related to reactive systems [Brooks, 1986, 1987; Kaelbling, 1986]. Purely reactive systems take the current perception as their only internal state space to be linked with motor actions. Some robotics research have emphasized purely reactive agents [Maes and Brooks, 1990; Mahadevan and Connell, 1991; Chapman and Kaelbling, 1991; Sutton, 1991]. Considering interactions among environments and agents, the control outputs only depend on the current perception in reactive systems.

In some non-Markovian[1] environments, purely reactive control could not succeed in solving hidden state problems [Whitehead, 1992; Singh et al., 1994; McCallum, 1993]. Hidden states often appear when sensors have a limited range of view about the surrounding environment or when there are a limited number of sensors. Such hidden state problems are often called perceptual aliasing problems [Whitehead and Ballad, 1991]. When an agent has only partial information about the surrounding environment through its sensory inputs, but the environmental situation requires different actions, the agent suffers from the perceptual aliasing problem. An alternative to this problem is to find actions leading to the situation where the agent has no ambiguous sensory pattern [Nolfi and Floreano, 2000]. However, this strategy is not a fundamental solution to the perceptual aliasing problem and it is effective only when the agent can find at least one sensory pattern with no aliasing problem. Some reactive robots experience hidden states and their control systems are not truly purely reactive [Brooks, 1991].

Many control systems extend the internal state space with a combination of the current perception and memory of past perceptions and actions as shown in Figure 2.10; classical linear systems with time differential components need this kind of control structure [Kuo, 1991]. A PID controller can be represented as $C(t) = K_1E(t) + K_2 \int E(t)dt + K_3\frac{dE(t)}{dt}$ where $C(t)$ is a control output and $E(t)$ is a control input. For its discretized system, $C_n = W_1C_{n-1} + W_2E_n + W_3E_{n-1} + W_4E_{n-2}$.

McCallum [1996] argued that internal memory should be added to solve perceptual aliasing problems in non-Markovian environments. He developed a reinforcement learning algorithm

[1]The environmental features are not immediately observable, but they have dependencies upon past states.

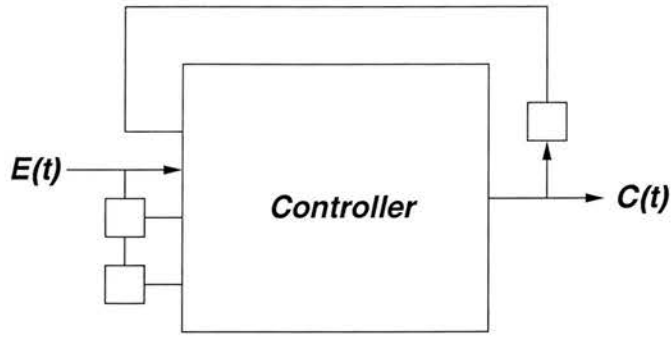


Figure 2.10: A diagram of a control system (small squares are delay elements)

incorporating memory to prevent hidden states. When an agent suffers from hidden states, the perceptions cannot directly define the agent's internal state space. Its decision will depend on the current perception and internal memory about past perceptions and actions. Memory plays a role to disambiguate aliased perceptions.

Colombetti and Dorigo [1994] developed ALECSYS, a classifier system to learn proper sequences of subtasks by maintaining an internal state and transition signals which prompts an agent to switch from one subtask to another. The size of memory states and how to use internal memory are defined at design time in their approach. In contrast, many classifier systems with internal memory define the amount of memory at the beginning and the system should learn how to use internal memory. Similar to the ZCSM experiments [Wilson, 1994], Lanzi [2000] has shown that internal memory is effective with adaptive agents and reinforcement learning, when perceptions are aliased. Stolzmann [2000] used anticipatory classifier systems where previous sensor values are remembered and used to disambiguate hidden states. There has also been research using a finite-size window of the current and past observations and actions [McCallum, 1996; Lin and Mitchell, 1992].

2.6 Summary

In this chapter, the background of evolutionary robotics is described. It first starts with behavior-based robotics and their concept. Then three kinds of evolutionary algorithms are explained. Genetic algorithms and genetic programming will be the main tools in the thesis. The issues in evolutionary robotics and the various approaches are described. The research groups and their methodologies are shown with a history of the research. The design of controllers is divided into three categories; evolving controllers in simulation only, evolving controllers and

then transferring them into a physical robot, and evolving controllers in physical robots. In the thesis, controllers evolved in simulation will be demonstrated.

As one of the main themes, the study of memory-based controllers is explained in detail in this chapter. One of the representative approaches in memory-based systems is using recurrent neural networks. They have proved powerful in many application fields. Also as a simple controller, Boolean logic networks have been considered. When memory is added to Boolean logic networks, it is equivalent to finite state machines. A variation of classifier systems, ZCSM, has been applied to grid world problems to show the effect of internal memory. Genetic programming with indexed memory or with sequential function is one of several popular approaches that have a memory structure. Some researchers have used memory as a tool for solving hidden state problems and designed a reinforcement algorithm maintaining internal memory.

In the thesis, Pareto optimization is used to consider two constraints, behavior performance and memory amount, with evolutionary algorithms. It will find robot controllers with the best performance for each level of memory amount, and will determine the minimal memory structure for the best performance. The evolutionary techniques assume that the interesting properties of the task can be encoded into a fitness function that evaluates candidate solutions; for robotic problems, this is non-trivial. Those issues will be discussed in detail in the next chapter.

Chapter 3

Methodology

In this chapter, various issues of methodologies for evolutionary experiments are introduced. The approaches for controller design, fitness function, performance measurement and multi-objective methods will also be described. They are the main tools used to investigate the performance of various memory structures and to analyze many experimental results.

The first section handles how to construct memory-based controllers and their genome representations. The proposed memory-based controllers, finite state machines, rule-based state machines and tree state machines, have state-based structures where it is easy to quantify the memory amount required for a given task. The application of each structure will depend on agent tasks. Also the issue of how to define fitness functions is described. In the following section, the measure of probabilistic distribution for fitness in evolutionary computation is given to estimate the behavior performance. The evolutionary algorithms have characteristics of stochastic and random distribution. They can be handled with sampling theory over fitness results. Deciding desirable evolutionary parameters and efficient control strategies is an interesting problem in the evolutionary approaches. They can be determined through the suggested performance measures.

One of the main questions in the thesis is how many memory elements are required for a given robotic task. It also involves the question of what is the best performance for given memory elements. A population of variable state machines is considered in evolutionary algorithms and a novel evolutionary method will search for genomes with the best behavior fitness and minimal amount of memory. Through a Pareto optimization technique, the best solutions will be introduced for each given number of memory elements. Thus, the concurrent evolutionary search for the minimal but optimal control system will decide the complexity of control systems

required to achieve robotic tasks.

3.1 Structure of the robot controller

Evolutionary robotic approaches frequently use recurrent neural networks as controllers for complex behaviors [Meyer et al., 1998; Harvey et al., 1992; Floreano and Mondada, 1996], combined with genetic algorithms to determine the network structure or parameterization. Such networks make effective controllers, offering the possibility for creating many possible dynamical systems with a variety of attractors, but it is hard to analyze them theoretically and to quantify the amount of memory a particular network provides to an agent.

We suggest that finite state machines have more possibilities for analysis of memory by their discrete expressions even though they are less powerful in representation than recurrent neural networks. One of the research goals is to explain the relation between memory effect and complexity of behaviors, and to demonstrate this relationship by evolving controllers with a small number of memory elements instead of pure reactive systems.

3.1.1 Memoryless approach

A *memoryless* reactive approach has a controller mapping functions from sensation to motor actions. The set of actions depend only on the current sensation. Purely reactive systems define one mapping function without any internal memory. If a feedforward neural network is constructed for the mapping, it may be a good solution for the reactive system, by the fact that it can approximate any continuous function with an arbitrary number of hidden nodes [Hornik et al., 1989].

There have been many memoryless reactive approaches in evolutionary computation and reinforcement learning [Kaelbling, 1986; Nolfi, 1997b; Lee, 1998].

3.1.2 Finite State Machines

A finite state machine can be considered as a type of Mealy machine model [Kohavi, 1970; Hopcroft and Ullman, 1979], so it is defined as $M = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$ where q_0 is an initial state, Q is a finite set of states, Σ is a finite set of input values, Δ is a set of multi-valued output values, δ is a state transition function from $Q \times \Sigma$ to Q, and λ is a mapping from $Q \times \Sigma$ to Δ,

where $\lambda(q,a)$ will have one of the output set Δ. $\delta(q,a)$ is defined as the next state for each state q and input value a, and the output action of machine M for the input sequence $a_1,a_2,a_3,...,a_n$ is $\lambda(q_0,a_1),\lambda(q_1,a_2),\lambda(q_2,a_3),...,\lambda(q_{n-1},a_n)$, where $q_0,q_1,q_2,...,q_n$ is the sequence of states such that $\delta(q_k,a_{k+1}) = q_{k+1}$ for $k = 0,..,n-1$.

There are several ways to represent finite state machines. Table 3.1(a) is an example developed for wall following behavior. It has four-bit output, where the first two bits are for the left wheel motor and the other two bits are for the right wheel motor. The output for motor control is divided into four levels, fast backward(00), slow backward(01), slow forward(10), and fast forward(11). The chromosome representation follows the content of the table in sequence as follows:

01011 01011 11000 11000 01001 10000 11110 11000

If we increase the number of levels for the output, it is necessary to increase the number of bits for the expression. The schemata expression of genes in Table 3.1(b) is a multivalued integer string instead of a binary string. This scheme reduces the length of genes, thus increasing the convergence speed to optimal points. The gene coding is defined here as a sequence of the pair (state number, state output) of each sensor value in canonical order of state number. For example, the gene coding in Table 3.1(b) used for exploration task is represented as:

275 050 047 335 243 350 160 001 337 326 073 306 376 267 306 003

Each building block corresponds to $(q_x,a_0,q_y,a_1,q_z,a_2,q_w,a_3)$ for each current state q_i, where $q_x = \delta(q_i,00),..,\ q_w = \delta(q_i,11),\ \lambda(q_i,00) = a_0,..,\ \lambda(q_i,11) = a_3$ with the above Mealy machine notation. With the integer representation of chromosomes, one crossover point is allowed only among integer loci, and the crossover rate is applied to each individual chromosome, while the mutation rate is applied to every single locus in the chromosome, instead of each building block or the whole individual chromosome. For example, if the mutation rate is 0.1, then one over 10 integers will experience the mutation and the mutation will change one integer value into a new random integer value. The integer encoding does not only greatly reduce the gene coding size, but also it will help find the desirable finite states efficiently because the crossover point is restricted by the boundary of the state numbers and output values. When binary strings allow the crossover in the middle of bits for state numbers or output values, the crossover point folding area causes a mutation effect on the bits and changes the state number instead of keeping the old value. From the finite state machine, Boolean logic circuits with flip/flops can be extracted. Figure 3.1 is the Boolean logic circuit with one memory element,

state	input 00	input 01	input 10	input 11
q_0	$q_0, L = 10, R = 11$	$q_0, L = 10, R = 11$	$q_1, L = 10, R = 00$	$q_1, L = 10, R = 00$
q_1	$q_0, L = 10, R = 01$	$q_1, L = 00, R = 00$	$q_1, L = 11, R = 10$	$q_0, L = 10, R = 00$

(a)

state	input 00	input 01	input 10	input 11
q_0	$q_2, L = 7, R = 5$	$q_0, L = 5, R = 0$	$q_0, L = 4, R = 7$	$q_3, L = 3, R = 5$
q_1	$q_2, L = 4, R = 3$	$q_3, L = 5, R = 0$	$q_1, L = 6, R = 0$	$q_0, L = 0, R = 1$
q_2	$q_3, L = 3, R = 7$	$q_3, L = 2, R = 6$	$q_0, L = 7, R = 3$	$q_3, L = 0, R = 6$
q_3	$q_3, L = 7, R = 6$	$q_2, L = 6, R = 7$	$q_3, L = 0, R = 6$	$q_0, L = 0, R = 3$

(b)

Table 3.1: Examples of finite state machine

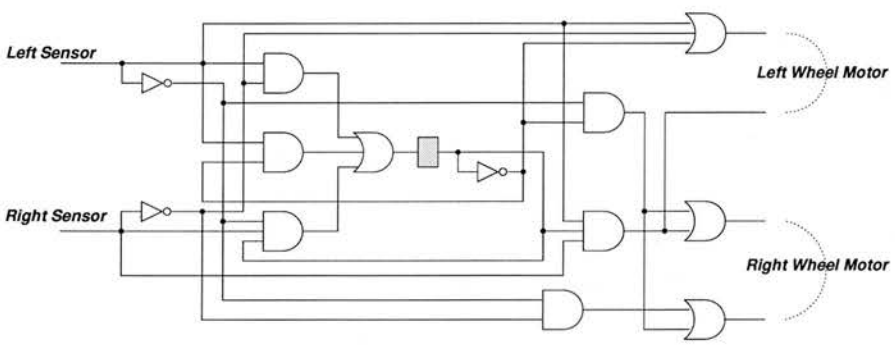


Figure 3.1: Boolean logic network with a memory element

which is equivalent to Table 3.1(a).

The encoding of the Mealy machine can easily represent sequential states as shown above. However, it needs a complete encoding for each state and scales badly with growing machine complexity.

3.1.3 Rule-based State Machines

A variation of a finite state machine is a rule-based system with binary inputs, similar to a classifier system. This structure also has a memory configuration where each rule belongs to a state, with the current state information always held globally. Each rule keeps the information of the matchable current state and the next state, much like tags in a classifier system.

When we have a small number of sensors and a few number of states, the gene encoding size of finite state machines is reasonable. But, the search time for the optimal gene or solution to desirable behaviors increases rapidly with the length of the chromosome. When we have n sensors, m motor actions and s states, the length of the chromosome for the representation of finite state machines will be $s(m+1)2^n$, assuming all the sensors have binary-values. For

example, when we need many sensors and several states, the gene encoding size increases exponentially.

A possible alternative is to use a set of rules where each rule has a state information and a sensor representation of short length. This structure is similar to a classifier system. Genetic operators are used for a set of rules encoded by bit strings. The basic structure can be seen as a rule-based system where preconditions for a set of rules are examined to see their applicability, when given the current situation. The preconditions have fixed-length bit encoding with values of 0, 1, #(don't care). The action part of the rule is also a fixed-length encoding with the values of motor action. Unlike a classifier system, evolutionary computation with a set of rules does not use any reinforcement learning[1]: the genetic algorithm creates new rules with genetic operators and a set of rules is tested for a given environment by evaluating the results of the chosen actions. As a result, this helps to construct a desirable set of rules for the best behavior even if it requires many sensor readings. Using a set of rules, where each rule has a state information and a sensor representation, k rules in a set, n sensors, m motor actions and s states results in a genome of length $k(n+s+m)$, which is linearly proportional to the number of sensors.

A set of rules including states is defined as a machine $M = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$ where q_0 is an initial state, Q is a finite set of states, Σ is a finite set of input values, Δ is a finite set of multi-valued output values, δ is a state transition function from $Q \times \Sigma$ to Q, and λ is a mapping from $Q \times \Sigma$ to Δ, $\lambda(q,A)$ will have one of the output set Δ, where q,A is an element of Q,Σ, respectively. Each rule R_i in machine M is defined as

$$R_i(M) = (A_i, (q_i, q_i^*), D_i) = ((\sigma_{i1}, \sigma_{i2}, \sigma_{i3}, ..., \sigma_{in}), (q_i, q_i^*), (d_{i1}, d_{i2}, d_{i3}, ..., d_{im}))$$

where A_i is an input string or sensor value, D_i is a motor output string, q_i is the current state before rule activation, q_i^* is the next state after rule activation.

σ_{i1}	σ_{i2}	σ_{i3}	...	σ_{in}	q_i	q_i^*	d_{i1}	d_{i2}	d_{i3}	...	d_{im}

In the rule structure, we can divide a set of state rules into several classes depending on states. We assume each rule should necessarily satisfy the condition of the current state to activate itself. State information can be generated from the best rule to match the current sensor readings. As shown in Figure 3.2, the state information can be seen as a medium to activate a set

[1] We are only interested in constructing Boolean logic circuits with evolutionary computation, since reinforcement learning allows dynamic features of weights for rule discovery

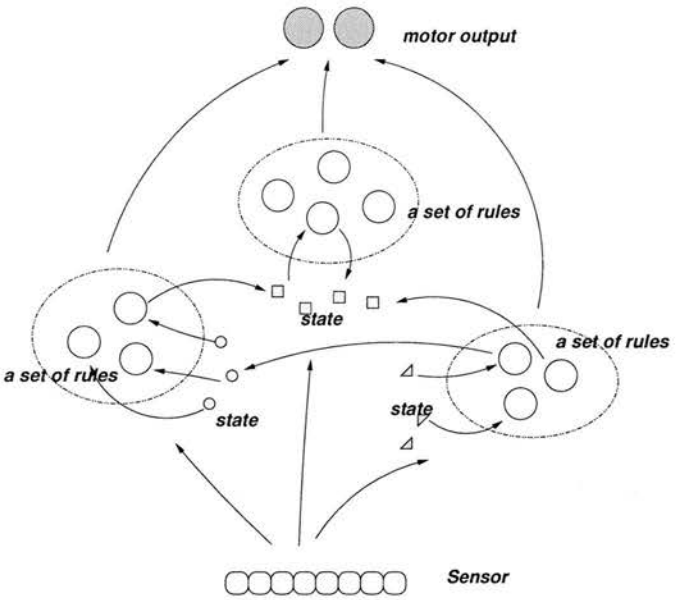


Figure 3.2: Diagram of a set of state rules

of rules, like neurotransmitters among neurons. When the current state, e.g. q_1, is set, it can activate only a set of rules which has state q_1 and the best rule among them will decide the next state q_2, which will activate another set of rules.

We assume that each rule is responsive to sensor readings and when the condition matches the current sensor readings, the rule action is activated. The firing activity of each rule depends on the number of matching conditions of the rule, and if there is a complete matching rule, the rule's activity is the highest. Each rule's motor actions are summed up as final motor action values depending on its activity. For the i-th motor value M_i can be calculated as follows:

$$M_i = \sum_n p_n R_{ni}$$

where $p_n = \frac{A_n}{\sum_i A_i}$, $A_n = e^{-z/\sigma^2}$, and A_n is an activity of the n-th rule, R_{ni} is the i-th motor action of the n-th rule, z is the number of preconditions matching with the current sensor values, σ^2 is a constant for activity degree, M_i is the i-th motor action. In this mapping, rules interact with each other to decide the final motor output.

Another way is to choose the single best matching rule in a winner-take-all strategy. If there is no rule whose condition matches the sensor readings completely, the best matching rule is selected. The best matching rule not only generates the next state, but also decides the motor output by itself. In our experiments, the evolutionary algorithm has shown a better performance

using this winner-take-all method.

The structure of rule-based state machines can be compared with the methods of radial-basis functions and fuzzy systems [Powell, 1987; Moody and Darken, 1989; Haykin, 1994; Zadeh, 1965; Jang et al., 1997].

3.1.4 Tree State Machines

Finite state machines and rule-based state machines can easily encode memory states, but they have the restriction of sensor values. Sensor readings are binarized to have logical values and thus it may ignore the potential of continuous-valued sensors. There is an alternative way of allowing variable thresholds for sensors. However, they still have the difficulty of representing multi-thresholds for each sensor, since the encoding size is greatly increasing. Thus, a tree structure to process sensor readings and the technique of encoding memory states are combined together. It will be called a tree state machine.

Definition 3.1 (Sensor state) A *sensor state* is defined to be a set of sensor values that have the same motor outputs and the same memory state to be activated next. It is defined within a fixed memory state. If the same notation is used as in the finite state machine,

$$\varepsilon(s^i) = \{s^* | \lambda(q_i, s^i) = \lambda(q_i, s^*) \ \wedge \ \delta(q_i, s^i) = \delta(q_i, s^*)\}$$

where λ is a mapping from $Q \times \Sigma$ to Δ, and δ is a state transition function from $Q \times \Sigma$ to Q.

A sensor state is an equivalence class over observed sensor values, such that all the sensor values in the sensor state have the same motor actions and the same next memory state in the transition functions. The sensor state equivalence classes form a partition of the set Σ^i of sensor values as shown in Figure 3.3. The function ε maps from a sensor value s^i to a set of sensor values, which belongs to sensor states $\sigma_k, k = 0, 1, 2, \ldots$. Then the set Σ^i for a state q_i is the set $\{\sigma_k\}$ of all sensor states. Within the same sensor state, all the sensor values induce the same future actions for the next time step.

In finite state machines, there are $\Pi_{k=1}^{n} \mu(S_k)$ sensor states where Π is the product of each value, n is the number of sensors and μ is a function from a sensor to the number of partitions formed by thresholds. This structure is quite efficient to evolve memory-based systems or sequential processes, while the number of sensors is not large. As an alternative, a rule-based system was designed to handle evolving controllers with many sensors. Even in this case, the number of

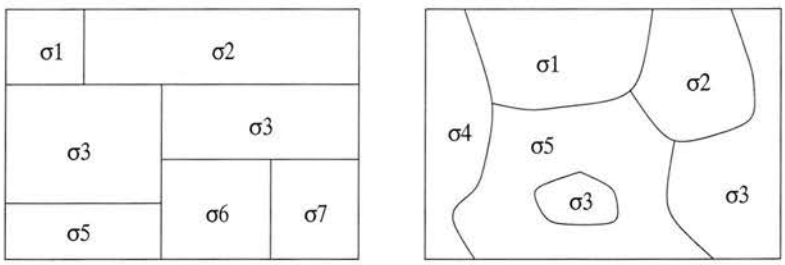


Figure 3.3: Representation examples of partitioning the set Σ^i

sensor states is restricted. When 20 rules are tested for a chromosome, it means a maximum of 20 sensor states are allowed.

A set of state trees is defined as a machine $M = (Q, T, \Sigma, \Delta, q_0)$ where q_0 is an initial state, Q is a finite set of states, T is a set of trees assigned to a state, Σ is a set of sensors and Δ is the set of multi-valued output values. A tree T_i is assigned for a state q_i. Each tree T_i in machine M is defined as

$$T_i(M) = \{(S_k, q_k^*, D_k) | S_k \in \Gamma, q_k^* \in Q, D_k \in \Delta\}$$

where Γ is a set of sensor states, and a sensor state is defined with n decision boundaries as $S_k = (r_{k1}, \sigma_{k1}, \theta_{k1}) \wedge (r_{k2}, \sigma_{k2}, \theta_{k2}) \wedge ...(r_{kn}, \sigma_{kn}, \theta_{kn})$, and then $(r_{kj}, \sigma_{kj}, \theta_{kj})$ is a decision classifier for the sensor $\sigma_{kj} \in \Sigma$. r_{kj} is a relation operator which will be an element of the set $\{>, \geq, <, \leq\}$ in a decision tree, $\sigma_{kj} \in \Sigma$ is the j-th sensor, and θ_{kj} is a threshold for the j-th sensor. We can omit insignificant sensors in the sensor state equation by evolving decision trees for sensor states.

For each sensor state, δ, λ are defined for the next state and motor outputs. δ is a state transition function from $Q \times \Gamma$ to Q, and λ is a mapping from $Q \times \Gamma$ to Δ, $\lambda(q, A)$ will have one of the output set Δ, where q, A is an element of Q, Γ, respectively. For the i-th sensor state, $(q_i^*, D_i) = (q_i^*, (d_{i1}, d_{i2}, d_{i3}, ..., d_{im}))$ will be the form of a terminal value in evolved decision trees. D_i is a motor output string, and q_i^* is the next state after the activation of motor output for a given sensor state.

A sensor state determines the next memory state and motor actions. Thus, this structure is similar to the rule-based system, but it has more flexibility in representation and also it allows to evolve thresholds for each sensor. It is expandable to generate many sensor states so that the system can handle the environment.

GP-automata has been applied only with two sensor states for several experiments [Ashlock, 1997] as shown in Figure 2.9; two sensor states were determined by odd or even values of

deciders. By the definition of sensor states, two sensor states say that one memory state can have a maximum of two actions and two transitions of memory states.

In the approach of tree state machines, we allow more than two sensor states using a tree structure. In the above tree state machine, we can theoretically generate an infinite number of sensor states. For any sensor, we can continue to divide each sensor space with multi-thresholds. There may be issues of how many sensor states will be useful to construct desirable controllers and also of how complex functions will be helpful for a fixed number of sensor states. It is presumed that these issues will depend on experimental tasks.

3.2 Fitness Function

In evolutionary robotics, the fitness function is used to evaluate the performance of controllers and control structures, or morphology. Normally fitness functions for robot controllers have constraints to estimate the performance for desirable behaviors. However, it is a non-trivial problem to choose such constraints and also it may be often the case without knowledge of how to represent the expected behaviors with constraints. Even if relevant constraints are obtained, there is still a problem of combining all constraints for desirable behaviors. Some fitness functions can have difficulty in making progress of evolution to expected behaviors, since they are too deceptive or they define only behavioral outcome of an evolutionary controller.

Nolfi and Floreano [2000] proposed three factors for the fitness function framework. The first factor is a functional-behavioral aspect to define whether the fitness function evaluates specific function components of the controller or whether the fitness function evaluates the behavioral outcome. The second factor is an explicit-implicit aspect to estimate the amount of constraints and variables for the fitness. The third factor is an external-internal aspect to define whether constraints and variables included in the fitness are computable using information available to the evolving agent. They insisted that those three factors determine the characteristics of the evolution of autonomous agents. The difficulty of fitness functions depends on the factors. In many cases, the fitness function cannot exactly describe good controllers for autonomous agents. Thus, the choice of fitness functions is one of difficult decisions.

In the thesis we handle several grid world problems and robotic tasks. Grid world problems consist of the artificial ant problem, the Tartarus problem, and the woods problem. In robotic experiments, several robotic tasks such as wall following, exploration, obstacle avoidance, box pushing, corridor following and T-maze decision are investigated. Those tasks reflect various

fitness characteristics for behavior performance. For some behaviors, the fitness function is defined by the penalty function to estimate penalty or cost for expected behaviors. The performance of a controller can be evaluated with the penalty or cost that the controller produces. It can be easily encoded as a penalty fitness.

When two or more constraints are given, combining constraints with relative weights can be a possible solution, but the fitness greatly depends on the relative weights. In our experiments, two constraints, behavior performance and memory amount, are mainly used to find genome controllers with the best performance and the minimum amount of memory. Instead of combining two constraints with relative weights, Pareto optimization is used to collect controllers to satisfy two independent constraints. It will show various robot controllers with the best behavior performance for each different level of memory amount.

The performance of controllers, defined by the fitness function, are compared one another to determine which controller is better. However, evolutionary runs are stochastic and a single run cannot support such a decision. Thus, the probabilistic distribution of trial runs should be considered.

3.3 Performance Test

In evolutionary computation, we are inclined to compare different strategies or different parameter settings. An easy way is to compare the distributions of fitness results of those strategies. Some strategies may reach optimal solutions at a slow speed but better fitness, while others reach desirable solutions at a fast convergence speed but with worse fitness. In many experiments we need a large number of generations to reach optimal solutions. Also independent experiments can be tested with a rather small number of generations to obtain desirable solutions within an error bound. This will determine how much effort is required to obtain a certain success level for controllers. Two performance tests will be applied to several tasks to compare various strategies and to see memory effects.

3.3.1 Fitness Test

When many independent experiments are run, the fitness results for each one will differ somewhat. Unless the fitness function is deceptive, the fitness performance will follow a trend of fitness curves over generations. We may assume that the fitness curve follows a Gaussian dis-

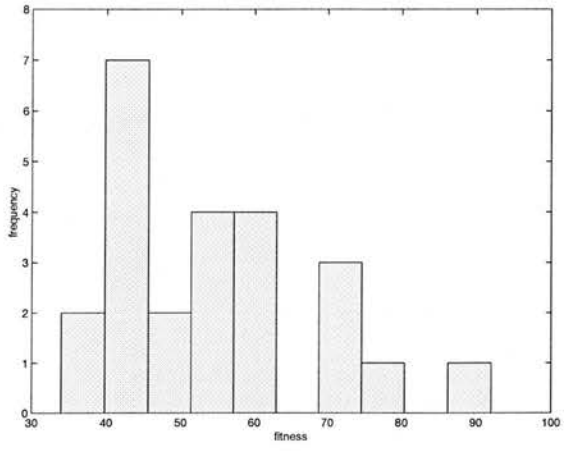


Figure 3.4: An example fitness distribution

tribution at a fixed number of generations. However, the number of runs for each experiment will be restricted, since one evolutionary run takes much computing time. Figure 3.4 shows an example penalty-fitness distribution obtained from 25 runs in an experiment. The distribution rarely follows a Gaussian distribution and in many cases it shows a skewed form. For various configurations or strategies, a set of independent runs are taken to compare performances. An easy way is to see the average fitness value at fixed generations.

We take a Student's t test over sample runs with a given strategy [Cohen, 1995]. It allows us to see confidence intervals of the average performance in the sample distribution. The population standard deviation is unknown, so it is estimated from the sample standard deviation $\frac{\sigma}{\sqrt{n}}$ where σ is the standard error of samples. Thus, 95% confidence intervals should be

$$[\bar{x} - t_{0.025}\frac{\sigma}{\sqrt{n}},\ \bar{x} + t_{0.025}\frac{\sigma}{\sqrt{n}}].$$

where n is the number of samples.

The data in Figure 3.4 have an average $\bar{x} = 54.75$ and standard deviation $\sigma = 14.51$. Its confidence interval will be $[54.75 - 5.99,\ 54.75 + 5.99] = [48.76,\ 60.74]$, when a t-distribution is assumed.

When we test the confidence intervals with bootstrap methods, it is a little different from our t-distribution test. Normally the confidence interval of the bootstrap method [Cohen, 1995] seems narrower than that of t-distribution. The estimated mean among two methods will depend on a sample distribution. From the above example runs, the bootstrap sampling method generated 5000 values of $\bar{x}^*$, the estimated mean calculated from bootstrap samples. The dis-

tribution of estimated mean values is given in Figure 3.5. From these data, the 95% confidence interval for the mean in terms of the standard normal distribution is $[\bar{x}-1.96\sigma_{\bar{x}^*},\ \bar{x}+1.96\sigma_{\bar{x}^*}] = [52.56-6.74,\ 52.56+6.74] = [45.82,\ 59.30]$. As a result, t-test and bootstrap test give similar results for an unknown distribution. In this thesis the fitness test will mainly check the confidence interval using the t-test, since it tends to cover more often the range of the worst case fitness.

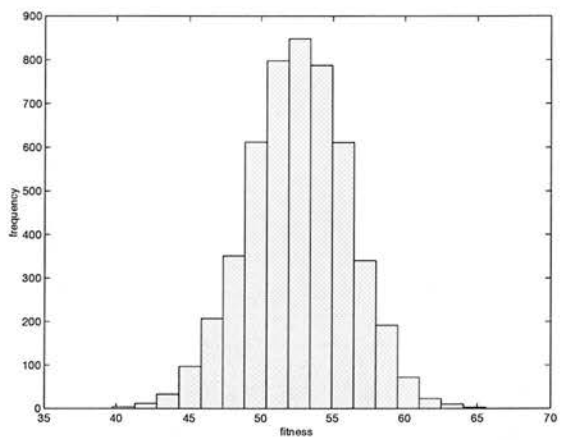


Figure 3.5: Sampling distribution derived from the bootstrapped distribution (5000 samples)

3.3.2 Effort Test

We are interested in obtaining a successful controller with minimum efforts in evolutionary experiments. If we increase the population size or the number of test environments[2], it will be more effective to find desirable controllers. However, if a successful controller is determined with the lowest computing cost, the applied methodologies can be recommended.

To determine the effort that evolutionary computation has tried to obtain a desirable robust controller, we assume that a single independent experiment is repeatedly run until a successful controller is found. A better strategy or methodology will have a smaller number of runs to find a successful controller. This performance test can be differentiated with the comparison of fitness distributions under different strategies.

This effort test was suggested by Lee [1998] to see the performance of different strategies for evolutionary experiments. He used the measure of the average computing cost needed before

[2]To build robust controllers, robots should be evolved and tested at many starting positions in various environments.

the first success run. In this thesis, its measure is extended more rigourously to show the confidence intervals of the effort cost in this thesis.

For a given success rate p over the controller test, the average number of trial runs before the first success run can be calculated as

$$E(X) = \sum_{x=1}^{\infty} xp(1-p)^x = \frac{1}{p}$$

Therefore, the effort cost before the first success will be $\frac{1}{p}C$ where C is a unit computing cost per run.

When it is assumed that $\alpha+\beta$ independent experiments experience α successes and β failures, the distribution of success rate p can be approximated with a Beta distribution (see appendix A). The Beta probability density function is given by

$$f(p,\alpha,\beta) = \frac{1}{B(\alpha+1,\beta+1)} p^{\alpha}(1-p)^{\beta}$$

where $B(\alpha+1,\beta+1) = \int_0^1 p^{\alpha}(1-p)^{\beta}dp = \frac{\Gamma(\alpha+1)\cdot\Gamma(\beta+1)}{\Gamma(\alpha+\beta+2)}$ and $\Gamma(n+1) = n\Gamma(n)$. A random variable X is defined as the amount of trial cost with a success probability p. The expected value and variance of the cost for the random variable X will be as follows:

$$\begin{aligned} E(X) &= \int_0^1 \frac{C}{p} \cdot \frac{1}{B(\alpha+1,\beta+1)} p^{\alpha}(1-p)^{\beta}dp \\ &= C \cdot \frac{\alpha+\beta+1}{\alpha} \\ V(X) &= E(X^2) - [E(X)]^2 = C^2 \cdot \frac{(\alpha+\beta+1)(\beta+1)}{\alpha^2(\alpha-1)} \end{aligned}$$

Similarly a negative binomial distribution can be defined such that a random variable Y is the number of failures before the first success. In this case, it will be a well-known geometric distribution.

$$\begin{aligned} E(Y) &= \frac{1}{p} \\ V(Y) &= \frac{1-p}{p^2} \end{aligned}$$

where p may be approximated with $\frac{\alpha+1}{\alpha+\beta+2}$ (appendix A) in the experiments and thus the expected trial cost will be $\frac{1}{p}C = \frac{\alpha+\beta+2}{\alpha+1}C$.

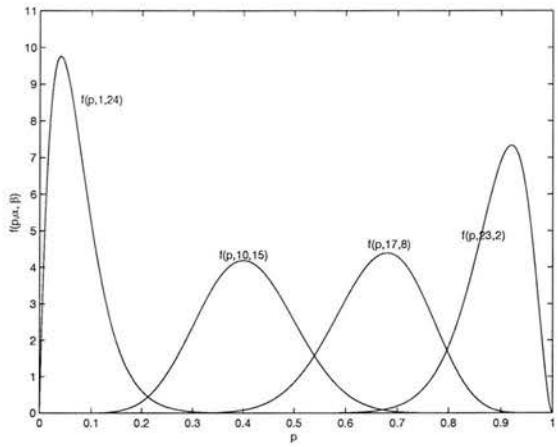


Figure 3.6: Probabilistic density functions of Beta distribution (degree: 25)

Two distributions show similar forms of expected costs, but the variances are different. When a different number of trial experiments are given with the same success probability $p = \frac{\alpha+1}{\alpha+\beta+2}$, the variance of Beta distribution will depend on the number of experiments.

Now we want to define confidence intervals about success/fail tests for a given strategy. If the number of experiments is large and the success rate is not an extreme value, then the distribution will be close to normal distribution. For most robotic tasks, evolutionary computation requires much time to test many experiments. The distribution of success probability will follow a beta distribution unlike a Gaussian distribution. Figure 3.6 shows the pdf functions $f(p,\alpha,\beta)$ over a different number of success cases among 25 runs. The beta has very diverse distributions depending on the success cases. For a small or large number of successes, the distribution is far away from standard normal distribution. Thus, the following numeric approach will be used for confidence intervals. Assume that a random variable X with a beta-distribution has the upper bound x_u and the lower bound x_l for confidence limits such that $P(x_l < X \leq x_u) = 1-\varepsilon$ and $P(X \leq x_l) = \varepsilon/2$. Then we can assert that $[x_l, x_u]$ is a $(1-\varepsilon)\cdot 100$ percent confidence interval.

If a success probability p is beta distributed, the confidence limits x_l, x_u can be obtained by solving the following equations:

$$\begin{aligned} \frac{\varepsilon}{2} &= \int_0^{x_l} \frac{p^\alpha (1-p)^\beta}{B(\alpha+1,\beta+1)} dp \\ \frac{\varepsilon}{2} &= \int_{x_u}^1 \frac{p^\alpha (1-p)^\beta}{B(\alpha+1,\beta+1)} dp \end{aligned} \tag{3.1}$$

From the lower and upper bound probability x_l, x_u, the 95% confidence effort cost will be

estimated with $[\frac{1}{x_u}C, \frac{1}{x_l}C]$.

For example, when 10 independent experiments are tested for a robotic task, assume that 9 successes and 1 failure happen for a given strategy. The 95 % percent confidence interval can be estimated with the above two integral equations with $\varepsilon = 0.05$. If we solve the equations in terms of x_l, x_u, then

$$\begin{aligned} \frac{0.05}{2} &= \int_0^{x_l} \frac{p^9(1-p)^1}{B(9+1,1+1)}dp \\ \frac{0.05}{2} &= \int_{x_u}^1 \frac{p^9(1-p)^1}{B(9+1,1+1)}dp \end{aligned}$$

$$11x_l^{10} - 10x_l^{11} = 0.025, \quad 11x_u^{10} - 10x_u^{11} = 0.975,$$

which gives an approximate solution $x_l = 0.587, x_R = 0.977$. Thus, $[0.587, 0.977]$ is a confidence interval for a success rate p in this example. The effort cost interval will be $[\frac{1}{0.977}C, \frac{1}{0.587}C]$ = $[1.02C, 1.70C]$ and its mean $\frac{\alpha+\beta+1}{\alpha} \cdot C$ is $1.22C$. We can estimate that 1.02–1.70 times the computing time of a single run experiment will be spent to obtain the first success with 95% confidence.

This effort test will be applied to check the performance of many single experiments in a short period of time. By repeating the single experiments with the same number of generations, we can decide a distribution over success rate. If it is evolved with good strategies or proper parameters, the success rate will be high enough. Otherwise, the success rate will be lower. The success rate distribution for each strategy will be compared to determine which strategy is more efficient.

3.4 Multi-Objective Optimization

In the robotic tasks, we are interested in finding a genome controller satisfying both the minimal expression and the best fitness together. The number of memory states and performance fitness are independent measures and the problem is one of multiobjective optimization problems.

Unless prior information about control structure is given, the design of the control structure should consider its structural complexity. The complexity is estimated with memory in our problems. The criterion for memory complexity will follow the minimum description length (MDL) principle [Rissanen, 1978, 1989]. We prefer the best performance with the minimal control structure and thus two criteria, fitness performance and memory amount, are used for

desirable controllers. It may be tested with a regularization method that considers the relative importance of the memory criterion with respect to fitness performance. Instead of allowing several regularization parameters, various Pareto optimal solutions under the two criteria will be investigated. This optimization process will be achieved with evolutionary algorithms.

Evolutionary algorithms have often been used for optimization of several conflicting objectives. This multiobjective optimization searches for multiple solutions in a single run. There exists a set of Pareto optimal solutions in a multiobjective optimization problem instead of a single optimal solution. Many classical optimization methods work with a single solution at each iteration, and multiple Pareto solutions can be obtained by applying the methods more than once with varying parameters, thus finding one distinct Pareto solution for each iteration. In contrast, evolutionary algorithms can work with a population of solutions, and the genetic search has its inherent parallelism. Therefore, many Pareto-optimal solutions can be obtained together in a single optimization run.

Evolutionary algorithms for multiobjective optimization have been studied since Schaffer's work [Schaffer, 1985]. A large number of applications have been tackled with the approach. In our experiments, a special-purpose evolutionary multiobjective optimization (EMO) approach is developed for robotic tasks. It will find Pareto optimal solutions, the best behavior performance for each number of memory states.

The fitness function can be defined with cost or penalty and for the penalty fitness, evolutionary algorithms will search for the minimal fitness. Thus, the minimization of cost as an objective can be achieved with evolutionary algorithms. We will assume that there are m objectives to be minimized for multiobjective optimization[3].

Definition 3.2 (Dominate) A vector $X = (x_1, x_2, ..., x_m)$ for m objectives is said to *dominate* $Y = (y_1, y_2, ..., y_m)$ (written as $X \prec Y$) if and only if X is partially less than Y, that is,

$$(\forall i \in 1, ..., m, x_i \leq y_i) \wedge (\exists i \in 1, ..., m, x_i < y_i)$$

Definition 3.3 (Pareto optimal set) A *Pareto optimal set* is said to be the set of vectors that are not dominated by any other vector.

$$\{X = (x_1, ..., x_m) | \neg(\exists Y = (y_1, y_2, ..., y_m), Y \prec X)\}$$

[3]If the objectives are to be maximized as a benefit function, the relational operator should be changed reversely.

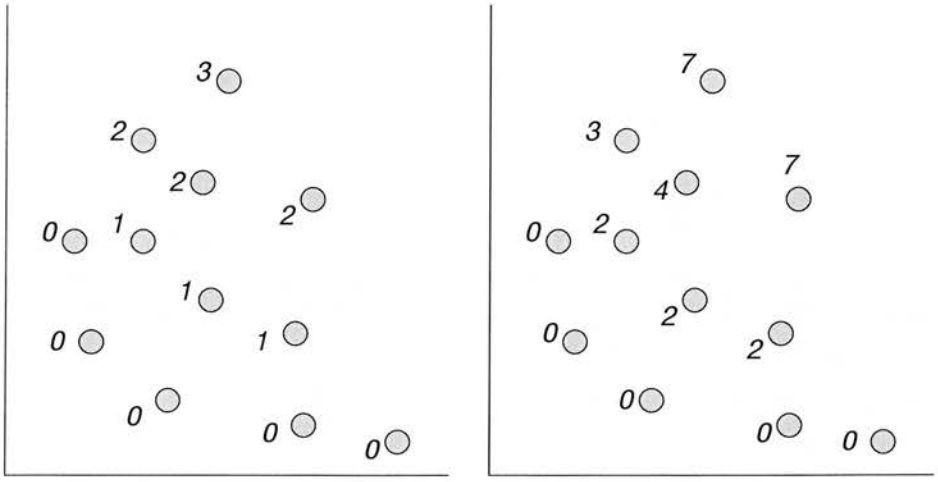


Figure 3.7: Pareto ranking example (a) Goldberg method (b) Dominating rank method

A population of vectors can be classified into classes of dominance levels [Goldberg, 1989]. We can apply a procedure of selecting non-dominated vectors (it means that vectors are not dominated by others) for the first dominance level. This procedure compares each element vector with every other element in a population and checks if it is dominated by any other vector. The resulting set of non-dominated vectors are marked as the first level and removed from the population of element vectors. The procedure of selecting non-dominated vectors is applied again to choose the next (second) dominance level. By repeating the selection of non-dominated vectors and their deletion, a hierarchy of dominance levels is achieved.

The vectors of the first-level dominance class are a set of possible Pareto optimal solutions and they should be given more probabilities to reproduce the next generation in evolutionary algorithms. The above Pareto ranking method designed by Goldberg [1989] is given in Figure 3.7(a).

A ranking method is used where the number of dominating vectors are counted in a population instead of using dominance levels (this ranking method was first suggested by Foncea and Fleming [1993]). Nondominating vectors will be given a rank 0, and then the rank of a vector will be defined as how many vectors dominate the given vector, that is, the number of other vectors dominating the vector.

There are two main issues to be addressed when applying evolutionary algorithms to multiobjective optimization problems. EMO algorithms are required to have a good search operator and fitness assignment to find the global Pareto-optimal solutions. Also they need to address

how to maintain a diversity of the population to prevent a biased distribution of solutions. The fitness assignment method and diversity will be mentioned below for the ranking method.

3.4.1 Fitness Assignment

The ranking method considers dominating relationships among solution vectors in multiobjective optimization.

For any vector X in a population, we define a dominating set over X as

$$D(X) = \{Y \in \Psi | Y \prec X, \quad X \in \Psi\}$$

where Ψ is a set of all vectors in a population.

Thus, the rank function will be given as

$$rank(X) = Card(D(X))$$

where *Card* is a cardinality function to count the number of elements in a given set. This *rank* function will not be consistent with the result of dominance levels suggested by Goldberg [1989]. Within the same dominance levels, the *rank* function results may be different, depending on the distribution of vectors in a population. EMO with the dominating rank method will assign fitness to the result of the *rank* function, that is, the number of dominating vectors. Figure 3.7(b) shows examples of how the above method assigns fitness to vectors.

3.4.2 Diversity in Pareto optimization

In general, discontinuity in the Pareto-optimal search space may lead to difficulty in an multiobjective evolutionary algorithm. It needs an efficient way of keeping diversity among discontinuous regions (see Figure 3.8). In this kind of problem, solutions can be obtained in local sub-regions, but competition among local regions may neglect to search some sub-regions and it may lose diverse solutions with a biased distribution. This diversity problem is one of the issues in EMO approaches. In our experiments, behavior performance is defined for each number of memory states. Solution space is discontinuous and we need to escape biased distribution of Pareto solutions.

Thus, we keep an elite pool of the best chromosomes for each number of memory states. Every

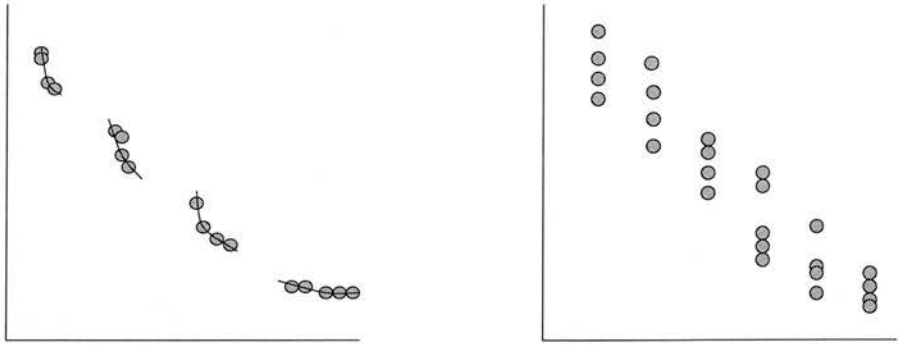


Figure 3.8: Multiobjective solutions in discontinuous Pareto space

generation the best chromosomes for each number of states in a population are collected and compared with the pool members. If the best chromosome in the population is better than the pool member for a specified state, the pool member is replaced. Otherwise, the pool member is still maintained in the elite pool. In this way, the elite chromosome for each state is stored. To keep diversity of Pareto solutions, one random member in the elite pool is selected and inserted into a new population. Such random selection for memory states will assist a uniform distribution of Pareto solutions. It is also effectively related to elitism strategy.

When we use tournament selection with a tournament of size four, a similar strategy can be applied. A population is shuffled into groups of four chromosomes for tournament selection. In each group of size four the two best chromosomes are crossed over to produce two new offspring that replace the two worst fit chromosomes in the group and a mutation is given for each new chromosome. The two best chromosomes also reproduce themselves for another two offspring. This process can retain the two old best values in the group of four and a variance of the whole population's fitness values can be reduced.

In multiobjective Pareto optimization, the rank cannot be linearly ordered. The dominating ranking method is applied to this tournament selection of size four as shown in Figure 3.9. When more than two offspring have the the same rank, the two best chromosomes are selected randomly without any bias to an objective. In this tournament selection, this random selection also plays an important role on providing a diverse population in the next generation.

3.4.3 Variable State Machines

A minimal expression of controllers to achieve a given goal is desirable. FSM approaches need to find the minimum number of states in terms of simple design. If a task can be com-

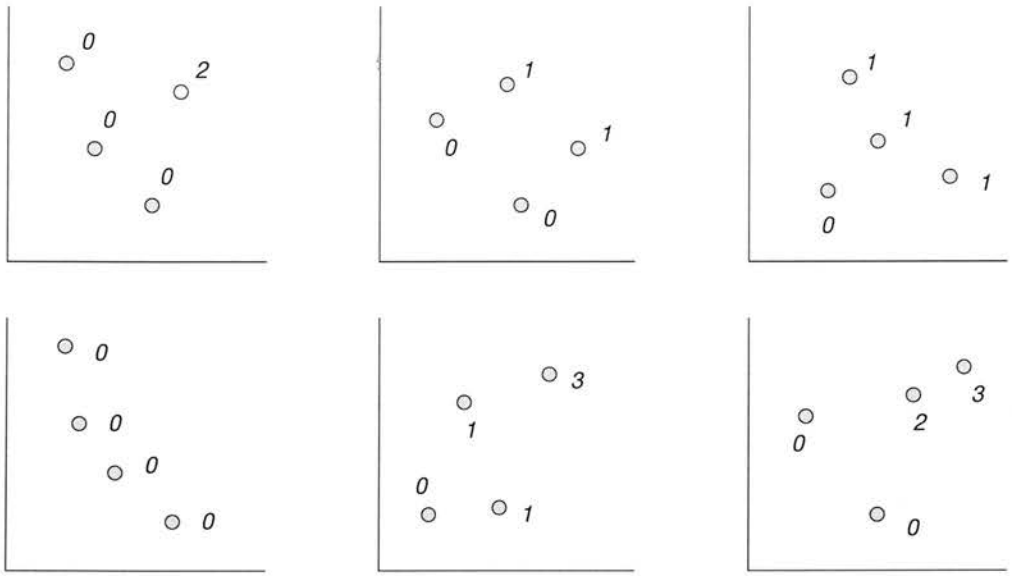


Figure 3.9: Dominating rank method in a tournament selection of group size four

pleted with a purely reactive system, it is represented as a 1-state FSM. We have developed an evolutionary algorithm with Pareto optimization for variable state machines. This does not only handle evolving compact representations of controllers, but also fitness distribution over variable states can be analyzed to determine a desirable number of memory elements. Two objectives, behavior performance and the number of memory elements, are used in Pareto optimization. Pareto optimization will try to maximize behavior performance as well as minimize memory states and thus a minimal expression of controllers can be searched out.

Unlike many applications of multiobjective optimization, a genetic pool should allow variable length chromosomes in the experiments of variable state machines. The size of finite state machines depends on the number of states and thus variable states have a different length of genome controllers.

In the experiments, a tournament selection of size four is used for Pareto optimization. A population is initialized with random length chromosomes. The two best chromosomes are selected with the dominating rank method as mentioned above. They reproduce themselves and the two worst chromosomes are replaced by two new offspring that experience one point crossover followed by mutation. In the application of variable state machines, a variable number of states should be permitted and generated to keep its diversity in a genetic pool. Thus, new offspring are produced with a size modifying operator, crossover and mutation by turns.

There are three different methods for a size-modifying operator as shown in Figure 3.10. When

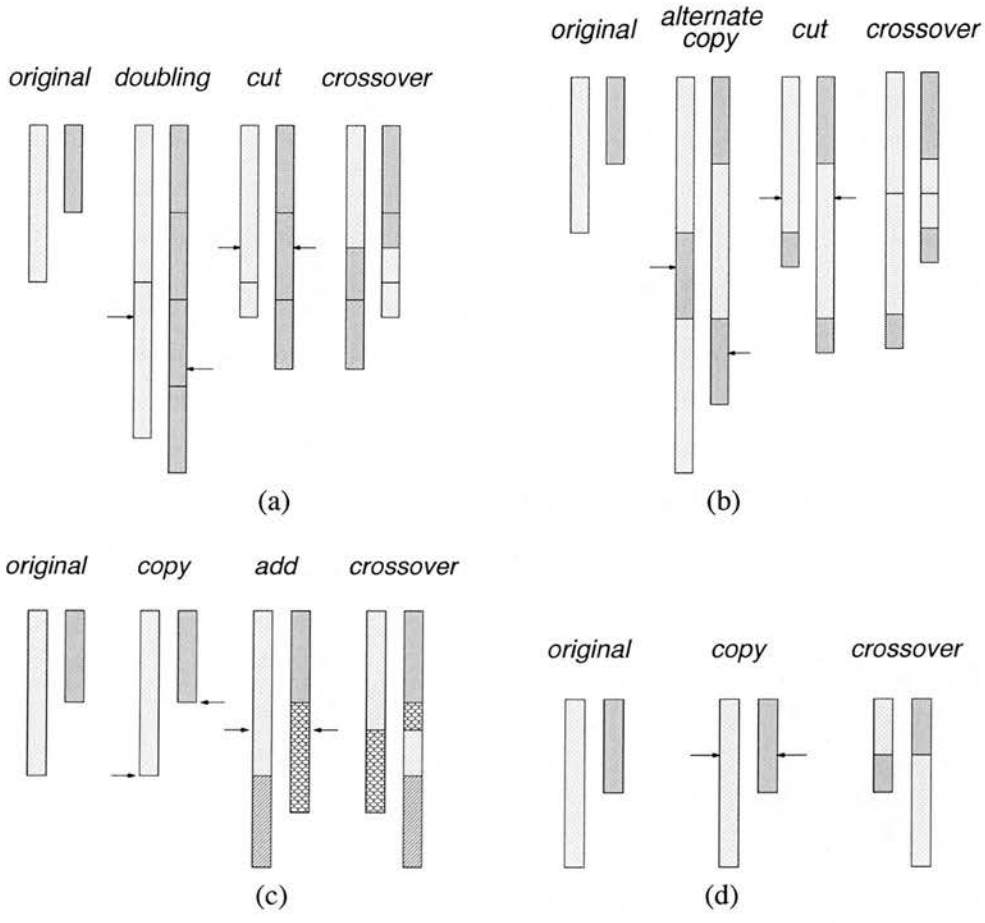


Figure 3.10: Genetic operators for variable state machines

new offspring are reproduced, memory states are randomly selected for the new offspring to produce diverse state machines. Thus, the chromosome size for each chromosome will depend on the pre-defined memory states. The following size-modifying operators will be used to reproduce new offspring such that they share same of the characteristics of the parents.

With the first operator, the number of states for the offspring chromosome string is chosen at random and the chromosome size is thereby fixed. Then it concatenates several copies of one of the parents. If it exceeds the chromosome length, it is cut by the pre-defined length. This is repeated for the other offspring, but the offspring is generated from the other parent – see Figure 3.10(a).

With the second operator, the number of states for the offspring is randomly chosen and so the chromosome length is calculated. It concatenates a copy of one parent and a copy from the other parent and then repeats to concatenate two copies alternately. The chromosome is cut at the pre-defined length. Another offspring is generated with alternate copies of parents but

starting with the other parent – see Figure 3.10(b).

With the third operator, similarly the number of states for offspring is randomly chosen. It maintains one copy of one of the parents and if its size is larger than the pre-defined chromosome length, it is cut by the length. If it is smaller than the length, random strings are added to the copy to build it up to the pre-defined length. Another offspring is generated in a similar way, but instead it starts with a copy of the other parent – see Figure 3.10(c).

After applying the above size-modifying operator, crossover of different length is executed on the two offspring. The crossover point is selected inside both chromosome strings after aligning the prefixes of the two strings. During this crossover process, the bit flipping mutation can be used to change one integer value in the strings.

The size-modifying operator is applied to 75% of new offspring, in experiments with variable state machines. When it is not used, the offspring keep the size of their parents as shown in Figure 3.10(d).

3.5 Summary

In this chapter, several methodologies for evolutionary algorithms are introduced. Control structures are based on state machines to help quantify the amount of memory required to achieve a given task. We suggest three kinds of control structures, finite state machines, rule-based state machines and tree state machines. They each have advantages and disadvantages in representation complexity and application levels. Finite state machines are useful for a small number of sensors, but they easily catch memory components in controllers. Rule-based state machines can be used for a large number of sensors, but they need a matching process between environmental states and conditions of each rule. Tree state machines can be used with a genetic programming to evolve multiple decision trees where a tree is defined for each internal state. They allow a continuous range of sensors, but are rather slow at obtaining desirable solutions.

Many evolutionary runs for a given control structure or given parameter setting are performed to see the fitness distribution. A fitness test and an effort test will be applied to see distinct differences among control structures and to determine desirable evolutionary parameters. The two tests are based on confidence intervals of experimental samples to produce significance statistics.

A special purpose EMO approach is developed to find the memory requirement for each robotic

task or agent problem. It manages a pool of variable state machines to compete against each other in a given environment. It uses genetic algorithms with variable-length coding. Also the approach employs a dominating rank method with elitism to accelerate the convergence to Pareto-optimal solutions. From the EMO results for memory analysis, we can easily analyze how many memory elements are required to reach a given performance or what is the limit of performance for a given number of memory elements.

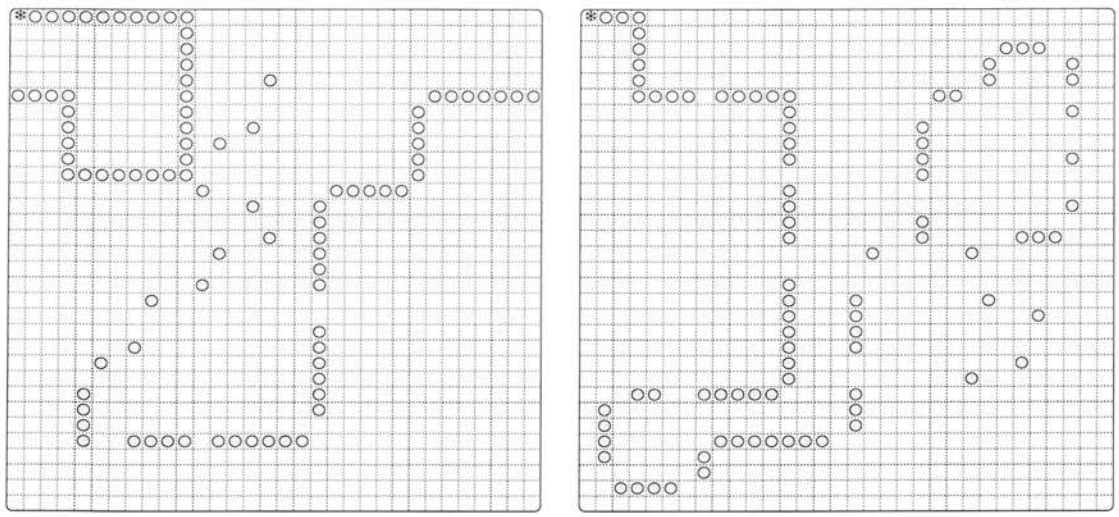
Chapter 4

Artificial Ant Problem

In this chapter, evolutionary algorithms are applied to the artificial ant problem. The artificial ant problem is a well-known problem that various evolutionary algorithms have handled. In this problem, an agent must follow irregular food trails in the grid world to imitate an ant's foraging behavior. The trails have a series of turns, gaps, and jumps on the grid and ant agents have one sensor in the front to detect food.

The artificial ant problem, as an agent problem, simplifies sensor readings and motor actions. Agents have restricted information of the surrounding environment. Yet they are supposed to collect all the food on the trails. Two trails, the Santa Fe trail and the John Muir trail, are handled with an ant agent in this chapter. The ant model is easy to characterize, but the task requires memorizing various environmental situations. The ant problem is one of the difficult problems that purely reactive systems cannot solve, and a desirable agent behavior should depend on memory elements. Thus, the artificial ant problem is a good example to see memory effects in behavior performance.

Several evolutionary issues are also discussed in this chapter. First, several factors are described; exploration time, the number of generations and population size, which are related to the computing cost for evolutionary runs. Then, the application of both the fitness test and the effort test to choose good parameter values for those factors is discussed. Also it explains how to find a desirable mutation rate and how to choose a genome selection mechanism. In a similar way, it discusses what control structures can help solve the artificial ant problem with a very limited sensor. Finite state machines are used to measure the amount of memory needed for the given task. The memory-based control structures, performance measure and Pareto optimization method mentioned in chapter 3 will be tools used to recognize the relevance and



(a) (b)

Figure 4.1: Artificial ant trails (a) John Muir trail (b) Santa Fe trail

importance of memory to solve the problem.

4.1 Artificial Ant Problem and Memory

The artificial ant problem is a simple navigation task that imitates ant trail following. The first work, by Jefferson and Collins [1991], used the John Muir trail, and another trail, called Santa Fe trail, was studied with genetic programming by Koza [1992]. The trails are shown in Figure 4.1. The task is for an artificial ant agent that must seek all the food lying in a grid and follow the irregular trail for the food.

This problem was first solved with a genetic algorithm to test the representation problem of controllers by Jefferson and Collins [1991]. A large population of artificial ants (65,536) were simulated in the John Muir trail with two different controller schemes, finite state machines and recurrent neural networks. In the John Muir trail, the grid cells on the left edge are wound adjacent to those on the right edge, and the cells on the bottom edge are wound adjacent to those at top. Each ant has a detector to sense the environment and an effector to wander about the environment; one bit of sensory input to detect food and two bits of motor actions to move forward, turn right, turn left and think (no operation). Its fitness was measured by the amount of food it traverses in 200 time steps. At each time step, the agent can sense the environment and decide on one of the motor actions. The behaviors of ant agents in the initial population were random walks. Gradually more food trails are traced by evolved ants.

```
(if-food-ahead  (move)
                (progn3  (left)
                         (progn2  (if-food-ahead (move)
                                                 (right))
                                  (progn2 (right)
                                          (progn2 (left) (right))))
                         (progn2  (if-food-ahead (move)
                                                 (left))
                                  (move))))
```

Figure 4.2: Control strategy for Santa Fe trail by Koza's genetic programming [Koza, 1992]

state	input 0	input 1
q_0	q_1, L	q_0, M
q_1	q_2, R	q_2, M
q_2	q_3, R	q_3, R
q_3	q_4, L	q_4, M
q_4	q_0, M	q_0, M

(a)

state	input 0	input 1
q_0	q_7, R	q_6, M
q_1	q_6, N	q_2, R
q_2	q_5, R	q_5, M
q_3	q_0, R	q_1, L
q_4	q_2, L	q_5, M
q_5	q_6, N	q_4, M
q_6	q_0, R	q_6, M
q_7	q_2, R	q_2, L

(b)

state	input 0	input 1
q_0	q_5, R	q_6, N
q_1	q_0, R	q_0, N
q_2	q_4, M	q_3, M
q_3	q_6, R	q_3, M
q_4	q_5, M	q_0, L
q_5	q_3, M	q_4, L
q_6	q_1, R	q_2, M

(c)

Figure 4.3: Finite state machines for Santa Fe trail problem (a) 405 time steps, with 8 states (b) 379 time steps, with 8 states (c) 356 time steps, with 16 states (input 1: food-ahead, input 0: no food ahead), output set is L (turn left), R(turn right), M (move forward), N (no-operation)

From an analysis of the Sante Fe trail, the best controller will be a 165 time step controller to memorize the entire trail perfectly. Since there are 89 food cells, 55 blank cells and 21 turns in the closest path to be traversed, $89+55+21 = 165$ time steps will be the best solution. In this case, much information is needed to memorize the trail path. For the John Muir trail, there are 89 food cells, 38 blank cells and 20 turns. The most efficient controller needs $89+38+20 = 147$ time steps to memorize the entire trail path.

Koza [1992] applied genetic programming to the artificial ant problem with the Sante Fe trail (see Figure 4.1(b)), where he mentioned it as a slightly more difficult trail than the John Muir trail, because the Sante Fe trail has more gaps and turns between food pellets. Instead of finite state machines, he used several primitive functions to evolve controllers. Sensor information is first processed with a `if-food-ahead` function. It has two conditional branches depending on whether or not there is a food ahead. The `progn` function connects an unconditional sequence of steps. For instance, the S-expression `(prog left move)` directs the artificial ant to turn left

and then move forward in sequence. Koza used a similar fitness measure to evolve controllers as in the John Muir trail problem, that is, the amount of food ranging from 0 to 89 traversed within 400 time steps. In his experiments, a genetic program did not specify the explicit state, so the evaluation of the S-expression is repeated if there is additional time available. Figure 4.2 is one of the best control strategies found [Koza, 1992].

A simple model of memory-based systems is a Boolean circuit with flip/flop delay elements. A Boolean circuit network with internal memory is equivalent to a finite state machine [Kohavi, 1970]. Its advantage is to model a memory-based system with a well-defined number of states, and allows us to quantify memory elements by counting the number of states. This control structure is useful to agents with a small number of sensors, since the chromosome size in finite state machines is exponentially proportional to the number of sensors.

For the Santa Fe trail problem, finite state machines are encoded as an integer string as mentioned in chapter 3. The chromosome represents a state transition table in a canonical order and its initial state is 0 by default. Finite state machines were first evolved such that ants collect all the food in the grid world. The strategies in Figure 4.3 were easily obtained in a small number of generations when 8 states and 16 states were assumed. The state transition tables are reduced from the original state transitions by removing redundant state transitions. If one looks into the strategy in Figure 4.3(a), the result is almost the same as the Koza's genetic programming result in Figure 4.2, even if they are of different formats. The only difference between them is that the genetic programming result has a redundant expression `(progn2 (left) (right))` in the middle of the S-expression. Its primitive functions `progn2` and `progn3` were intentionally set for sequential actions, and these can be represented as a set of states in finite automata. Moreover, Figure 4.3(a) is not the best strategy in terms of its efficiency, and it takes 405 time steps to pick up all 89 pellets of food, while Figure 4.3(b) and Figure 4.3(c) controllers take 379 time steps and 356 time steps, respectively.

Generally, when we need a sequence of actions, $(A_1, A_2, A_3, ..., A_n)$ which cannot be reducible, it requires n states in finite automata, that is, $\lceil \log_2 n \rceil$ Boolean memory elements in switching circuit networks. Koza used a `progn` function in many applications [Koza, 1992], which corresponds to a sequence of states in finite automata.

Figure 4.2 has a different representation of memory states, but it is equivalent to an evolved FSM in Figure 4.3(a). The result indirectly implies that five internal states are sufficient to collect all 89 pellets of food. We will investigate later what is the minimal amount of memory required to achieve the task.

4.2 Experimental Issues

Many evolutionary approaches related to the artificial ant problem considered the fitness as the amount of food that the ant has eaten within a fixed number of time steps [Jefferson and Collins, 1991; Koza, 1992; Angeline, 1998; Silva et al., 1999]. They showed the potential to solve the problem, but the approaches were only demonstrated with the best case of their own fitness performance. They neglected comparing their approaches with others through performance measures. Quantitative or qualitative comparisons among strategies or control structures are much needed in evolutionary computation.

Thus, the following issues are raised for the artificial ant problem.

- In the artificial ant problem the ants are given a fixed number of time steps to explore the environment. If the maximum allowed amount of time is changed, what will happen to the performance?
- If mutation rate is changed, is it a significant factor to influence the performance?
- What kind of selection mechanism in evolutionary algorithms, linear rank selection, tournament-based selection or fitness-proportional selection, with/without elitism, will be useful to solve the artificial ant problem?
- Is it possible for a pure reactive control system to solve the artificial ant problem?
- One needs to find out how many memory states are required to solve the artificial ant problem. What is the maximum amount of food that the ant can eat with each level of memory amount? Can we obtain the solutions in a single run of evolutionary computation? How close can a given number of memory elements get to reach the most efficient strategy?

The first three questions are related to parameters in evolutionary computation. They are common problems of interest with many evolutionary approaches. They will presumably depend on what kind of tasks or applications are tested. It is suggested that both the fitness test and the effort test explained in chapter 3 should be used to compare strategies or parameter changes. It will also be useful to compare control structures. The main problems of concern here are the last two issues, related to memory. These issues will be addressed in the thesis for many problems ranging from simple to complex tasks.

The experiments for the above issues were tested for the John Muir trail and the Santa Fe trail together. A performance comparison will help us understand which problem is more difficult

to solve. It should be answered in conjunction with how the difficulty or complexity of tasks can be defined.

In the artificial ant problems, the penalty fitness function F is defined as follows:

$$F = t_A - \alpha \cdot Q_{food} \tag{4.1}$$

where t_A is the number of time steps required to find all the food, or the maximum allowed amount of time if the ant cannot eat all of them. Q_{food} is the amount of food the ant has eaten. α is a scaling coefficient and it is set to 2. This fitness function considers finding the minimum amount of time to traverse all the food cells. It will be equivalent to a well-known traveling salesman problem [Paradimitriou and Steiglitz, 1982].

In terms of computation time, the computing cost $\widetilde{C}$ can be estimated with the following formula:

$$\widetilde{C} = \delta \cdot P \cdot G \cdot T \cdot S \tag{4.2}$$

where δ is a scaling coefficient of a delay time for one robot action, P is a population size, G is the number of generations, T is the exploration time, and S is the number of trials for a genome agent in random environments. Each of the four parameters is assumed to be independent; strictly speaking, G and P, for instance, are not independent since the computing time of the mate selection process for each generation may depend on the population size. The number of trials S is related to how many trials are evaluated for one fitness calculation and it has many factors such as starting positions of a robot, the number of tested environments and repeated evaluations in random or noisy environments.

The experiments with 800 time step exploration have double[1] the computing cost of 400 time step experiments and four times that of 200 time step experiments if the same population size and the same number of generations are applied. Similarly a population size of 200 will have double the computational cost of a population size of 100. The number of generations is also a variable part in the computing cost.

[1]This may not be true in terms of computation time. When an ant succeeds in collecting all food before 800 time steps, the exploration process can instantly stop not to wait for the whole 800 time steps to complete. But we assume that the whole 800 time steps are spent for one experiment.

4.3 Decision of Parameters in Evolutionary Computation

4.3.1 How to Determine Computing Cost Parameters

From the computing cost equation 4.2, exploration time, the number of generations and population size, are variable factors in the artificial ant problem; the number of trials is fixed here. The parameters of the minimal computing cost are determined for later experiments on memory-based controllers.

– Exploration time

We are first interested in determining whether or not the maximum allowed amount of time to explore is an important factor. The question is about whether the artificial ant will find good trails more easily when the amount of exploration time is increased.

Figure 4.4 shows the fitness curve over generations with different exploration time, using genetic algorithms. Finite state machines are taken as control systems for every artificial ant experiment. Each independent experiment was run 25 times. A maximum of 16 states were allowed and 10,000 generations were applied for all experiments. The population size was 100 and a tournament-based selection method was used. In the tournament selection, the group size four is applied, the two best chromosomes breed and the two worst are replaced by new offspring. The new offspring are reproduced through crossover and mutation over the two best chromosomes. The crossover rate was 0.6 and mutation rate was 0.1 (the above tournament selection takes a half of population for elitism, and so the high mutation rate will give more chance of diversity to a new population. Various mutation rates will be investigated in section 4.3). The error bars of 95% confidence intervals are displayed with the average fitness over 25 runs by assuming that the fitness values follow t-distributions. When the amount of exploration time increased, the standard deviations of fitness values tended to decrease. Also all 89 food pellets are collected earlier by the ants. There should be a rigorous way to measure this performance. Thus, the effort test mentioned in chapter 3 is used to compare the performance with various exploration times. The performance cannot be estimated with the fitness test in this case. There is no significant difference of fitness after 10,000 generations among the various exploration time experiments.

Table B.1 (see appendix B) shows a collection of success cases from the experiments. A success is defined as the case that all 89 food pellets are collected by the ants. We are interested in how early the success is achieved. An efficient method will lead to more frequent success cases

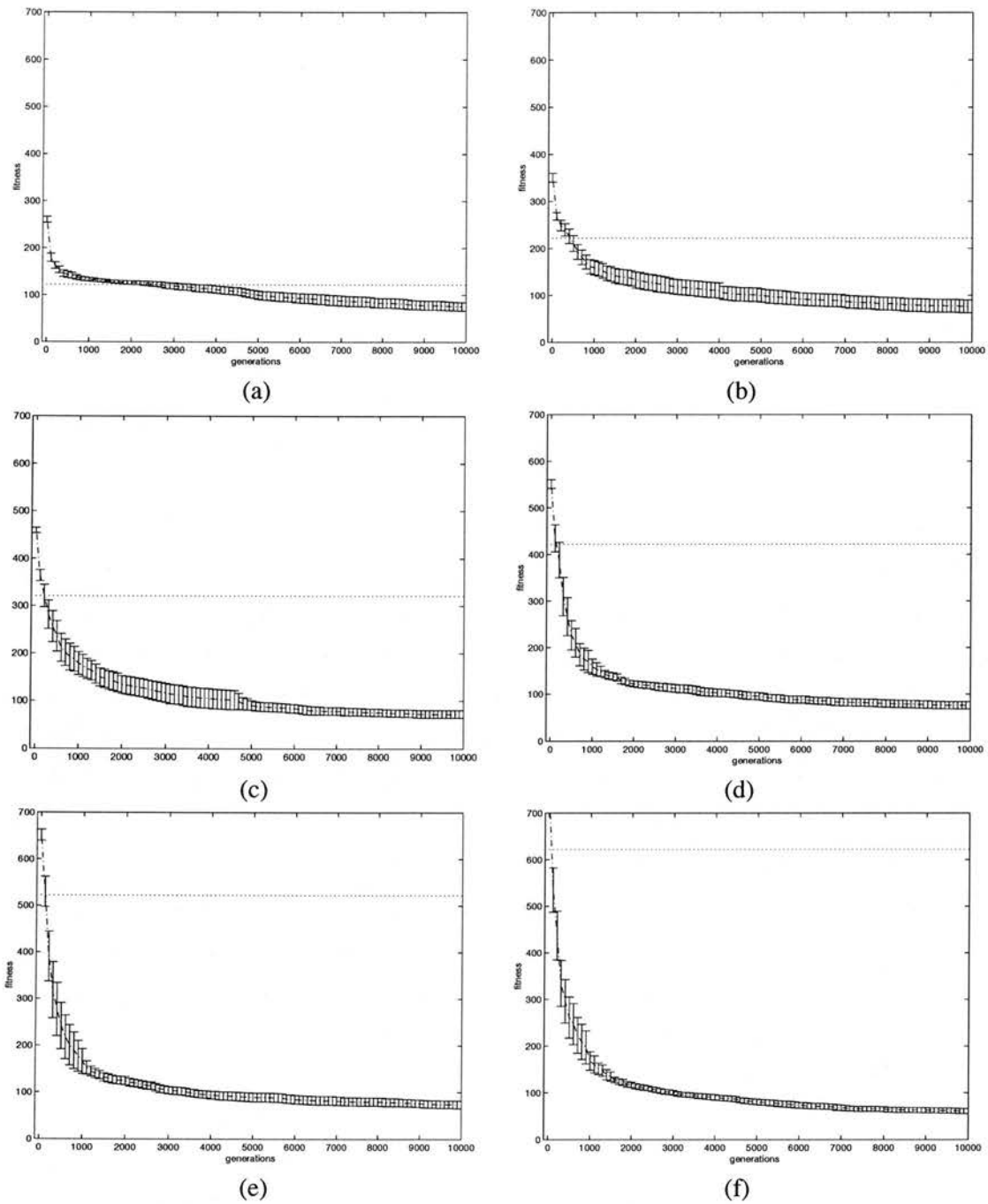


Figure 4.4: The Santa Fe trail experiments with various exploration time (dotted: the fitness that all food are collected) (a) 300 time steps (b) 400 time steps (c) 500 time steps (d) 600 time steps (e) 700 time steps (f) 800 time steps

within a smaller number of time steps. It seems that there is a tendency to find more efficient strategies when exploration time is increased. With a maximum exploration time of 800 time steps, there were 9 successful experiments among 25 experiments after 10,000 generations such that all 89 food pellets are picked up by the ants within 230 time steps. In contrast, 3 experiments were successful with 400 exploration time steps. If we assume that the success rate follows a beta distribution, the confidence interval can be estimated with equation 3.1. In these experiments, the 95% confidence interval of success rate for 800 exploration time steps is [0.202, 0.557], and the 95% confidence interval for 400 exploration time steps is [0.044, 0.302]. It is not significantly different in this case, although the performance with 800 exploration time steps is better.

If we compare experiments where all 89 food pellets are eaten within 240 time steps, 18 successes are achieved among 25 experiments for 800 exploration time steps, but only 7 successes for 400 exploration time steps. The 95% confidence intervals are [0.522, 0.856] and [0.143, 0.478], respectively. Thus, it can be said that 800 time step runs are statistically more effective than 400 time step runs, although it needs twice as much exploration time to find solutions for control strategies.

It will be a fair idea to consider how much effort is performed to achieve the solutions. This can be measured with the effort test instead of only seeing the success rate. In the evolutionary experiments, the effort cost is proportional to the computation time consumed to acquire desirable solutions and it can be represented with the notation of the computing cost in equation 4.2. Therefore, one can assume that the effort cost consists of population size, maximum exploration time, the number of generations, and the number of trials for a genome agent in random environments. In Table B.1 (see appendix B) experiments, a chromosome was tested once (there is no randomness in the artificial ant environment and it has only one starting position) and population size was 100. Only exploration time is a variable and so the 800 exploration time step run can be said to take double the effort of the 400 exploration time step run with the condition that other factors are constant in the experiments. According to the effort test, $\frac{1}{p}C$ is the average effort cost where p is a success rate and C is the computing cost for a single experiment. From the 95% confidence intervals for success probability, the effort cost can be estimated as $[\frac{1}{0.478}C, \frac{1}{0.143}C] = [2.09C, 6.99C]$ for 400 time step test and $[\frac{1}{0.856}2C, \frac{1}{0.522}2C] = [2.33C, 3.83C]$ for 800 time step test, where C is the effort cost for a single 400 time step experiment. For the effort cost, two experiments are not significantly different. However, the 400 time step test has more variance in the performance of its resulting control structures.

	Generations							
Time	$G \leq 3{,}000$		$G \leq 4{,}000$		$G \leq 5{,}000$		$G \leq 10{,}000$	
300	4	$[2.15C_3, 11.44C_3]$	6	$[1.71C_4, 6.48C_4]$	15	$[0.98C_5, 1.85C_5]$	22	$[0.78C_{10}, 1.07C_{10}]$
400	8	$[1.93C_3, 5.81C_3]$	14	$[1.36C_4, 2.70C_4]$	18	$[1.17C_5, 1.92C_5]$	23	$[1.02C_{10}, 1.33C_{10}]$
500	10	$[2.10C_3, 5.35C_3]$	15	$[1.63C_4, 3.08C_4]$	17	$[1.51C_5, 2.59C_5]$	22	$[1.31C_{10}, 1.79C_{10}]$
600	9	$[2.69C_3, 7.42C_3]$	13	$[2.14C_4, 4.49C_4]$	16	$[1.88C_5, 3.38C_5]$	22	$[1.57C_{10}, 2.15C_{10}]$
700	12	$[2.62C_3, 5.85C_3]$	17	$[2.11C_4, 3.63C_4]$	17	$[2.11C_5, 3.63C_5]$	22	$[1.82C_{10}, 2.51C_{10}]$
800	15	$[2.61C_3, 4.92C_3]$	20	$[2.20C_4, 3.29C_4]$	23	$[2.05C_5, 2.67C_5]$	24	$[2.02C_{10}, 2.49C_{10}]$

Table 4.1: Effort costs for exploration time with generations in the Santa Fe trail problem: C_i is the basic unit effort cost for a single exploration time 400 step experiment (a success within 278 time steps). Each pair of values represent the number of successes among 25 experiments and its 95% confidence interval for effort cost.

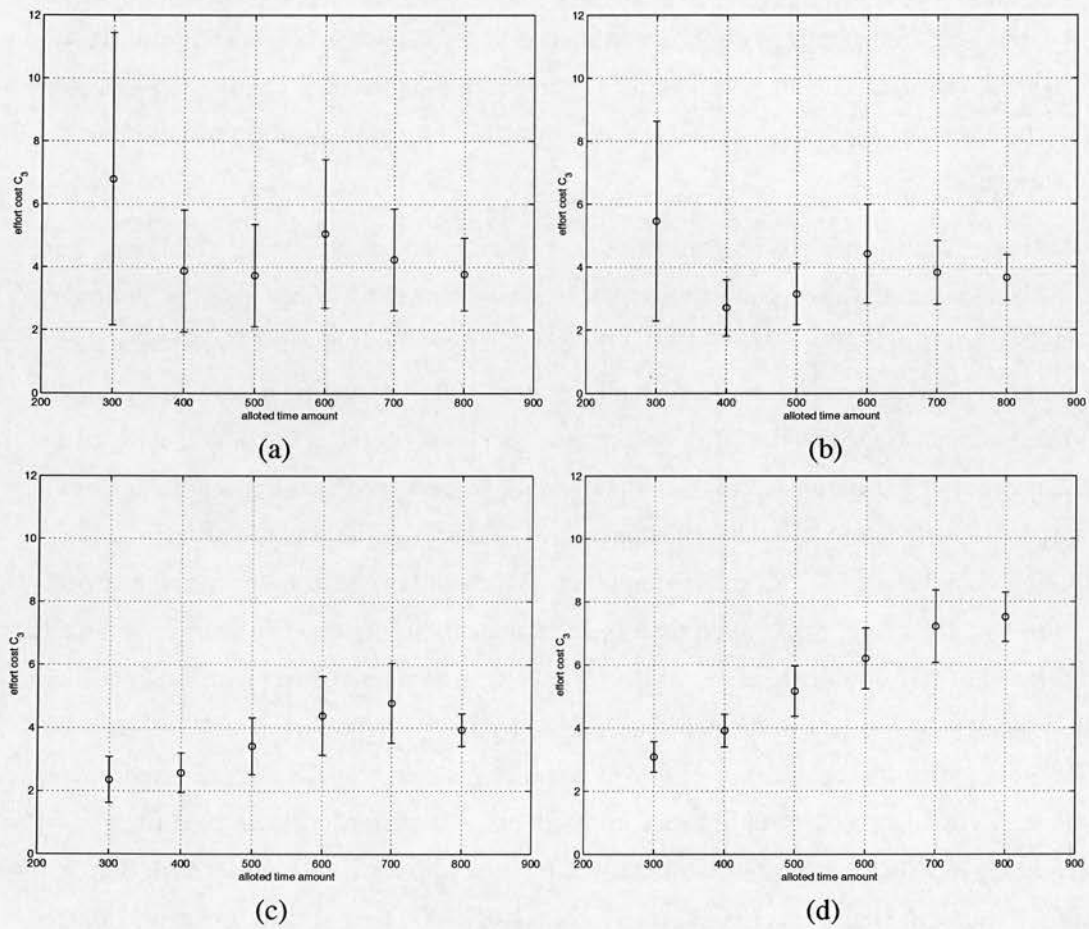


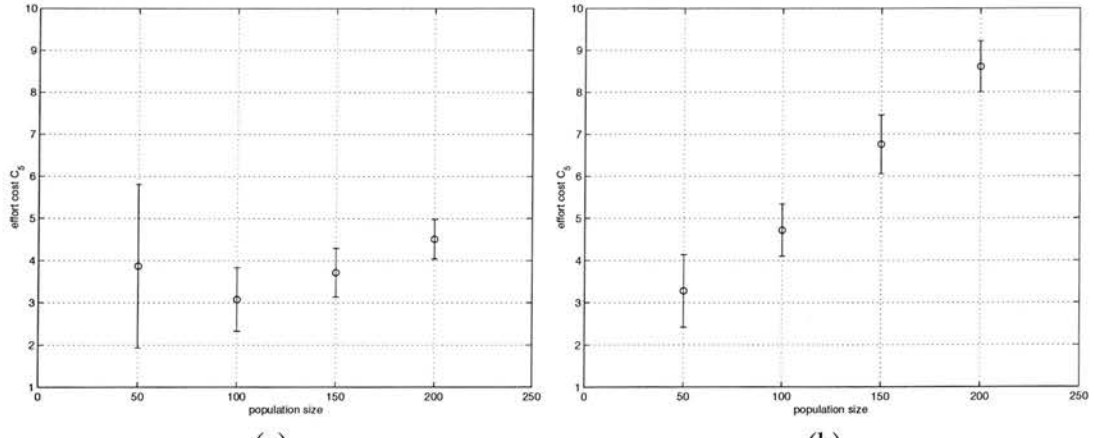
Figure 4.5: Confidence intervals for effort costs with different numbers of generations (a) 3,000 generations (b) 4,000 generations (c) 5,000 generations (d) 10,000 generations

– Number of generations

The estimated effort costs for experiments where all 89 food pellets are picked up within 278 time steps (corresponding to the fitness value 100 = 278 - 89 * 2) are given in Table 4.1. It shows experimental results with several exploration times and generations. The confidence intervals after different numbers of generations, 3000, 4000, 5000 and 10000 generations, are shown in Figure 4.5. For the effort cost of different generations, $C_4 = C_3\frac{4}{3}, C_5 = C_3\frac{5}{3}, C_{10} = C_3\frac{10}{3}$ was applied. For experiments with 3,000 and 4,000 generations, there is no significant difference in effort performance among different exploration time runs. However, experiments of 5,000 generations or more show that 300-400 time step runs are significantly better in terms of effort cost than the 700-800 time step runs. In the experiments of 300 time step runs, the effort cost has a large variance for experiments less than 5000 generations. Thus, we may recommend that 400 time step runs will economically and reliably have good solutions of fitness 100. It also means that 400 time steps are adequate to explore the environment with evolutionary algorithms.

As shown in Table B.5 (see appendix B), it seems that it is extremely difficult to collect all 89 food pellets with only 200 exploration time steps for the Santa Fe trail. Even if 20,000 generations are tested (not shown in the table), evolutionary runs could not get the solution. In Figure 4.5 (c)-(d), experiments of 10,000 generations have significantly higher levels of effort cost in many cases than those of 5,000 generations. This suggests that 10,000 generation experiments spent too much effort to reach desirable solutions of fitness 100.

– Population size



(a) (b)

Figure 4.6: Confidence interval for effort costs with various population sizes (a) 5,000 generations (b) 10,000 generations

			Generations			
Psize	fitness (time limit)		G $\leq$ 5000		G $\leq$ 10000	
50	100	(278)	8	$[1.93C_5, 5.81C_5]$	17	$[1.21C_{10}, 2.07C_{10}]$
	72	(250)	1	$[5.09C_5, 105.71C_5]$	3	$[3.32C_{10}, 22.96C_{10}]$
	62	(240)	1	$[5.09C_5, 105.71C_5]$	1	$[5.09C_{10}, 105.71C_{10}]$
100	100	(278)	18	$[2.33C_5, 3.83C_5]$	23	$[2.05C_{10}, 2.67C_{10}]$
	72	(250)	3	$[6.63C_5, 45.91C_5]$	12	$[3.00C_{10}, 6.68C_{10}]$
	62	(240)	2	$[7.96C_5, 81.77C_5]$	7	$[4.19C_{10}, 13.96C_{10}]$
	52	(230)	1	$[10.18C_5, 211.42C_5]$	3	$[6.63C_{10}, 45.91C_{10}]$
150	100	(278)	22	$[3.14C_5, 4.29C_5]$	24	$[3.03C_{10}, 3.73C_{10}]$
	72	(250)	13	$[4.28C_5, 8.99C_5]$	19	$[3.04C_{10}, 4.21C_{10}]$
	62	(240)	9	$[5.39C_5, 14.83C_5]$	15	$[3.91C_{10}, 7.39C_{10}]$
	52	(230)	9	$[5.39C_5, 14.83C_5]$	11	$[4.76C_{10}, 11.28C_{10}]$
200	100	(278)	24	$[4.04C_5, 4.98C_5]$	25	$[4.00C_{10}, 4.61C_{10}]$
	72	(250)	10	$[6.73C_5, 17.13C_5]$	23	$[4.10C_{10}, 5.34C_{10}]$
	62	(240)	6	$[9.16C_5, 34.56C_5]$	15	$[5.21C_{10}, 9.86C_{10}]$
	52	(230)	2	$[15.91C_5, 163.53C_5]$	12	$[6.00C_{10}, 13.37C_{10}]$

Table 4.2: Effort cost confidence intervals with various population size in the Santa Fe trail (C_i is a basic unit effort cost for a single experiment with population size 50 and exploration time 400 steps, $C_{10} = 2C_5$)

The effort cost is also proportional to the population size. Table B.2 (see appendix B) shows the number of successes within a given time limit for a fixed population size 50, 100, 150 and 200. The confidence intervals of effort costs are recorded in Table 4.2, using the information of success cases in Table B.2. Increasing the population size does not improve its evolutionary search. Population size 100 is significantly better in effort cost than population size 200, but there is no significant difference between population size 50 and 100 as shown in Figure 4.6. Population size 50 has a large variance at 5,000 generations. Thus, population size 100 will be a good parameter for these evolutionary experiments.

To test the same effort cost but with different parameters, three experiments are designed with double the effort cost of $(P = 100, G = 10000, T = 400)$. It is presumed that those are of the same effort cost by the equation 4.2. In Figure 4.7, experimental results are displayed with $(P = 100, G = 10000, T = 800), (P = 200, G = 10000, T = 400), (P = 100, G = 20000, T = 400)$. They are not significantly different in performance, but increasing the number of generations causes the most effect to acquire desirable solutions. It again assures us that 400 time step runs are sufficient for exploration time, because 800 time step runs are not better than the other runs with the same effort cost.

From the experiments with the Santa Fe trail, we can say that instead of increasing population size, it is slightly better to increase the number of generations if an assigned computational time for the evolutionary task is limited. It supports the idea that solutions gradually progress from old populations as the evolution process continues. An appropriate exploration time is

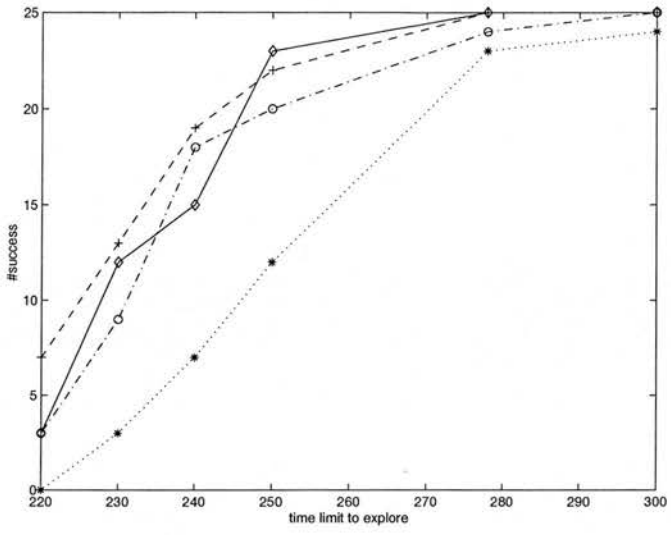


Figure 4.7: The Santa Fe trail problem results with the same effort cost ($\circ$: $(P = 100, G = 10000, T = 800)$, $\diamond$: $(P = 200, G = 10000, T = 400)$, +: $(P = 100, G = 20000, T = 400)$, *: $(P = 100, G = 10000, T = 400)$)

another important factor and if it is over a certain limit, the performance does not improve with the similar effort cost given.

4.3.2 How to Determine the Mutation Rate

The above experiments used a mutation rate of 0.1. The mutation rate is another variant in evolutionary computation, which can significantly affect the performance. Different mutation rates of 0.01, 0.02, 0.03, 0.05, 0.1 and 0.2 were tested with the same evolutionary configurations. Figure 4.8 (a)-(f) are the results with different mutation rates, but with the same population size 100, exploration time 400 steps and 10,000 generations. Tournament selection of group size four was applied to all mutation rate experiments.

The fitness distributions are not significantly different for the mutation rates ranging from 0.03 to 0.1 and they are better than mutation rate 0.01 and 0.2. Mutation rate 0.1 seems to reach solutions stably with small variance. When the effort test is applied to the fitness data as shown in Table 4.3, the rates 0.02, 0.03 and 0.05 are significantly better than the rate 0.1 for fitnesses 50 and 25, with 5000 generations. It is hard to reach low penalty fitness values with a mutation rate of 0.1. It is due to the fact that high mutation rates become close to a random search. With a mutation rate 0.2, the evolutionary computation cannot reach fitness 100 and it has a very

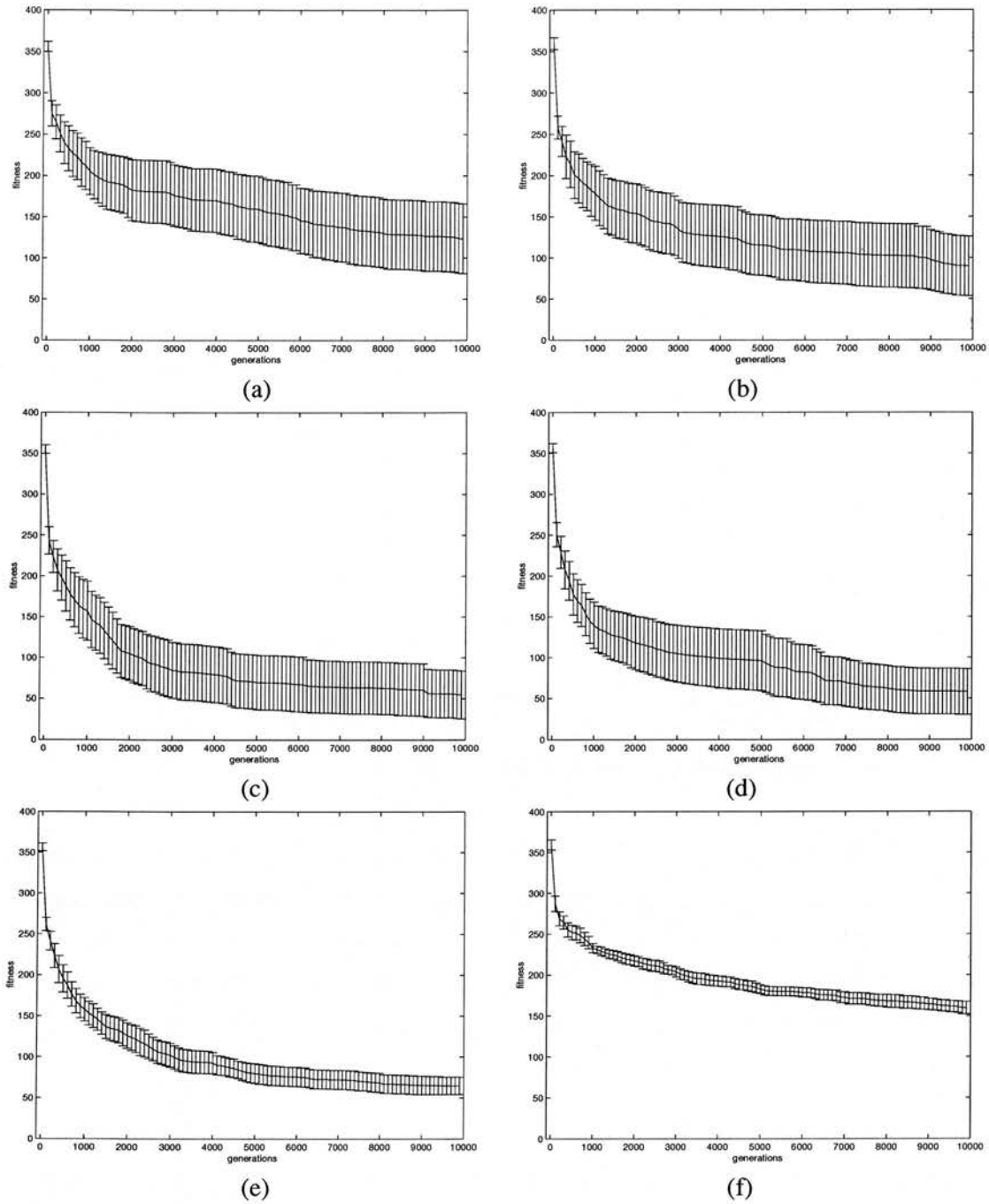


Figure 4.8: Mutation rate experiments in the Santa Fe trail problem (a) mutation rate 0.01 (b) mutation rate 0.02 (c) mutation rate 0.03 (d) mutation rate 0.05 (e) mutation rate 0.1 (f) mutation 0.2

		Generations			
mutation rate	fitness	G $\leq$ 5000		G $\leq$ 10000	
0.01	100	7	$[2.09C_5, 6.98C_5]$	13	$[1.43C_{10}, 3.00C_{10}]$
	50	5	$[2.54C_5, 11.14C_5]$	9	$[1.80C_{10}, 4.94C_{10}]$
	25	5	$[2.54C_5, 11.14C_5]$	5	$[2.54C_{10}, 11.14C_{10}]$
0.02	100	15	$[1.30C_5, 2.46C_5]$	18	$[1.17C_{10}, 1.92C_{10}]$
	50	10	$[1.68C_5, 4.28C_5]$	14	$[1.36C_{10}, 2.71C_{10}]$
	25	3	$[3.32C_5, 22.96C_5]$	9	$[1.80C_{10}, 4.94C_{10}]$
0.03	100	21	$[1.07C_5, 1.54C_5]$	21	$[1.07C_{10}, 1.54C_{10}]$
	50	17	$[1.21C_5, 2.07C_5]$	20	$[1.10C_{10}, 1.65C_{10}]$
	25	7	$[2.09C_5, 6.98C_5]$	14	$[1.36C_{10}, 2.71C_{10}]$
0.05	100	17	$[1.21C_5, 2.07C_5]$	22	$[1.05C_{10}, 1.43C_{10}]$
	50	12	$[1.50C_5, 3.34C_5]$	17	$[1.21C_{10}, 2.07C_{10}]$
	25	3	$[3.32C_5, 22.96C_5]$	8	$[1.93C_{10}, 5.81C_{10}]$
0.1	100	21	$[1.07C_5, 1.54C_5]$	23	$[1.03C_{10}, 1.34C_{10}]$
	50	1	$[5.09C_5, 105.71C_5]$	8	$[1.93C_{10}, 5.81C_{10}]$
	25	0	$[7.57C_5, --C_5]$	0	$[7.57C_{10}, --C_{10}]$
0.2	100	0	$[7.57C_5, --C_5]$	0	$[7.57C_{10}, --C_{10}]$
	50	0	$[7.57C_5, --C_5]$	0	$[7.57C_{10}, --C_{10}]$
	25	0	$[7.57C_5, --C_5]$	0	$[7.57C_{10}, --C_{10}]$

Table 4.3: Effort cost confidence intervals with various mutation rates in the Santa Fe trail: each pair represents the number of successes and confidence intervals (C_i is a basic unit of effort cost for a single experiment with population size 100 and exploration time 400 steps, $C_{10} = 2C_5$)

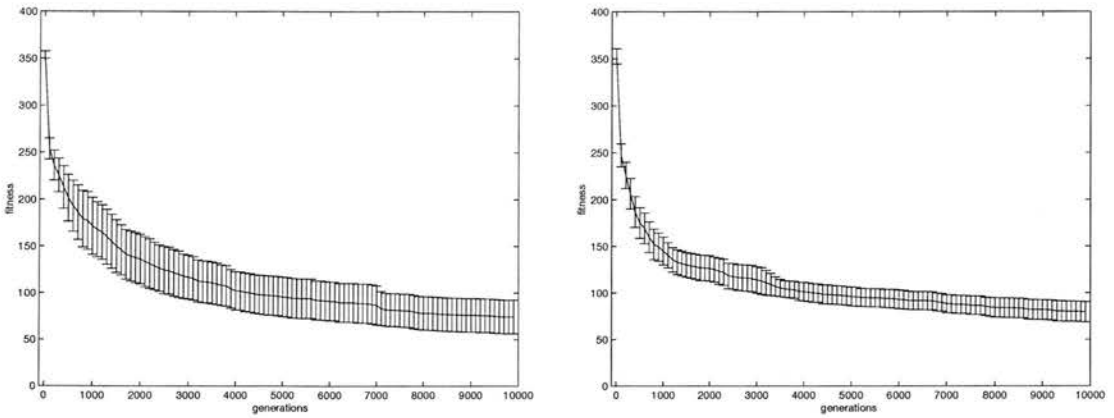
high effort cost[2]. From the table, one can determine that the desirable mutation rate is 0.03 for the experiments. It is worth noting that low mutation rates help to obtain very good solutions below fitness 25, but they have a few bad results out of 25 runs, causing large variances as in Figure 4.8(a)-(d).

The effort cost of experiments for 10,000 generations doubles that of 5,000 generations. If more successes are obtained with many generations, the effort cost will not increase so much. If the fitness of solutions does not improve significantly, many generations will waste computation time. For example, the effort cost to achieve fitness 100 does not improve much while continuing the evolutionary processes after 5,000 generations with a mutation rate 0.03, 0.05 and 0.1. Thus we can conclude that 5,000 generations are sufficient for fitness 100.

4.3.3 How to Determine the Selection Mechanism

The selection process in evolutionary computation handles how to choose the best chromosomes for breeding. The three main ones are: Among the many kinds of selection mechanisms in evolutionary algorithms, fitness-proportional, rank-based and tournament-based. The

[2]No success is found with mutation rate 0.2, but its estimated success rate is not zero, since it is assumed that sampling has taken over unknown probabilistic distributions.



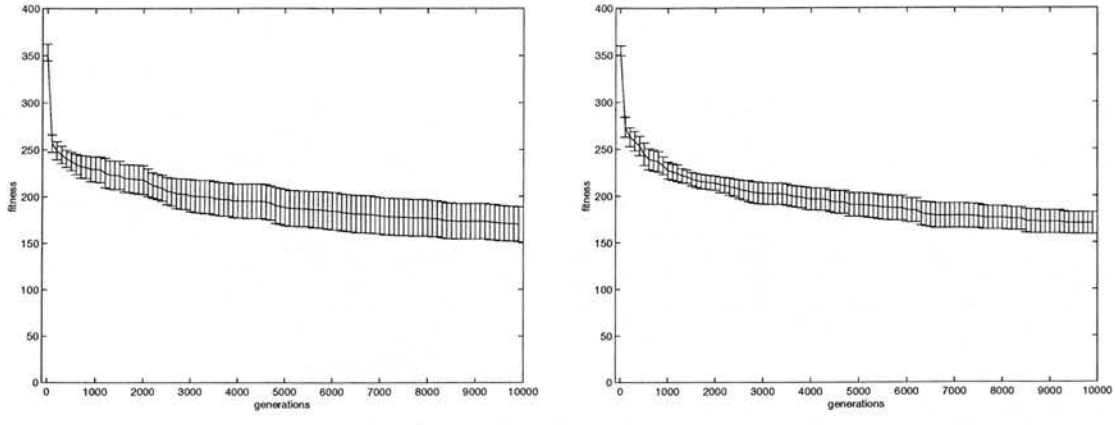
(a) (b)

Figure 4.9: Fitness-proportional selection experiments in the Santa Fe trail problem (a) mutation rate 0.03 (b) mutation rate 0.1

fitness-proportional method, often using roulette wheel selection, gives higher probability to better fitness chromosomes. The probability is proportional to the fitness value. The rank-based method first sorts a population by fitness and gives higher probability to higher rank chromosomes in the population. The probability is based on relative rank in the population and so it is dynamically changed even for the same fitness chromosome, according to the fitness distribution of chromosomes in the population. The tournament-based method can be seen as a local rank-based method. A group of individual chromosomes are first randomly chosen in a population and the selection of good chromosomes in a group follows the rank-based method. The best chromosomes in the group breed with high probability.

The best selection mechanism for the Santa Fe trail problem can be determined using the same process to decide an appropriate mutation rate. The experiments used a population size of 100, mutation rates 0.1 and 0.03, a crossover rate 0.6, a exploration time 400 steps, and 10,000 generations. A maximum of 16 states were allowed for the expansion of states. In the experiments, the fitness-proportional and rank-based methods use elitism to keep the best chromosomes from an old population. This elitism strategy significantly influences the performance. In the tournament selection, the group size four is applied, the two best chromosomes breed and the two worst are replaced by new offspring. They are reproduced through crossover and mutation over the two best chromosomes.

Three different selection methods were applied to the Santa Fe trail problem with two different mutation rates 0.1 and 0.03. The tournament selection method was already tested with various mutation rates as shown in Figure 4.8. Figure 4.9 shows the result of the fitness-proportional



(a) (b)

Figure 4.10: Rank-based selection experiments in the Santa Fe trail problem (a) mutation rate 0.03 (b) mutation rate 0.1

			Generations			
selection	mutation rate	fitness	G $\leq$ 5000		G $\leq$ 10000	
fitness-proportional	0.1	100	14	$[1.36C_5, 2.71C_5]$	19	$[1.13C_{10}, 1.77C_{10}]$
		50	0	$[3.68C_5, --C_5]$	3	$[1.66C_{10}, 11.46C_{10}]$
fitness-proportional	0.03	100	17	$[1.21C_5, 2.07C_5]$	20	$[1.10C_{10}, 1.65C_{10}]$
		50	2	$[1.98C_5, 20.44C_5]$	9	$[0.90C_{10}, 2.47C_{10}]$
tournament-based	0.1	100	21	$[1.07C_5, 1.54C_5]$	23	$[1.03C_{10}, 1.34C_{10}]$
		50	1	$[2.55C_5, 52.85C_5]$	8	$[0.97C_{10}, 2.91C_{10}]$
tournament-based	0.03	100	21	$[1.07C_5, 1.54C_5]$	21	$[1.07C_{10}, 1.54C_{10}]$
		50	17	$[0.60C_5, 1.04C_5]$	20	$[0.55C_{10}, 0.83C_{10}]$
rank-based	0.1	100	0	$[7.57C_5, --C_5]$	0	$[7.57C_{10}, --C_{10}]$
		50	0	$[3.68C_5, --C_5]$	0	$[3.68C_{10}, --C_{10}]$
rank-based	0.03	100	0	$[7.57C_5, --C_5]$	0	$[7.57C_{10}, --C_{10}]$
		50	0	$[3.68C_5, --C_5]$	0	$[3.68C_{10}, --C_{10}]$

Table 4.4: Effort cost confidence intervals with selection mechanisms in the Santa Fe trail: each pair represents the number of successes and confidence interval (C_i is a basic unit effort cost for a single experiment with population size 100 and exploration time 400 steps, $C_{10} = 2C_5$)

method. It has a similar performance to a tournament selection with mutation rate 0.1 in Figure 4.8(e). Figure 4.10 shows rank-based selection and it has a much worse performance. Each configuration experiment was repeated 25 times with the same population size of 100. Using a fitness test based on t-distribution on mutation rate 0.03 and 0.1, the fitness-proportional method and tournament selection method are significantly better than the rank-based selection method with confidence interval 95%.

The experimental results with two different population sizes 50 and 100 are shown in Table B.3 (see appendix B). By observing success cases among 25 runs for each configuration, the effort test result was built in Table 4.4. According to the effort test result, tournament selection has better performance than the other methods. Fitness-proportional selection is not significantly

different from tournament selection with population size 100, even though it has a slightly lower performance. Especially with a small-size population of 50, the tournament-based selection is significantly better than any other selection method. Mutation rate is an important factor in influencing the performance in both tournament selection and fitness-proportional selection. In this experiment, what is the best mutation rate has not been searched for all the selection methods. To compare selection methods more strictly, the best mutation rate, crossover rate and other parameters need to be investigated together.

4.3.4 Summary of Evolutionary Parameters

This section was devoted to determine various evolutionary parameters for the Santa Fe trail problem. First, the fitness test and effort test were applied to the computing cost parameters including exploration time, the number of generations and population size. 400 exploration time steps could reach fitness 62 (240 time steps) with a reasonable effort cost. 5000 generations was sufficient to reach fitness 100. Population size 100 was appropriate for various fitness levels. Several mutation rates were also tested and mutation rate 0.03 could obtain good solutions of low penalty fitness, although it has large variances. When three selection mechanisms, fitness-proportional, rank-based and tournament-based methods, were compared, the tournament-based selection method was better with small-sized population 50 than the other methods. Fitness-proportional method had similar performance to tournament-based method with population size 100.

4.4 Decision of Memory States

One of the main interests is how many memory states are required for ants to traverse all the food cells. This should also be addressed together with the question that a given time limit of exploration may influence the number of memory states.

4.4.1 Fitness Test for Memory Analysis

We have run many experiments over a different number of memory states. To quantify memory amount, finite state machines are applied to the artificial ant problem. For each different state machine, 25 independent runs are repeated and their fitness distribution is considered for analysis of memory states. From the results of parameter decision experiments, a population

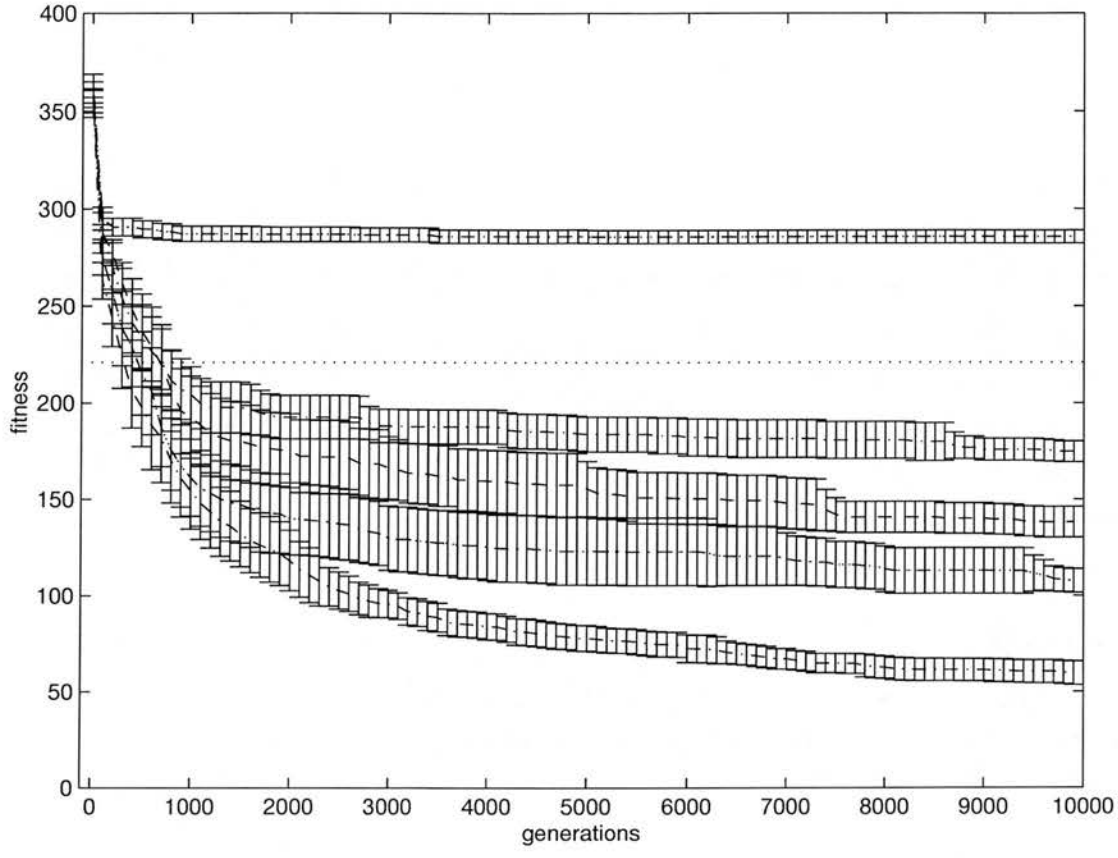


Figure 4.11: The Santa Fe trail experiments with a different number of memory states; comparison for 4 states, 5 states, 6 states, 7 states and 12 states (dotted: the fitness where all food is collected)

size of 100 and 400 time steps were employed for ant exploration. Tournament selection of size four was adopted with mutation rate[3] 0.1 and crossover rate 0.6.

The memory requirement for the task can be determined by the average fitness of multiple runs, that is, the fitness test. Each number of memory state has its own fitness curve with 95% confidence intervals and it will be checked if the average fitness can reach the boundary of desirable performance. The result is displayed in Figure B.1 (see appendix B). According to the figures, four memory states or less[4] are inadequate for the artificial ant problem. Thus, memoryless reactive controllers fail to cover all the food. To cover all 89 food pellets, at least five states are needed. In this case, without the help of the effort test, it is clear that fives states

[3]Mutation rate 0.03 has better performance than 0.1, but it has a large variance in the fitness distribution. Thus only the results for mutation rate 0.1 will be displayed.

[4]Results of memory states from one to three are not shown, but their performances are lower than four-state performance.

or three register bits are required for this problem. The result of genetic programming in Figure 4.2 by Koza [1992], which used five memory states if an redundant expression is removed, is confirmed with the analysis of memory requirements by FSM experiments.

We still have a question of how many memory states are required if a time limit is given for ants' successful traversals, or whether or not ants can collect all 89 food pellets within the time limit. One can guess that the shorter time limit will lead to more memory states to record environmental states. In the previous experiments, maximum 16 states were allowed for memory space. Then more memory space, maximum 20 states, was used to find efficient controllers, and the same set of evolutionary parameters for exploration time, population size and mutation rate were applied. However, the search for new evolutionary parameters may be required[5]. We assumed that the evolutionary parameters determined in the previous experiment would be independent of the amount of memory.

Through evolution the most efficient controller to cover all food pellets with the smallest time steps was a 173 time step controller with fitness -5 (= 173- 89*2) in the experiments as shown in Figure 4.12. This is comparable to the optimal performance of 165 time step, in which case the controller memorizes the entire trail completely.

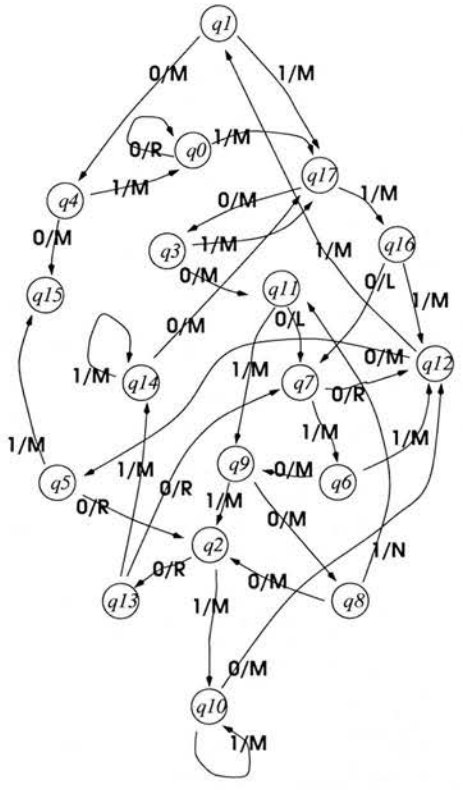
From Figure B.1 (see appendix B), the very low fitness 100 (278 time steps for time limit) can be rarely reached with less than eight states, while more than four states could succeed to cover all the food on the grid. The fitness distributions gradually improve up until 11 states. After 11 states, increasing the number of states does not improve performance. Instead they show small variances when compared to 10 or 11 states. There is a hierarchy of fitness performance in terms of the number of states as shown in Figure 4.11. Better fitness implies that ants explore the trail environment more efficiently and it reduces the number of time steps needed to collect all the food. Thus, more states have more ability to explore the environment within a short time limit. They memorize the environmental states and trigger the next actions promptly by sequentializing a series of actions.

The number of states in finite state machines (FSM) in the experiments is not exactly the same as the number of states that the evolved controllers use for exploration. It allows a maximum limit over the number of finite states. The Hopcroft algorithm can minimize the number of finite states for a given finite state machine [Hopcroft and Ullman, 1979]. The Hopcroft algorithm

[5]Maximally allowed memory amount changes the chromosome length. It may influence evolutionary parameters.

state	input 0	input 1
q_0	q_{17}, R	q_0, M
q_1	q_4, M	q_{17}, M
q_2	q_{13}, R	q_{10}, M
q_3	q_{11}, M	q_{17}, M
q_4	q_{15}, M	q_0, M
q_5	q_2, R	q_{15}, M
q_6	q_9, M	q_{12}, M
q_7	q_{12}, R	q_6, M
q_8	q_2, M	q_{11}, N
q_9	q_8, M	q_2, M
q_{10}	q_{12}, M	q_{10}, M
q_{11}	q_7, L	q_9, M
q_{12}	q_5, M	q_1, M
q_{13}	q_7, R	q_{14}, M
q_{14}	q_{17}, M	q_{14}, M
q_{15}	q_{13}, M	q_{17}, M
q_{16}	q_7, L	q_{12}, M
q_{17}	q_3, M	q_{16}, M

(a)



(b)

Figure 4.12: The best evolved control structure FSM in the Santa Fe trail problem (a) transition table (b) state diagram (input 1: food-ahead, input 0: no food ahead, output set is L (turn left), R(turn right), M (move forward), N (no-operation))

removes redundant states and combines separate but equivalent[6] states into one state. When the algorithm is applied to the finite state machines of evolved controllers, the controllers did not fully use the given search space and several states turned out to be redundant. As a result, the control structures evolved are not of compact representation.

Figure 4.13 shows the histogram of the minimized number of states for evolved controllers. The evolved controllers when minimized, with maximum states ranging from 15 to 20 states, are collected. The average number of states was 10.77 and it is consistent with the result that more than 10 memory states has stable fitness curves for the Santa Fe trail experiments in Figure B.1 (see appendix B). For all the experiments ranging from 4 to 20 states, controllers that do not use all memory states are often evolved in spite of a given maximum memory limit. If the memory amount is over a limit value, e.g. 11 states, most evolved controllers have spurious states. Normally evolving controllers need room with more than the required memory amount. More

[6]Equivalent states are defined as states with the same state transitions and the same outputs

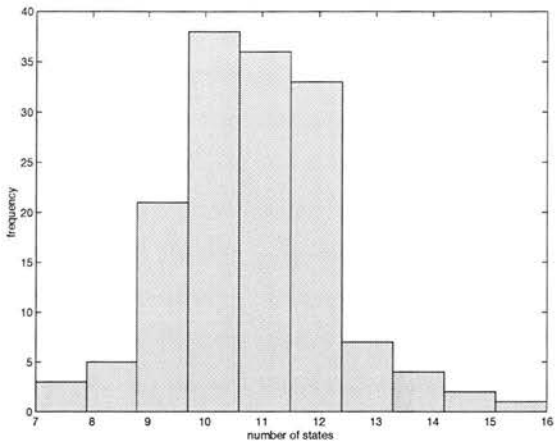


Figure 4.13: Distribution of the minimized number of finite states over the best controllers in the Santa Fe trail (the best controllers with more than 14 states)

free space in the chromosome representation can give a chance of developing desirable state transitions easily. obtaining compact representations of controllers directly through evolution.

To find out an appropriate number of states for a given exploration time limit, experiments with a fixed number of states should be repeated many times as described above. A few runs may not guarantee that a given number of states in the FSM can reach solutions, and one cannot guess in advance how many states are needed to acquire desirable agent behaviors. It requires much effort and time for the analysis of memory states and their relation with agent behaviors. For the Santa Fe trail problem, each of the various memory states, ranging from 4 states to 19 states, had 25 runs for its fitness distribution. A total of $17 \cdot 25 = 425$ runs were performed to analyze the memory states.

4.4.2 Pareto Optimization For Memory Analysis

An evolutionary multiobjective optimization (EMO) approach is developed so that a single run can show the best performance for each number of memory states. The method is related to the optimization of two objectives, memory structure and behavior fitness through evolutionary algorithms. Thus, it belongs to the field of multiobjective optimization problems. It is proposed that Pareto optimization to search for both the minimal memory structure and the best task behavior should be applied to control systems. Memory hierarchy is defined as the number of states in finite state machines. If we know what is the limit of the best behavior fitness for each number of states, we can easily explain the behavior fitness hierarchy in terms of memory. EMO approach will enable us to determine the complexity of behaviors in a single

configuration experiment, while evolutionary computation with only one objective, behavior fitness, requires separate independent experiments to differentiate memory capacity.

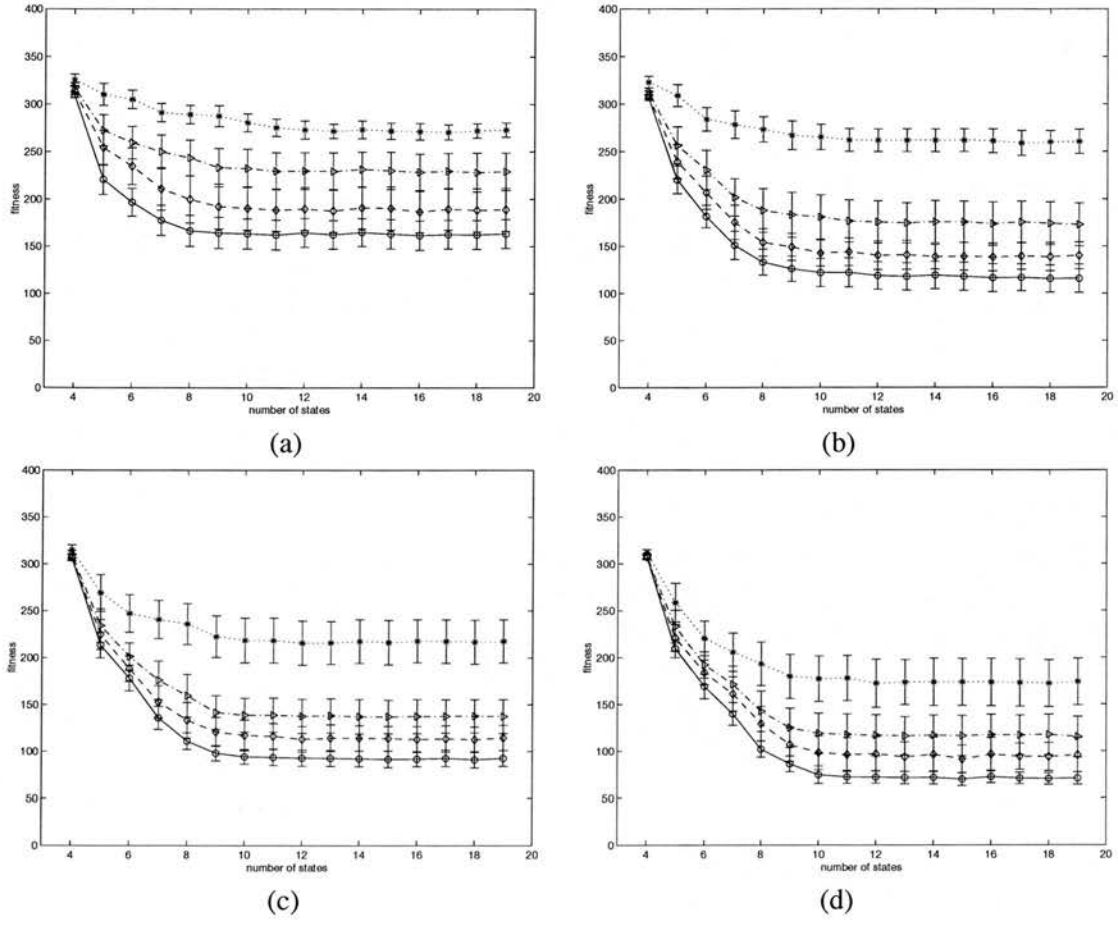


Figure 4.14: Pareto optimization without elitism in Santa Fe trail problem (a) population size 25 (b) population size 50 (c) population size 100 (d) population size 200 (dotted: 1,000 generations, dotdashed: 5,000 generations, dashed: 10,000 generations, solid: 20,000 generations)

Figure 4.14 shows the results of Pareto optimization for both control structure and control performance. In this experiment, tournament selection of group size four without elitism strategy was applied. Each chromosome should be ranked in terms of Pareto distribution and its rank is determined by the number of dominating vectors – the details are described in chapter 3. Multiobjective optimization for both the number of states and the trail fitness was run 25 times. Equation 4.1 was applied for the trail fitness. Mutation rate 0.03 and crossover rate 0.6 were used. The best fitness results with 95% confidence intervals are displayed after 1000, 5000, 10000 and 20000 generations; the error bars in Figure 4.14 show 95% confidence intervals of the t-distribution. They have concave curves for memory states versus behavior fitness at a fixed number of generations and as the evolutionary process continues, the curves gradually

move to the inner side towards achieving better fitness. Increasing the population size improves the performance and it reaches low fitness earlier during the same generations. However, unlike the previous experiments with a fixed number of finite states, the method could rarely reach under fitness 50.

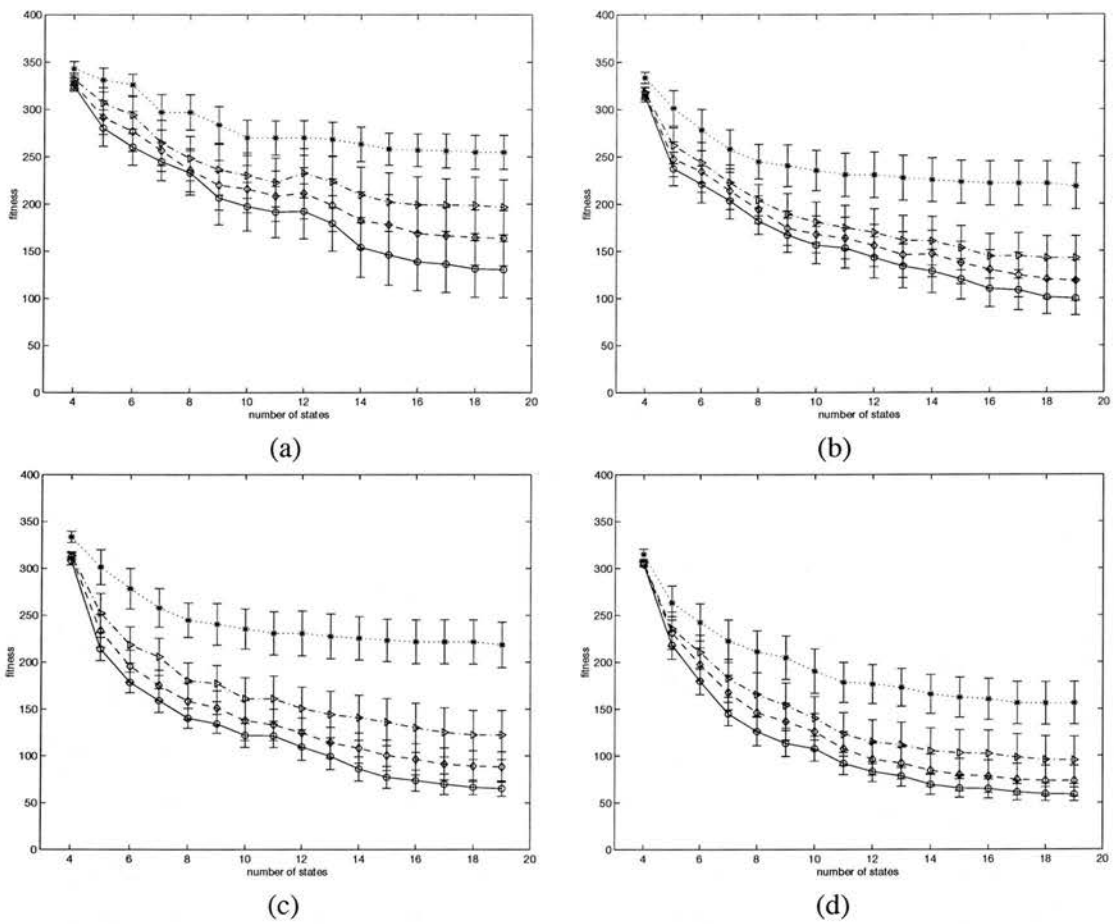


Figure 4.15: Pareto optimization with fitness elitism in the Santa Fe trail problem (a) population size 25 (b) population size 50 (c) population size 100 (d) population size 200 (dotted: 1,000 generations, dotdashed: 5,000 generations, dashed: 10,000 generations, solid: 20,000 generations)

Elitism is quite essential to obtain good solutions even in Pareto optimization. It keeps previous best chromosomes and manages gradual improvements of population fitness. A *fitness elitism* was first applied to the Pareto optimization. The individual with the best trail fitness, regardless of the number of states, is preserved for the next generation. It helps find solutions of low trail fitness. It tends to deepen its search with large-sized memory as shown in Figure 4.15. Fitness elitism often fails to find good solutions for a small number of states, because the elitism of the best trail fitness gives a biased direction towards the trail fitness in Pareto optimization and

neglects the search for the minimal memory states.

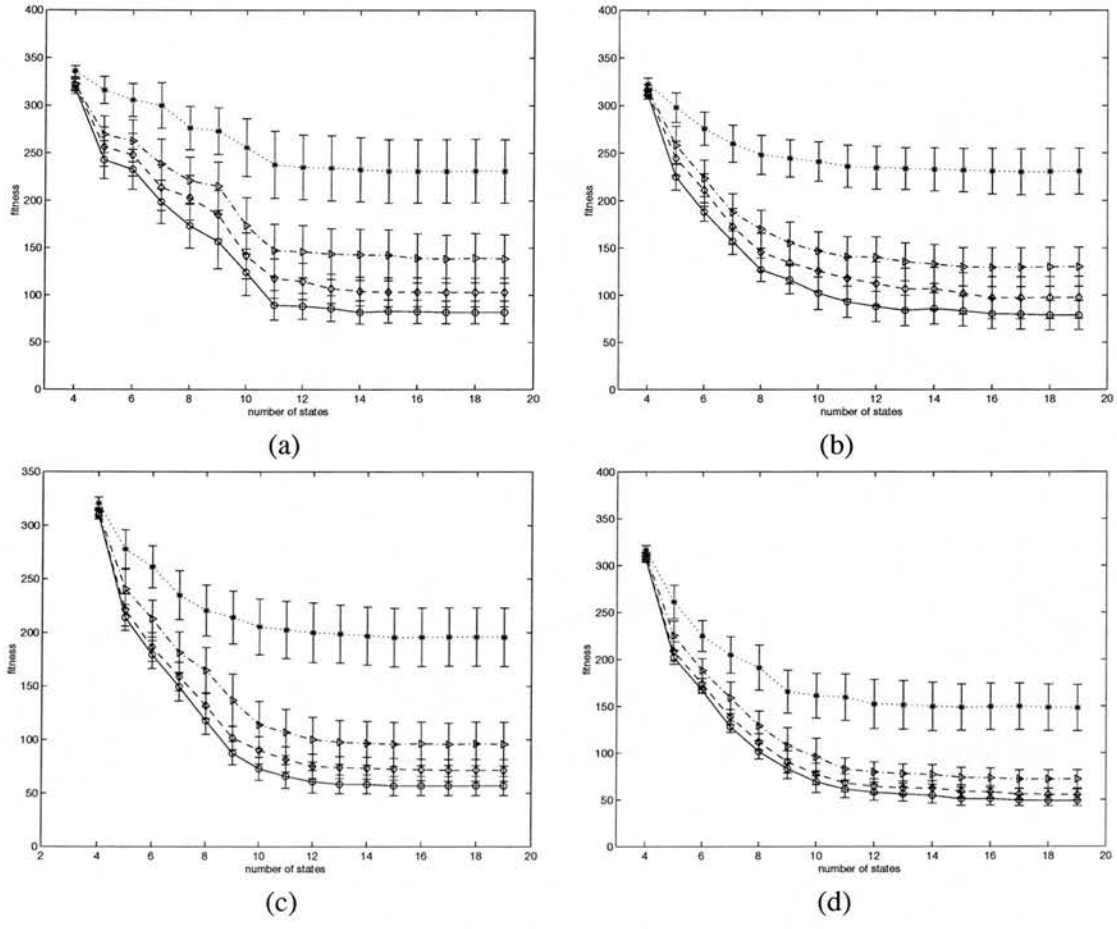


Figure 4.16: Pareto optimization with N-state elitism in the Santa Fe trail problem (a) population size 25 (b) population size 50 (c) population size 100 (d) population size 200 (dotted: 1,000 generations, dotdashed: 5,000 generations, dashed: 10,000 generations, solid: 20,000 generations)

Thus, it is proposed that a diverse state elitism, that is, a uniform distribution of elitism over states should be used in the Pareto optimization. It will be called *N-state elitism.* A pool of the best solutions, in terms of Pareto optimization for (the number of states versus trail fitness), are maintained for elitism. They are the chromosome strings with the best trail fitness for each number of state machines. A state counter is randomly chosen and then the best FSM chromosome with as many states as the state counter is selected among the pool of the best solutions for elitism in every generation. The best FSM will be inserted into the new population. This method will help to search for the best solution for each number of states. Figure 4.16 shows the results of the N-state elitism method. Better concave curves over (fitness versus memory size) are produced. Compared with the experiments shown in Figure B.1, this

parameters		Generations			
method	fitness	G $\leq$ 10,000		G $\leq$ 20,000	
no elitism	100	13	$[10.38C_1, 29.17C_1]$	74	$[2.54C_2, 3.69C_2]$
	50	0	$[70.87C_1, --C_1]$	9	$[15.22C_2, 52.94C_2]$
fitness elitism	100	54	$[3.33C_1, 5.29C_1]$	71	$[2.46C_2, 4.39C_2]$
	50	3	$[26.11C_1, 206.61C_1]$	8	$[14.59C_2, 54.43C_2]$
N-state elitism	100	136	$[1.50C_1, 1.86C_1]$	165	$[1.27C_2, 1.48C_2]$
	50	8	$[14.59C_1, 54.43C_1]$	43	$[4.04C_2, 6.89C_2]$

(a)

		Generations			
selection method	fitness	G $\leq$ 10,000		G $\leq$ 20,000	
no elitism	100	86	$[4.47C_1, 6.23C_1]$	160	$[2.61C_2, 3.08C_2]$
	50	0	$[141.74C_1, --C_1]$	0	$[141.74C_2, --C_2]$
fitness elitism	100	130	$[3.12C_1, 3.90C_1]$	156	$[2.67C_2, 3.17C_2]$
	50	3	$[52.22C_1, 413.22C_1]$	29	$[11.17C_2, 21.88C_2]$
N-state elitism	100	187	$[2.29C_1, 2.57C_1]$	206	$[2.12C_2, 2.29C_2]$
	50	34	$[9.81C_1, 18.13C_1]$	99	$[3.95C_2, 5.31C_2]$

(b)

Table 4.5: Effort test for selection method in the Santa Fe trail problem (a) population size 50 (b) population size 100 (each pair represents the number of successes and confidence interval, C_i is a basic unit effort cost for a single experiment with population size 50 and exploration time 400 steps, $C_2 = 2C_1$)

method shows quite similar performance results as experiments of fixed state machines after running 20,000 generations.

Figure 4.16 shows the average trail fitness monotonically decreasing in terms of the number of states. One can determine what is the appropriate number of memory states for a given trail fitness. From the figure result, The figure shows that more than 10 states gives a stable fitness distribution and this is consistent with the experiments in Figure B.1 (see appendix B). Figure B.2 shows the Pareto distribution of the best case controllers with 25 runs. When the population size is increased, the performance is improved. Although the elitism strategy is employed, selection by tournament-based ranking is mainly biased towards the small-sized state machines. The small-sized state machines have a much higher chance to reproduce new offspring, since small-sized state controllers with the same trail fitness have priority over large-sized states. It helps to optimize the size of memory elements for a given trail fitness and thus it will try to find compact representations of controllers without redundant memory states.

Table 4.5 shows the effort test over three Pareto experiments with no elitism, fitness elitism, and N-state elitism. Each experiment was run 25 times to sample fitness results. The number of successes is counted for more than 10 states machines, ranging from 11 states to 19 states; we assumed independent experimental runs for each number of states. The total number of

possible successes will be 225 for each experiment (= 9*25); only 25 trial runs are used for the whole configuration. N-state elitism is significantly better than the other two methods by comparing the effort costs. This fact can also be proved by the fitness test in Figure 4.14-4.16. With N-state elitism for the fitness 100, population size 50 is significantly better in effort cost than population size 100, and 10,000 generations are better than 20,000 generations. However, for fitness 50, population 100 is slightly better than population size 50 and similarly 20,000 generations are slightly better than 10,000 generations. This says that for solutions as good as fitness 50, much evolutionary time is required.

4.5 John Muir Trail

In the John Muir trail problem, ant agents have the same style of sensors and motor actions. Even though it is known that it is seemingly easier than the Santa Fe trail problem, it will be checked with experimental results for memory analysis.

Jefferson and Collins [1991] showed in their experiments, with the John Muir trail shown in Figure 4.1(a), that 81-scoring[7] finite state machines can be evolved typically within 20 generations with a population of 65,536 chromosomes. The $20 \cdot 65536 \approx 10^6$ ants were evaluated while evolving. In our experiments, 2,000 generations with a population of 100 were sufficient to obtain 81 scores for every independent experiment as shown in Table B.5 (see appendix B). Tournament selection with group size four was applied, and the best half of the chromosomes in a population were reproduced. Thus, a total of $2000 \cdot 100/2 + 50 \approx 10^5$ ants were tested to find desirable solutions. In addition, we succeeded in evolving a finite state machine that had a perfect score 89 within 167 time steps. The John Muir trail results of success cases within a given time limit are shown in Table B.4. The results show that the effort cost of exploration time 300 steps is significantly smaller than that of exploration time 700 steps. From Table B.4, 300 time steps are sufficient for exploration in the John Muir trail. As in the Santa Fe trail, the John Muir trail requires at least five states to collect all the food. Both trail problems have the complexity of three bit registers.

Evolving controllers in the John Muir trail can be compared with the Santa Fe trail, using the same evolutionary parameters (see Table B.1 and Table B.4 – appendix B). A success is defined as the case of when all 89 food pellets are collected within a given time limit. It is shown that the former trail needs less time for a success and as well has more success cases with the same

[7]The score fitness was defined as the amount of food to be collected by an ant agent.

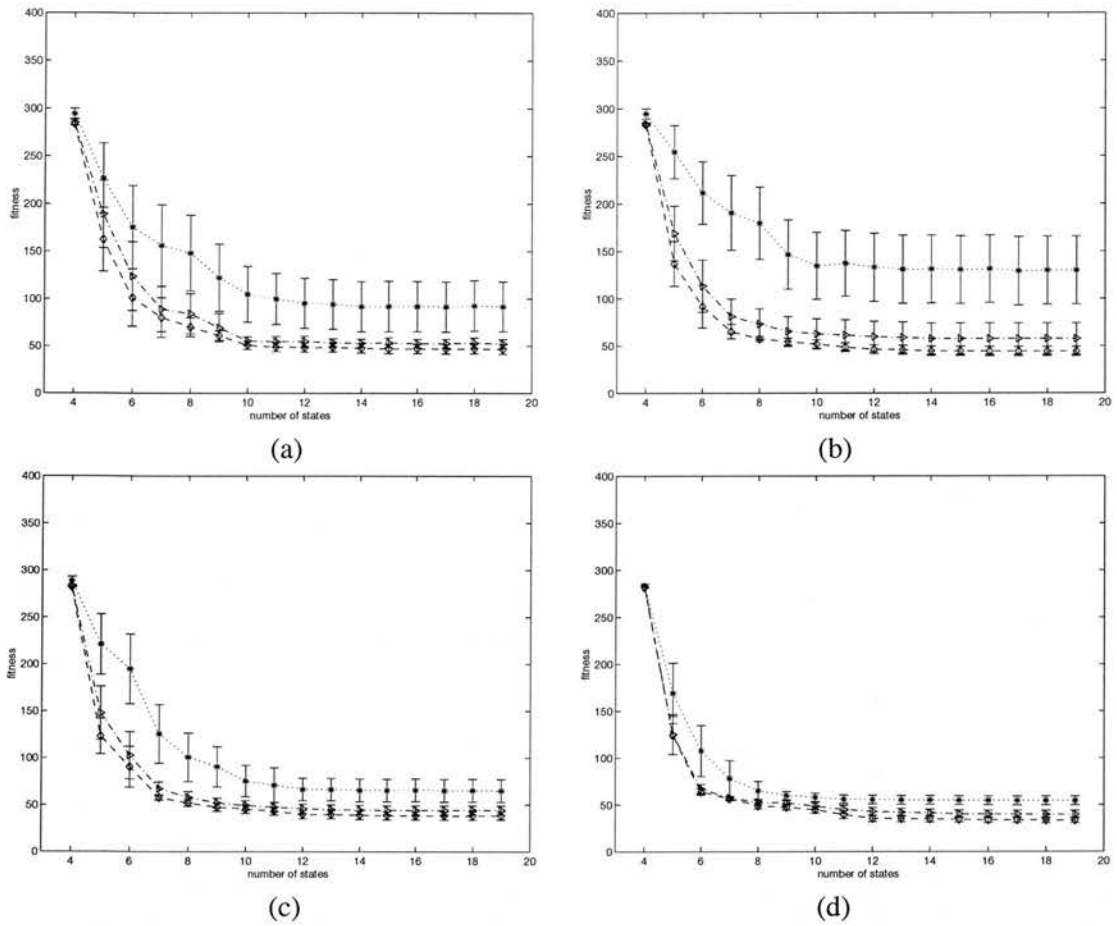


Figure 4.17: Pareto optimization with N-state elitism in the John Muir trail problem (a) population size 25 (b) population size 50 (c) population size 100 (d) population size 200 (dotted: 1,000 generations, dotdashed: 5,000 generations, dashed: 10,000 generations)

fitness or within a time limit bound. It is also true for collecting less than 89 food cells as shown in Table B.5. Figure 4.17 shows the result of Pareto optimization for memory structure and trail fitness in the John Muir trail. The performance of population size 25 is similar to that of a larger size population. Experimental results with 10,000 generations in the John Muir trail have as good a performance as 20,000 generations in the Santa Fe trail (compare Figure 4.16 and Figure 4.17). To get under fitness 100 in Santa Fe trail, nine states are needed as shown in Figure 4.16. In the John Muir trail, six states are enough to reach under fitness 100 and eleven states under fitness 50. Thus, we can conclude that the Santa Fe trail is a more difficult trail. As a result, the experiments support the intuitive idea[8] that Sante Fe trail problem is more difficult than John Muir trail problem.

[8]Koza [1992] mentioned this difficulty without measuring performance.

4.6 Summary

The artificial ant problem has one binary sensor and one environmental configuration. Relatively many memory states are required to remember environmental features. Its performance is measured by how many time steps are required for ant agents to collect all food on a trail.

Both a fitness and an effort test were applied to determine desirable evolutionary parameters or strategies. They proved valuable in the decision of mutation rate and selection mechanisms as well as parameters related to computing costs, population size, the number of generations and exploration time. In many cases, the effort test was useful where the fitness test could not discriminate the performance. Regarding selection mechanisms, tournament-based selection was better than other methods, especially in the Santa Fe trail problem.

The performance evaluation methods can be applied to decide many kinds of evolutionary parameters or test the difference of two distinct methodologies. Evolutionary computation is stochastic and a few evolutionary runs are not sufficient to compare various parameters or control strategies. The tests based on sampling theory are quite useful and they have much potential for application.

To develop more efficient strategies in the artificial ant problem, agents need more memory states according to the memory analysis and also there is a limit level of memory over which the performance is rarely improved. The evolutionary multiobjective optimization (EMO) approach is presented to obtain the minimal expression of the best controllers. It can produce compact representations of controllers in terms of memory requirement and also reduce the computation time required in the fitness test. Elitism was an important technique to accelerate convergence to solutions even in the EMO approach. N-state elitism, maintaining a pool of the best trail fitness for each number of states, showed the best performance among three EMO methods, fitness elitism, N-state elitism and no elitism. The N-state elitism EMO approach will be applied to various tasks in the following chapters.

Two artificial ant problems, Santa Fe trail problem and John Muir trail problem, requires five memory states from the memory analysis. More memory space and more efficient strategy allows ants to memorize the environment. It may be different from general purpose controllers. For example, the best evolved controller in Santa Fe trail with 18 internal states shows bad performance in John Muir trail. Rather a small number of memory states can work on both trails. For future work, the relationship between problem space and memory amount should be investigated.

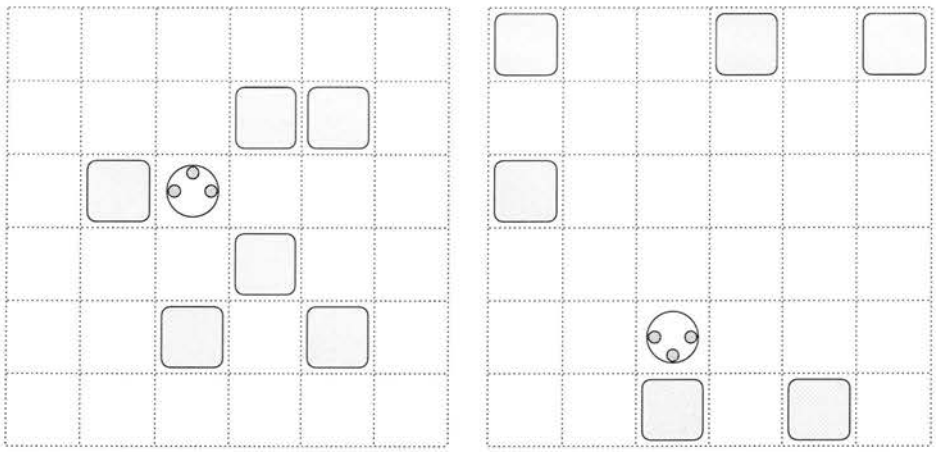
Chapter 5

Grid World Problems

In this chapter, the importance of memory is demonstrated with experiments on grid world problems. Two problems, the Tartarus problem [Teller, 1994; Ashlock, 1998] and the woods problem [Wilson, 1994; Lanzi, 2000] are used to investigate the ability of memory-based systems: They have been investigated by many researchers to explore memory-based systems and they are well known as difficult problems to solve with memoryless control systems [Teller, 1994; Wilson, 1994].

The Tartarus problem is a box pushing task. One agent has a maximum of eight sensors to detect the surrounding environment. This problem requires a new evolving structure to handle many sensors. Also the environment has too many configurations for evolutionary computation to handle, so a sampling process is applied to choose random environments. Finite state machines with three sensors are first applied to solve the problem and then tree state machines to cover eight sensors are used. For each control structure, the effect of memory is explained. The woods problem is a goal-search problem for an agent to try to find a goal position, starting at random initial positions. The agent has eight sensors and it needs to find the shortest path to the goal. We discuss the potential of evolving memory-based control structures to solve the problem and how it works to improve performance. Also it is investigated how the choice of sensors can influence the performance and memory requirement to achieve tasks.

In grid world problems, sensor readings and motor actions are simplified without any noise. The restrictions of agent sensors and motors distinguish them from robotic agents in reality, but they have been studied by many researchers to imitate real robotic tasks or study conceptual modeling for agent behaviors. The above two problems, the Tartarus problem and the woods problem, have been referred to as difficult problems requiring memory elements.



(a) (b)

Figure 5.1: Examples of the Tartarus environment (a) initial configuration (b) final configuration with score 8

5.1 Tartarus problem

The Tartarus problem was originally designed by Teller [1994]. It has a 6×6 board environment and six boxes randomly distributed in the inner 4×4 grid. A single agent has eight sensors for its eight nearest neighbors, each of which can detect a box, wall, or empty space. The agent has three motor actions: move forward, turn left or turn right. The agent can move forward only if there is an empty space in front of the agent itself or there is a box in front with an empty square beyond it. Attempting to push two boxes, or a box against a wall, results in no movements. The agent can have four possible directions and the direction can be changed by two operations, turn left or turn right. In the initial configuration six boxes are randomly placed in the inner grid and an agent is put in an empty random position in the inner grid, facing in a random direction.

The agent's goal is to push all six boxes to the walls. After a given exploration time passes, the agent gets a score by checking the number of boxes against walls. Score two is given for boxes in corners and score one for boxes in non-corner edge positions. The maximum score that the agent can achieve is thus 10. Eighty time steps are allowed for an agent and the time limit will lead the agent to find an efficient strategy for high score. Figure 5.1 shows examples of the Tartarus environment.

The Tartarus problem is different from the artificial ant problem in many respects. An agent has several sensors with integer values to distinguish two objects, wall, box, and empty space instead of binary values. Its environment is small in size, but random configurations are allowed.

More precisely, an agent has eight sensors to look around the environment and there are more than 300,000 Tartarus boards ($\approx_{16} C_6 \cdot_{10} C_1 \cdot 4 = \frac{16!}{6! \cdot 10!} \cdot \frac{10!}{1! \cdot 9!} \cdot 4$). The agent position and box positions are randomly selected in the initial configuration of the Tartarus environment, while the artificial ant problem has one fixed configuration for an agent. It has been pointed out that this problem requires intelligent use of states and so it is one of the benchmarking problems for memory states.

It has been shown that agents using memory in the Tartarus problem have an advantage over agents that do not use memory [Teller, 1994; Ashlock, 1998; Silva et al., 1999]. There have been many evolutionary approaches to handle the Tartarus problem. Teller used indexed memory in tree structures with genetic programming [Teller, 1994]. Also recurrent neural networks have been evolved to solve the problem [Balakrishnan and Honavar, 1996; Silva et al., 1999]. New structures with memory encoding, called GP-automata and ISAc (If-Skip-Action) lists, have been applied [Ashlock, 1998]. The research so far has focused on how a memory architecture can be organized to solve the problem. However, the complexity level of the Tartarus problem itself has not been considered seriously. Its complexity will be measured by the structure of evolved controllers with the best performance, and thus the complexity level of control systems will be investigated for this problem.

The following issues for the Tartarus problem will be answered in this section.

- How can we set up parameters in evolutionary algorithms?
- The Tartarus environment has many possible board configurations. How can the evolutionary computation manage all possible configurations?
- Is it possible for Boolean logic circuits with flip/flops to solve the Tartarus problem? How many memory elements are required to obtain satisfactory behavior?
- A Tartarus agent has eight sensors to detect its environment. Are all the sensors necessary? Or are only a subset of the entire sensors sufficient to produce its proper behavior? Is there any relationship between the number of sensors and the number of memory elements?
- FSM has a scaling problem in chromosome size. Is there any alternative to the memory architecture?

The first two issues handle evolutionary algorithm parameters such as population size, the number of generations, exploration time and the number of initial configurations or trials. The other three issues are related to memory.

5.1.1 Evolutionary Parameters

An FSM approach has been used to tackle the artificial ant problem, but for the Tartarus environment, an agent has eight sensors each of which needs three different values to detect wall, box or empty space. If an FSM is constructed with integer encoding for this problem, the genetic algorithm needs to have a chromosome size $3^8 \cdot 2n = 13,122n$ for a n-state machine. It is not practical to evolve controllers with this structure. Thus, it is suggested that only the three sensors at front, left and right among the 8 neighbor sensors should be used to detect the environment. The chromosome length can be reduced to a reasonable size $3^3 \cdot 2n = 54n$ for a n-state machine.

The Tartarus problem intrinsically has a large number of possible configurations and it is not efficient to evaluate a controller over all of these while evolving. The *N-K sampling method*[1] is applied to this kind of random environment [Kim and Hallam, 2001; Lee, 1998; Gathercole, 1998]. First, N initial configurations to represent the entire possible configuration of environments are randomly selected. Every generation K samples among N configurations are evaluated for each genome controller and as the evolutionary process continues, it intermittently but regularly evaluates all of N configurations for the best chromosome and records its fitness. This strategy can reduce greatly computation time and effectively accumulate the result of evolving controllers in random environments. Ultimately it will find the best controller to survive in the N environments. In the Tartarus problem, $N = 10,000$ is used; it is presumed that it is large enough to cover the whole Tartarus environment. The number of random boards, K, is fixed for one experiment. K boards are randomly selected among N configurations each generation and all the individuals in the same population are evaluated with the same K boards. N boards are evaluated every 50 generations for the best chromosome in the experiment and the best fitness results are saved.

We are interested in deciding what evolutionary parameters are most influential to find desirable solutions. Exploration time was fixed at 80 time steps for every experiment; 80 time steps was often used to explore the Tartarus environment [Ashlock, 1998]. Maximally eight states in the FSM were allowed. Each experiment was run 25 times and it employed a mutation rate of 0.0055 and a crossover rate of 0.6. Tournament selection with group size four was applied as used in the artificial ant problem.

The fitness test was applied to various parameter experiments, involving population size and

[1] Gathercole called it a random subset selection. It will be discussed in detail in chapter 7.

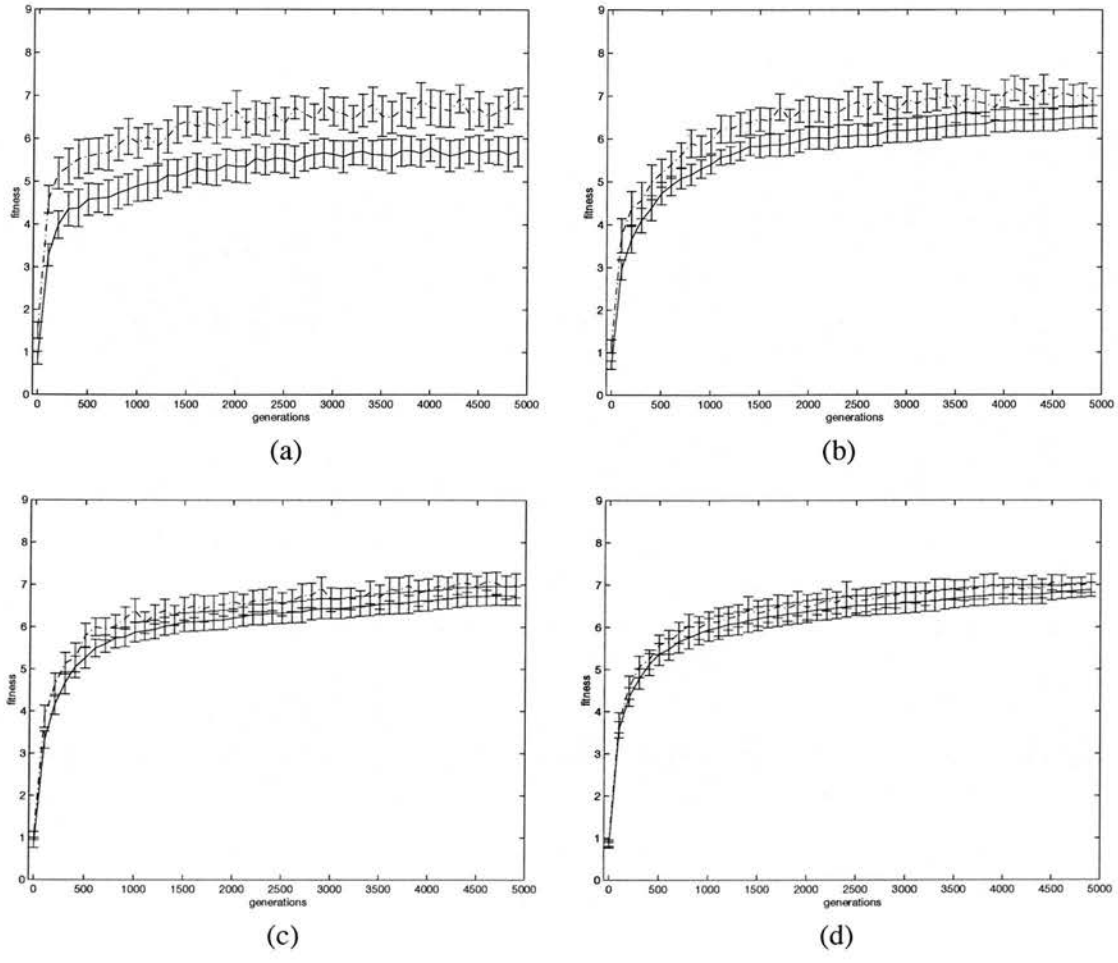


Figure 5.2: Experiments with various sample size in the Tartarus problem with 8 states and population size 50 (a) 5 samples (b) 10 samples (c) 20 samples (d) 50 samples (dashdot: average over K samples, solid: average over N boards)

the number of sample boards. Experiments with a fixed number of random sample boards were first tested with a different size population (see Figure C.1-C.3 in appendix C). In N-K sampling, $K = 10, 20, 50$ was used with 5000 generations[2]. For each fixed sample size K, the performance of a large-sized population was slightly better than that of a small-sized population. However, their performance was not significantly different. The same can be said of varying number of random samples with fixed population size. Sample size 50 had slightly better performance than sample size 10 in the experiments with population size 25, 50 and 100, but it was not significant. Figure 5.2 shows the difference of performance on sample size 5, 10, 20 and 50 with population size 50. Sample size 5 is significantly worse than any other sample

[2]Lee [1998] did some systematic investigation of the subset (sample) size using the average effort measure.

parameters			Generations					
population	time steps	boards	≤ 1000	≤ 2000	≤ 3000	≤ 4000	≤ 5000	effort cost
20	80	10	0	0	0	0	0	$[1.51C, --C]$
25	80	10	0	0	0	0	0	$[1.89C, --C]$
50	80	10	0	0	1	4	4	$[2.87C, 15.26C]$
100	80	10	0	0	1	3	8	$[1.93C, 5.81C]$
200	80	10	1	2	5	8	10	$[3.37C, 8.56C]$
20	80	20	0	0	0	3	6	$[0.92C, 3.46C]$
50	80	20	0	0	4	6	6	$[2.29C, 8.64C]$
100	80	20	1	2	4	7	9	$[3.59C, 9.89C]$
200	80	20	0	5	8	11	11	$[6.34C, 15.04C]$
20	80	50	0	1	6	8	9	$[1.80C, 4.94C]$
50	80	50	1	3	7	10	11	$[3.96C, 9.4C]$
100	80	50	2	6	8	11	14	$[6.81C, 13.54C]$
200	80	50	0	1	7	11	14	$[13.62C, 27.09C]$
20	80	100	0	5	6	8	9	$[3.59C, 9.89C]$
25	80	100	0	2	7	10	11	$[3.96C, 9.40C]$
50	80	100	2	5	9	13	15	$[6.52C, 12.32C]$
100	80	100	4	9	11	15	16	$[12.54C, 22.56C]$
200	80	100	3	10	15	17	18	$[23.34C, 38.30C]$

Table 5.1: Test with various configurations in the Tartarus problem (effort cost is evaluated for 5000 generations, C is a unit cost for a single run with population 100 and 10 random sample boards)

size and thus we can conclude that sample size 5 is too small for the Tartarus problem. When the sample size is smaller, there is a larger gap between the performance over K boards and the performance over N boards. The N-K sample method improves the performance over N boards by evolving controllers for K boards. When sample size K is small, it has a little slow convergence speed to increase the performance over N boards.

Various parameters for population size, the number of sample trials and the number of generations were tested for effort cost. Each parameter experiment was run 25 times and then the number of successes was counted; a success[3] is defined as the case when the fitness value averaged over 10,000 boards is greater than 7. The effort test, estimating the computing cost with the number of runs required to find the first success, was taken over the number of success cases. Table 5.1 shows the effort cost for various parameters. In terms of the effort cost, population size or the number of sample trials are not significant factors. In the experiments, exploration time is fixed. The effect of the remaining parameters, the number of samples, the number of generations and the size of population, was investigated. When population size or the number of samples is increased, the computing cost C given in equation 4.2 is increased. Population size 200 and 100 random boards can obtain 18 successes at 5000 generations, but

[3]This success is hard to achieve within a small number of generations and it takes many generations; the experiments will show the limit of memory-based control structures and so it should be run for a long time. Previous approaches show performance below score 6 with 250 generations.

population	boards	generations	# success	effort cost
200	10	5000	10	$[1.68C, 4.28C]$
200	20	2500	9	$[1.80C, 4.94C]$
200	50	1000	0	$[7.57C, --C]$
200	100	500	0	$[7.57C, --C]$
100	10	10000	13	$[1.42C, 3.00C]$
100	20	5000	9	$[1.80C, 4.94C]$
100	50	2000	6	$[2.29C, 8.64C]$
100	100	1000	4	$[2.87C, 15.26C]$
50	10	20000	9	$[1.80C, 4.94C]$
50	20	10000	15	$[1.30C, 2.46C]$
50	40	5000	12	$[1.50C, 3.34C]$
50	50	4000	10	$[1.68C, 4.28C]$
50	100	2000	5	$[2.54C, 11.14C]$
40	50	5000	12	$[1.50C, 3.34C]$
25	80	5000	9	$[1.80C, 4.94C]$
25	100	4000	10	$[1.68C, 4.28C]$
20	10	50000	17	$[1.21C, 2.07C]$
20	20	25000	17	$[1.21C, 2.07C]$
20	50	10000	17	$[1.21C, 2.07C]$
20	100	5000	9	$[1.80C, 4.94C]$

(a)

population	boards	generations	# success	effort cost
200	10	5000	10	$[1.68C, 4.28C]$
100	20	5000	9	$[1.80C, 4.94C]$
50	40	5000	12	$[1.50C, 3.34C]$
40	50	5000	12	$[1.50C, 3.34C]$
32	62	5000	15	$[1.30C, 2.46C]$
25	80	5000	9	$[1.80C, 4.94C]$
20	100	5000	9	$[1.80C, 4.94C]$

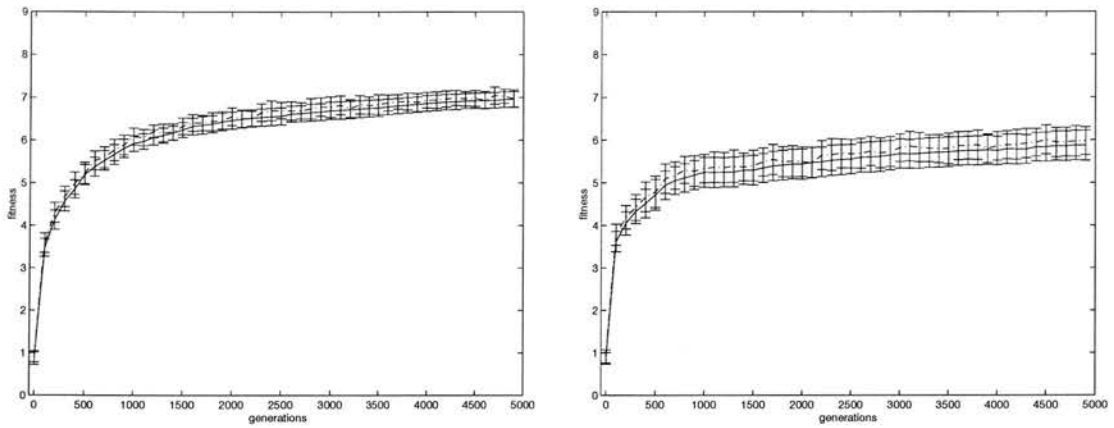
(b)

Table 5.2: Effort test on various configurations at the same computing cost in the Tartarus problem (a) variable generations (b) fixed generations (C is a unit cost for a single run)

the most intensive computing cost was invested to get the result. The effort cost reflects such a computing load. In terms of the effort cost, population size 20 and sample size 20 has the most efficient and economical effort to find the first success in Table 5.1.

Table 5.2 shows the effort test results at the same computing cost. If the population size is fixed at 200, 100 or 50, using a larger number of generations is more effective than using a larger number of random boards. Even if the number of random sample environments is fixed, increasing the number of generations is more effective rather than the population size. Thus, increasing the number of generations is recommended for good performance, instead of changing population size and the number of trials for random boards, if the same computation cost is assumed. When population size is small, it is desirable that the number of trials for each chromosome should be over or equal to 20. The experiments also compare the effects of population size and the number of random boards, when the number of generations is fixed at 5000. From Table 5.2(b), one can say that varying the number of random boards or varying the

size of the population is not significantly effective in terms of the effort cost when they run at the same computing cost.



(a) (b)

Figure 5.3: Comparison of finite state machine and GP-automata with 8 states, population size 20, 100 random boards. (a) finite state machine (b) GP-automata (dotdash: the average over K sample boards, solid: the average over $N(= 10,000)$ boards)

The above FSM approach is compared with the GP-automata suggested by Ashlock [1998]. In GP-automata, eight sensors are allowed to see the surrounding environment and sensor readings are handled in tree structures using genetic programming. Each state has its own tree and state transitions as shown in Figure 2.9. Figure 5.3 shows example results which demonstrate that finite state machines outperform GP-automata when the same computing cost is given (more figures are in Figures C.4-C.5 in appendix C). Generally the performance of GP-automata degrades with a small population. In contrast, the FSM approach is not influenced by the population size. This may be due to the fact that genetic programming tends to develop new good offspring through crossover of individuals in a large sized population rather than with the mutation operator [Nolfi and Floreano, 2000]. The FSM approach used only three sensors among the eight possible sensors. The above results indirectly show that the three sensors are sufficient to generate desirable behaviors.

5.1.2 Memory and Pareto Experiments

Various numbers of memory states were tested and the fitness distributions with 95% confidence intervals were examined (see Figure C.6 in appendix C). A 1-memory state machine corresponds to a memoryless reactive control system and its average score is poor, below 1. As the number of memory states increase, the average performance improves remarkably. Around

parameters				Generations					
# states	population	time steps	boards	≤ 1000	≤ 2000	≤ 3000	≤ 4000	≤ 5000	effort cost
2	100	80	50	0	0	0	0	0	$[7.57C, --C]$
4	100	80	50	0	0	0	0	1	$[5.09C, 105.71C]$
5	100	80	50	0	0	0	1	3	$[3.32C, 22.96C]$
6	100	80	50	1	3	3	4	6	$[2.29C, 8.64C]$
7	100	80	50	1	3	5	10	11	$[1.59C, 3.76C]$
8	100	80	50	2	6	8	11	14	$[1.36C, 2.71C]$
10	100	80	50	1	5	7	9	10	$[1.68C, 4.28C]$
12	100	80	50	1	1	5	10	14	$[1.36C, 2.71C]$

Table 5.3: Effort test with various memory states in the Tartarus problem (effort cost is evaluated for 5000 generations, and a success is defined as the case whose average score is above 7. C is a unit cost for a single run with population 100 and 50 random sample boards)

6 states, it reaches peak performance. For more than 6 states, there is no significant change. Thus, this fitness test says that the Tartarus problem requires at least 6 states to obtain satisfactory performance (above score 6.5) and purely reactive systems cannot solve the problem. Table 5.3 also proves it requires at least six states to get good performance. The table shows the number of successes among 25 trials with many generations. The performance of six states or more is significantly different from that of five states or less in effort cost with 5000 generations.

Multiobjective optimization over memory size and performance is applied to see a hierarchy of memory-based control structures. Figure 5.4 shows the result of evolving the best controllers with a given number of states. Pareto results also specify the limitation for each number of states. As shown in Figure 5.4(c), six states or more have a plateau curve in fitness performance. The FSM controllers with more than six states do not use their full memory state for the best performance; any additional memory states over six is redundant. It thus shows that six states are sufficient for the performance. The result is consistent with the experiments in Figure C.6.

It was shown that mutation rate can be chosen using the fitness test and effort test in the artificial ant problem. Mutation rate may be a factor even in the EMO approach. The chromosome in EMO has a variable-length coding. Various mutation rates were tested with the Pareto optimization method. In the Pareto experiments, mutation rate is not significantly effective as shown in Figure 5.5. However, mutation rate 0.0053 is slightly better in average fitness than the other mutation rates. It seems that the following mutation rate[4] is a rule-of-thumb with the

[4]This rate has been observed effective in our experiments of tournament selection, and it corresponds to the change of two genes per chromosome. We need more experiments to say if the mutation rate is better than any other rate.

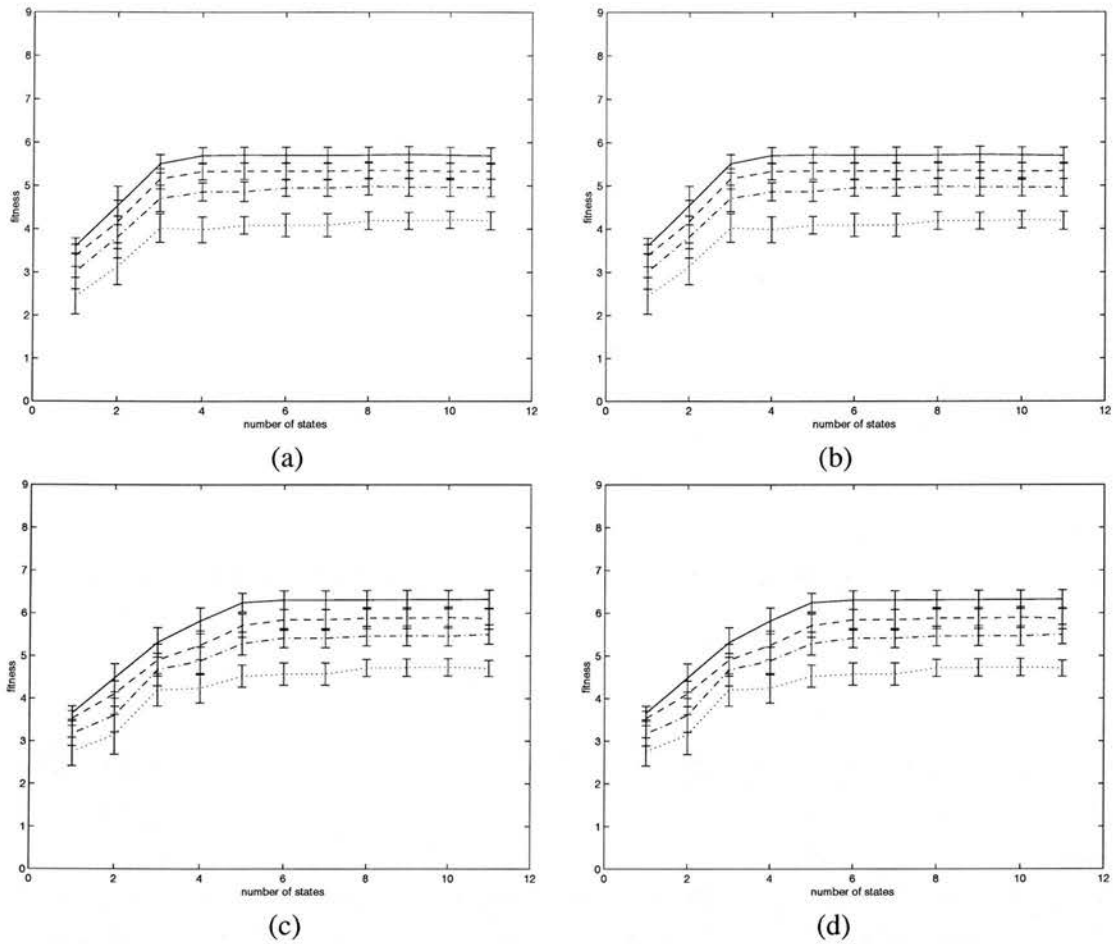


Figure 5.4: Pareto results in the Tartarus problem with population size 100 (a) 10 samples (b) 20 samples (c) 50 samples (d) 100 samples (dotted: 1,000 generations, dotdashed: 5,000 generations, dashed: 10,000 generations, solid: 20,000 generations)

experiments of tournament selection.

$$\text{rate} = \frac{2}{\text{chromosome size}} \tag{5.1}$$

When we assume that seven states are desirable for the Tartarus problem, mutation rate 0.0053 was calculated with the above formula.

Another experiment is to use a variable mutation rate in Pareto optimization. Evolutionary multiobjective optimization (EMO) uses variable length chromosomes for variable states. When chromosomes with various sizes are created for new offspring, variable mutation rates depending on the chromosome size are applied to each offspring. Figure 5.5(d) shows the EMO result. It is slightly better than the other mutation rates at three states and four states with 1000 gener-

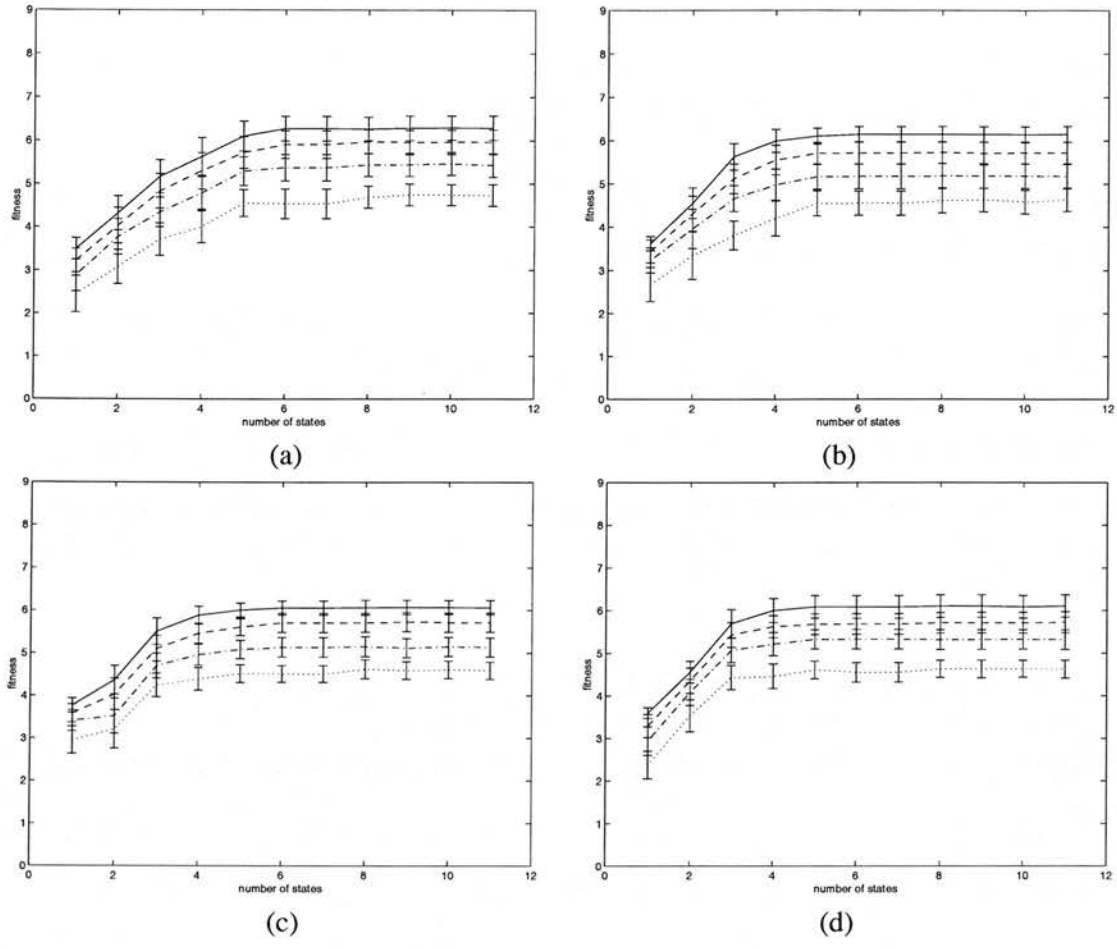
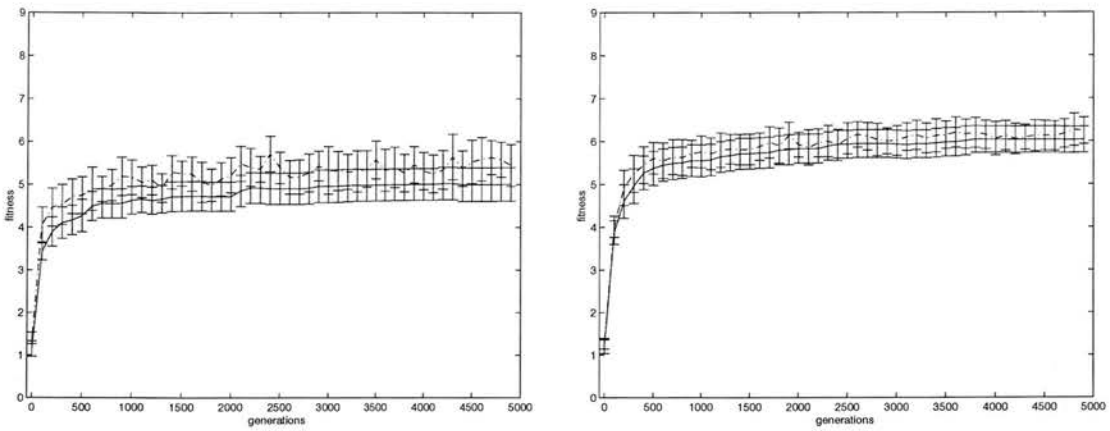


Figure 5.5: Pareto results in the Tartarus problem with various mutation rates; population size 100 and 50 random samples (a) mutation rate 0.0053 (b) rate 0.008 (c) rate 0.01 (d) variable rate (dotted: 1,000 generations, dotdashed: 5,000 generations, dashed: 10,000 generations, solid: 20,000 genera tions)

ations.

5.1.3 Tree State Machines

The FSM approach is powerful in obtaining state transition functions and count the number of memory elements, but it still has shortcomings in handling many sensors. In the Tartarus problem, an agent has eight sensors. It is very difficult to represent all sensor states in an FSM. A dynamic structure to represent features in sensor readings is a tree structure which can increase or decrease its size. In the dynamic tree structure, a leaf node corresponds to a sensor state and it has motor actions and the next state transition. All the internal nodes traversed from



(a) (b)

Figure 5.6: Tree state machine in the Tartarus problem; population 100 (a) 10 samples (b) 50 samples (dotdash: the average over K sample boards, solid: the average over N(= 10,000) boards)

the root node are linked together by logic AND operation over sensors and their thresholds. The structure is similar to a decision tree often used to classify data. Instead of defining class at leaf nodes shown in decision trees, motor actions and the next state transition are specified at leaf nodes.

The chromosome of a tree state machine is defined as a set of trees and each state has its own decision tree. Thus, a tree state machine has trees as many as the number of states. The crossover operator on tree state machines is defined as swapping a subset of trees of two parents and it is similar to the crossover in the finite state machines since some part of the whole chromosome is exchanged among two parents. The mutation operator is defined for one chromosome and it has two different schemes: One is to delete a subtree and create a new random subtree, and the other is to choose two subtrees among a set of trees for one chromosome and then swap two subtrees.

The tree state machine can expand sensor states up to its maximum limit, which can be defined by the user. Various sample sizes are tested with tree state machines. Tree state machines can easily expand the sensor space and so eight sensors are applied to the Tartarus problem. Maximally eight states or ten states were allowed and their performance did not show significant difference. Figure 5.6 shows the result obtained with population size 100 and maximum eight states. Its performance is worse than FSM approach and it has more variance.

Like GP-automata, the tree state machine reaches fitness 4 with a small number of generations, but it needs many generations to go over fitness 6.5. A genetic algorithm with FSM on the

Tartarus problem gradually increases the fitness and normally attains a fitness of over 6.5. It is believed that this is owing to the difference of search style between genetic algorithms and genetic programming. A genetic algorithm constantly improves its solution by the mutation operator, while genetic programming uses tree structures which are not significantly improved by the mutation operator.

5.2 Woods Problem

Wilson [1994] suggested that temporary memory can be added to a zeroth-level classifier system (ZCS) to solve problems in non-Markovian environments. It includes a few binary bits in a memory register to be set or reset by actions. The Woods problems has often been mentioned in association with the perceptual aliasing problem. The environment includes many ambiguous situations that require different actions. There have been many variations of the woods environment: Sutton [1990]'s gridworld, McCallum [1996]'s maze, maze10 [Lanzi, 1998], woods102 and woods14 [Cliff and Ross, 1995]. They range from simple to complex environments. They have been tackled with classifier systems or reinforcement algorithms [Lanzi, 1998, 2000]. The reinforcement systems are mainly based on purely reactive systems [Kaelbling, 1986]. The memory-encoding methods with reinforcement learning have been mostly successful on the woods problems [Lanzi, 2000; Cliff and Ross, 1995].

A woods environment is a grid world where agents have discrete sensors and a limited number of motor actions. An agent explores the environment to find its food but there are many trees as obstacles to block moving forward. The agent can recognize the status of each of its eight neighboring cells through eight sensors and so process only local information about its surrounding environment. The performance of a strategy or solution to a woods problem is measured by the average number of time steps needed to reach the goal from all possible initial cells. An optimal solution will be the minimum average steps to the goal.

The proposed method for memory analysis, evolutionary multiobjective optimization (EMO), was applied to woods problems. The penalty fitness function is defined as follows:

$$F_g = \frac{1}{|S|} \sum_{p \in S} \min(40, d(p, G)) \tag{5.2}$$

where S is the set of initial cells, G is the goal position and $d(u, v)$ is the path length from position u to position v. If the goal is not reachable from a cell, then the path length becomes

infinity and a high penalty value 40 is taken; 40 time steps are assigned for each initial cell and if agents can reach the goal from an initial cell, the number of time steps spent for exploration will be the path length from the initial cell to goal. The fitness F_g is the average time step taken from any empty cell to the goal. There may be at least one cell which cannot reach the goal when a strategy is applied. In this case we say the strategy cannot *solve* the problem.

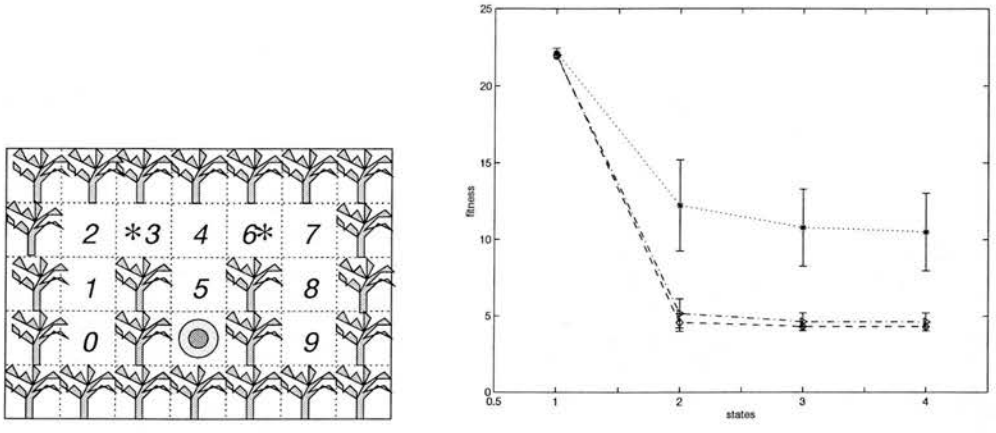
To prove that a given woods problem cannot be solved by specialized control systems, for example, purely reactive systems, we need to test all possible strategies which belong to the class of purely reactive systems. This is a very exhaustive search. In the experiments a control structure is given to solve the problem and evolutionary search will be applied with the control structure to try to find the best performance. If every trial is unsuccessful in finding a solution, then we assume that the control structure is not suitable for the problem. The proof of solvability of control structures for problems cannot be validated by theoretical foundation and it is based on experimental sampling data.

In the woods problems, FSM structure was used for memory analysis. Crossover rate was 0.6 and mutation rate followed the equation 5.1. Tournament selection with group size four was applied. For significance statistics, the EMO method was applied for memory analysis in the following woods problems instead of the fitness test and effort test.

5.2.1 Woods101 Problem

Figure 5.7(a) shows a simple maze that belongs to the class of the woods problems. It was tested to show hidden states by McCallum [1996] and it is called *McCallum's maze* or *woods101*. In the woods101 environment, an agent is placed at any empty cell and it can take an action towards one of four directions per one step; up, down, left and right. Normally woods problems allow eight directional moves by considering diagonal movements, but we give restrictions on motor actions. This makes the problem more difficult. The task is to find one food cell at the middle bottom in the shortest path. If a food cell is seen as the goal, this navigation problem becomes a goal search problem.

This problem cannot be solved by purely reactive agents. At some cells with '*' marks in Figure 5.7(a), agents have the same sensory patterns but need different actions. In the picture, an agent should move left at one cell and move right at the other. It has been shown that temporary memory can help solve such perceptual aliases [Cliff and Ross, 1995; McCallum, 1996]. A question can be raised, 'How many states are required to solve the perceptual aliasing problem in woods environments?'. An evolutionary multiobjective optimization (EMO) approach was



(a) (b)

Figure 5.7: Woods101 environment (a) grid world (b) memory analysis (*: 10 generations, ▷: 30 generations, ◇: 50 generations)

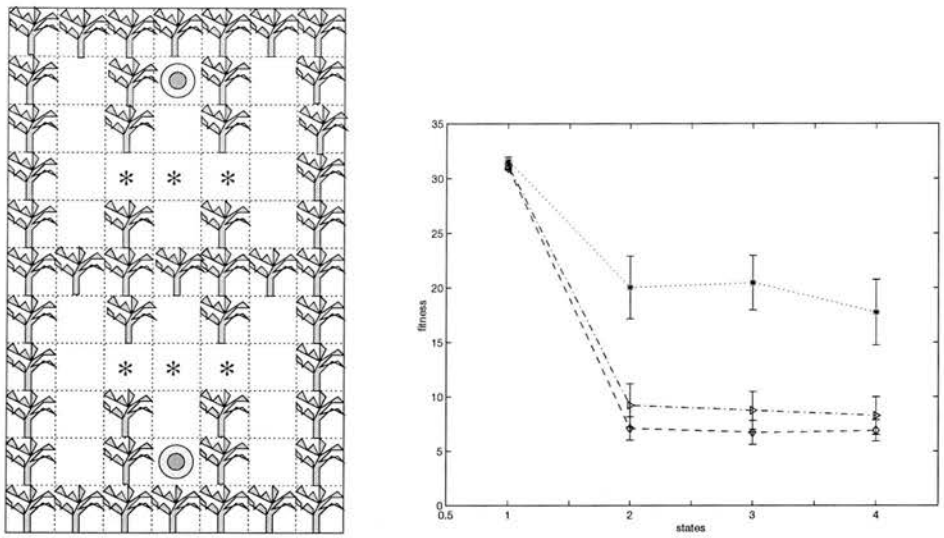
cell no.	trajectory							path length
0	D(1)	U(0)	U(0)	R(0)	R(0)	D(0)	D(1)	7
1	U(0)	R(0)	R(0)	D(0)	D(1)			5
2	R(0)	R(0)	D(0)	D(1)				4
3	R(0)	D(0)	D(1)					3
4	D(0)	D(1)						2
5	D(1)							1
6	R(0)	L(1)	L(0)	D(0)	D(1)			5
7	L(1)	L(0)	D(0)	D(1)				4
8	U(0)	L(1)	L(0)	D(0)	D(1)			5
9	D(1)	U(0)	U(0)	L(1)	L(0)	D(0)	D(1)	7

Table 5.4: An evolved solution of woods101 problem (each symbol in the trajectory represents motor action and state number in FSM, U: up, D: down, L: left, R: right)

applied to find the solution. The answer is two states. Even if four sensors[5] are used for an agent instead of eight sensors, the result still shows that two states are sufficient for this problem and more than two states are superfluous. When four sensors are used, it has six aliased positions. Sensor limitation increases the complexity of the problem by generating more sensor aliases. However, according to memory analysis, the problem requires two states as seen in Figure 5.7(b). Instead the agent with four sensors will need more time steps to find the goal. It wanders around the environment, looking for the situations which can trigger proper actions.

From the EMO approach, the best solutions are easily obtained within a small number of generations. An example solution is given in Table 5.4. It is noteworthy to see how it handles sensor aliases with internal memory. Cell 6 and cell 3 have the same sensory pattern. At cell 3, an agent moves right, down and down. At cell 6, it first moves right and sees a corner at

[5]Four sensors among eight are selected with front, back, left and right directions



(a) (b)

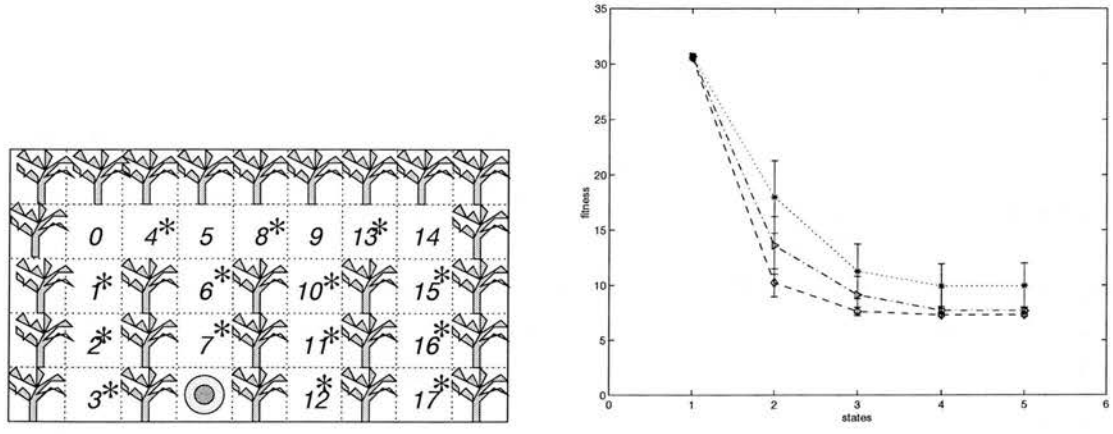
Figure 5.8: Woods102 environment (a) grid world (b) memory analysis (*: 10 generations, ▷: 30 generations, ◇: 50 generations)

right. Then it changes its state and turns left. The agent is now at cell 6. It experiences the same sensory pattern as at cell 3 but it has a different internal state and thus moves left, down and down. Cell 0, 5 and 9 also have the same sensations[6]. By utilizing internal memory states, the agent can reach the goal from any initial cell. Increasing the number of aliases does not necessarily require increasing the number of internal memory states. Agents can efficiently use their internal states by distributing sensor states over memory states. The EMO approach can find compact forms of internal states to achieve desirable solutions. In the woods101 problem, one bit of memory register is sufficient even for 4-sensor agents. Its performance has a total of 43 steps; this is the total length of the path from all initial positions to the goal. Its average score is 4.3 time steps, which is the fitness value shown in Figure 5.7(b).

5.2.2 Woods102 Problem

A new environment called woods102 was designed to have more difficult situations than woods 101 [Cliff and Ross, 1995]. This environment is more complex than woods101: there are several cells requiring different actions with sensor aliases marked '*' in Figure 5.8(a). Cliff and Ross [1995] mentioned that two bits of memory are sufficient to disambiguate the sensor aliases. We give restrictions on sensors such that four sensors are used. An agent will

[6]It is assumed that agents have binary sensors and so they cannot distinguish trees and food



(a) (b)

Figure 5.9: Maze10 environment (a) grid world (b) memory analysis (*: 10 generations, ▷: 30 generations, ◇: 50 generations)

experience more ambiguous situations with this sensor limitation.

When the EMO method with memory states ranging from one to four is applied, it shows that this problem still requires two states as in Figure 5.8(b). Thus, a one bit register can solve the problem. However, a memoryless policy – a one state machine – cannot handle perceptual aliasing and starting at many empty cells, the agent cannot reach goal. It has a high penalty in its fitness and Figure 5.8(b) shows that it is easily distinguishable from memory-based policies. Woods102 has more sensor aliases than woods101. In this problem, increasing the number of internal states improves performance. The EMO method was again applied to this problem with memory states from two to five, because one state performance is very different from the performance of machines with more than one state. This will give memory-based approaches a chance to search for the best performance (see Figure D.1 in appendix D). As a result, two states produce the best solution of 155 time steps in total or an average of 5.96 steps to the goal. The three state strategy has 146 time steps or an average of 5.62 steps and the four state strategy has 137 time steps or an average of 5.27 steps. More than four states have the same performance as four states. Thus, the best performance can be achieved with 2-bit registers.

5.2.3 Maze10 Problem

Maze10 is a problem tested by Lanzi [1998] and it has three aliasing positions when eight sensors are taken. He used an extension of XCS classifier system [Wilson, 1995] with internal memory to handle the problem. He emphasized the importance of exploration strategy in the

experiments to accelerate the convergence to an optimal solution. One bit of internal memory is sufficient to disambiguate the aliasing problems in the maze10 environment [Lanzi, 1998]. In the EMO experiment for the maze10 problem, four sensors are used for an agent instead of eight sensors. This generates many sensor aliases as shown in Figure 5.9(a). Cells 4, 8 and 13 share the same sensory input, as do cells 1, 2, 6, 10, 11, 15 and 16, as do cells 3, 7, 12, and 17.

Figure 5.9(b) shows the result of the EMO method for 50 generations. Its performance is not optimal, but we can infer from the diagram that more than two states are required to find near-optimal strategies. The memoryless policy fails to solve the maze10 problem and two states are sufficient to escape perceptual aliases. When more than 100 generations are run 25 times with the EMO method, the most of the best strategies are obtained; memory states range from two states to six states in this experiment (The result is displayed in Figure D.2 – see appendix D). As expected, memory with more than three states does not improve solutions of three internal states. Memory is not only a tool to escape perceptual aliasing, but also it plays a role of improving the performance by recording or marking environmental situations.

Table 5.5 shows the trajectory of the best performance. Two states give a performance of a total 141 time steps, an average of 7.83 time steps. Three states show the best performance with total 125 time steps, an average of 6.94 time steps (see Figure D.2 in appendix D). Starting at cell 17 with two internal states, an agent enters the hall of cell 10 at cell 9, visits cell 11 and turns back to cell 9. The agent cannot distinguish the situations of cell 5 and cell 9 and follows the given hall. As a result, it wastes time at cell 9 and it takes 15 time steps to the goal from cell 17. Adding one more state escapes this dilemma by memorizing environmental situations and taking a series of actions. Starting at cell 17 with three states, an agent skips entering cells 10-11 from cell 9 and goes straight to cell 8. It thus takes a total of 11 time steps to the goal from cell 17. The agent with three states records or marks the environmental situation at cell 14. At the time of passing cell 14, one internal state is marked and this marking makes the agent go straight without being distracted at cell 9. When cell 13 is a starting position, the agent does not have this sign for directions and so it visits cell 10 and 11. Furthermore cell 8 and cell 13 are not distinguishable in perceptions and going down the hall from cell 8 to cell 7 is the best policy. Visiting cell 11, starting from cell 13, cannot be avoided.

5.2.4 Sutton's Gridworld

In Sutton's gridworld [Sutton, 1990] as shown in Figure 5.10(a), an agent can sense eight neighbouring cells and takes one of four directional moves. The environment has 46 empty

cell no.	trajectory	path length
0	R(1), R(1), D(1), D(0), D(1)	5
1	U(0), R(1), R(1), D(1), D(0), D(1)	6
2	U(0), U(0), R(1), R(1), D(1), D(0), D(1)	7
3	D(1), U(0), U(0), U(0), R(1), R(1), D(1), D(0), D(1)	9
4	L(1), R(1), R(1), D(1), D(0), D(1)	6
5	L(0), L(1), R(1), R(1), D(1), D(0), D(1)	7
6	U(0), L(0), L(1), R(1), R(1), D(1), D(0), D(1)	8
7	D(1)	1
8	L(1), D(1), D(0), D(1)	4
9	L(0), L(1), D(1), D(0), D(1)	5
10	U(0), L(0), L(1), D(1), D(0), D(1)	6
11	U(0), U(0), L(0), L(1), D(1), D(0), D(1)	7
12	D(1), U(0), U(0), U(0), L(0), L(1), D(1), D(0), D(1)	9
13	L(1), D(1), D(0), U(0), U(0), L(0), L(1), D(1), D(0), D(1)	10
14	L(0), L(1), D(1), D(0), U(0), U(0), L(0), L(1), D(1), D(0), D(1)	11
15	U(0), L(0), L(1), D(1), D(0), U(0), U(0), L(0), L(1), D(1), D(0), D(1)	12
16	U(0), U(0), L(0), L(1), D(1), D(0), U(0), U(0), L(0), L(1), D(1), D(0), D(1)	13
17	D(1), U(0), U(0), U(0), L(0), L(1), D(1), D(0), U(0), U(0), L(0), L(1), D(1), D(0), D(1)	15

(a)

cell no.	trajectory											path length
0	R(1)	R(1)	D(1)	D(2)	D(1)							5
1	U(2)	R(1)	R(1)	D(1)	D(2)	D(1)						6
2	U(2)	U(2)	R(1)	R(1)	D(1)	D(2)	D(1)					7
3	D(1)	U(0)	U(2)	U(2)	R(1)	R(1)	D(1)	D(2)	D(1)			9
4	L(1)	R(1)	R(1)	D(1)	D(2)	D(1)						6
5	L(0)	L(1)	R(1)	R(1)	D(1)	D(2)	D(1)					7
6	U(2)	L(0)	L(1)	R(1)	R(1)	D(1)	D(2)	D(1)				8
7	D(1)											1
8	L(1)	D(1)	D(2)	D(1)								4
9	L(0)	L(1)	D(1)	D(2)	D(1)							5
10	U(2)	L(0)	L(1)	D(1)	D(2)	D(1)						6
11	U(2)	U(2)	L(0)	L(1)	D(1)	D(2)	D(1)					7
12	D(1)	U(0)	U(2)	U(2)	L(0)	L(1)	D(1)	D(2)	D(1)			9
13	L(1)	D(1)	D(2)	U(2)	U(2)	L(0)	L(1)	D(1)	D(2)	D(1)		10
14	L(2)	L(0)	L(0)	L(1)	D(1)	D(2)	D(1)					7
15	U(2)	L(2)	L(0)	L(0)	L(1)	D(1)	D(2)	D(1)				8
16	U(2)	U(2)	L(2)	L(0)	L(0)	L(1)	D(1)	D(2)	D(1)			9
17	D(1)	U(0)	U(2)	U(2)	L(2)	L(0)	L(0)	L(1)	D(1)	D(2)	D(1)	11

(b)

Table 5.5: Evolved solutions of maze10 problem (a) 2 internal states (b) 3 internal states (each symbol in the trajectory represents motor action and state number in FSM, U: up, D: down, L: left, R: right)

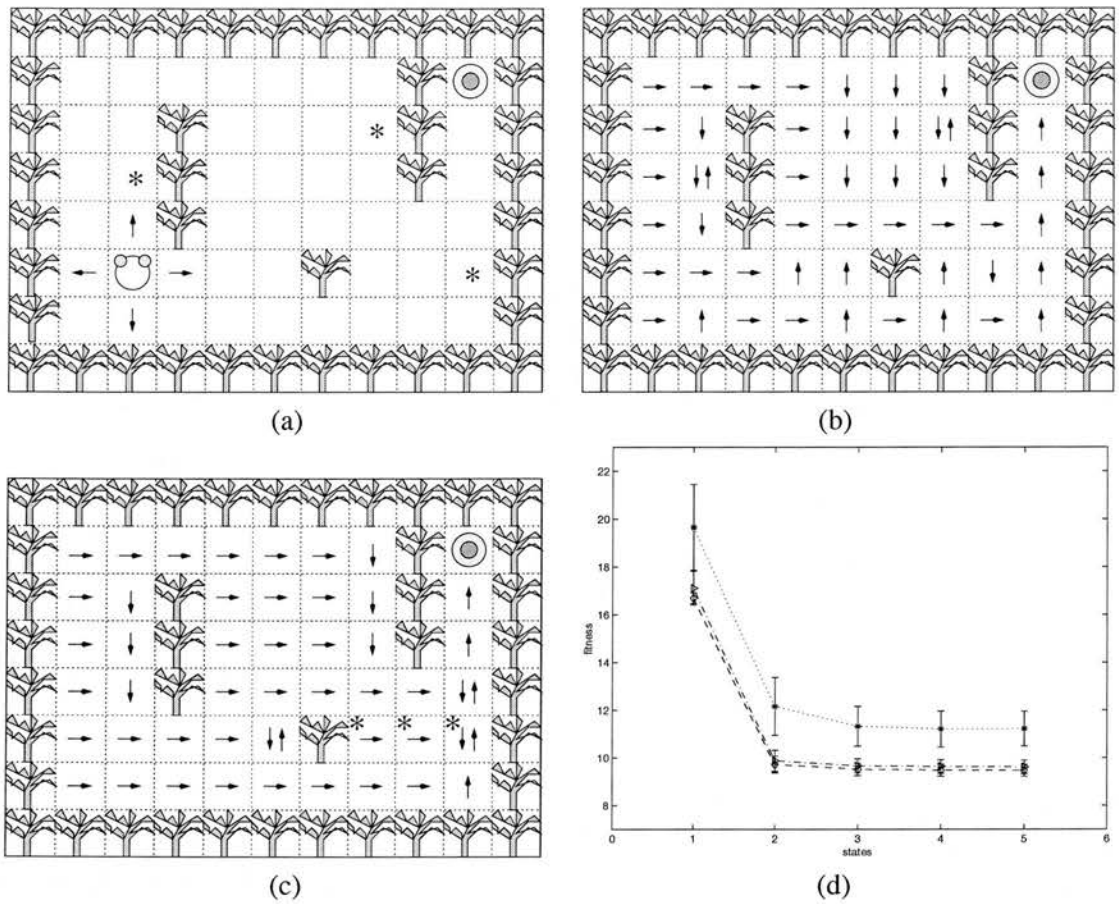


Figure 5.10: Sutton's grid world (a) grid world (b) the best solution with 8 sensors (c) the best solution with 4 sensors (d) Pareto result (*: 50 generations, ▷: 200 generations, ◇: 500 generations)

cells, 30 distinct sensor states and one goal position. Littman [1994] used a branch-and-bound algorithm to find the best memoryless policy. The best memoryless strategy takes a total of 416 steps, an average of 9.04 steps (= 416/46) to reach the goal. Especially at cells with three trees on the right side (there are three cells marked with '*') the best memoryless agent experiences ambiguous perceptions and takes inefficient movements, even though it can reach the goal from any initial cell. The agent chooses a roundabout way due to perceptual aliasing situations.

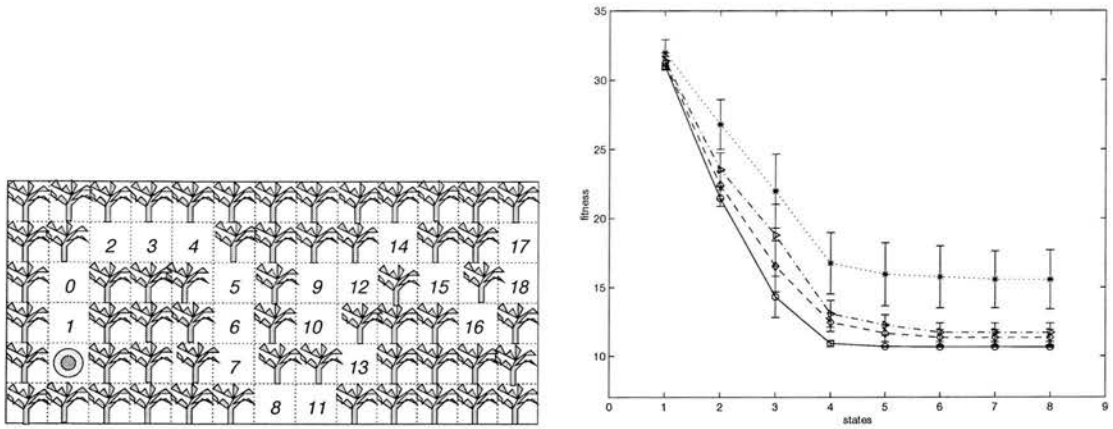
When internal memory is used, two internal states yield an optimal strategy with performance of 410 total steps to goal, an average of 8.91 steps. Its trace is displayed in Figure 5.10(b). Two directional signs at one cell means that internal states are involved to give agents appropriate actions. Internal states can be seen as milestones to say which direction is appropriate and they are determined by the past history that the agent has experienced.

When four sensors among all eight sensors are taken as in woods101 experiments, its performance is worse than eight sensor experiments. Agents have more sensor aliases for a tree on the right. The memoryless strategy cannot solve this problem unlike eight sensor experiments. There are several cells from which the goal cannot be reached. Its performance deviates from the fitness of memory-based strategy as shown in Figure 5.10(d). The best policy with two states has a performance of total 418 time steps, an average of 9.08 steps to goal. The best strategy as shown in Figure 5.10(c) is different from the eight sensor strategy in Figure 5.10(b). An agent chooses to go down when it first sees a tree on the right side. If there are trees on the right and below, it is bounced back to go up with an internal state marker. Otherwise, the agent moves right. The EMO method shows that more than two states have a similar performance as displayed in Figure 5.10(d). One can say that two states are sufficient for this problem with four sensors. However, 418 time steps is not an optimal solution with four sensors. An optimal strategy with four states was found with a total of 416 time steps. In this strategy, two additional states record the perceptual situation (it is marked '*' in Figure 5.10(c)) where a tree is detected on the left side and after moving right in two steps, another tree is seen on the right side. It thus blocks going down when the agent sees a tree on the right side and the agent can go straight up, while two or three state machines waste time to go down and return to the cell.

If an agent can recognize every feature about the entire environment, it will obtain an optimal performance of 404 steps from all empty cells, average of 8.78 steps to goal. However, the agent has only local information about its neighboring cells. Internal memory states help agents overcome the shortcomings of the local information about the environment. The performance of memory-based policy is quite close to the optimal performance of 404 steps. The number of sensors is also an important factor to obtain efficient solutions. In this problem, an optimal memoryless strategy with eight sensors has the same performance as an optimal four-state strategy with four sensors. Reducing sensors or restricting sensor range often requires increasing internal states. In other words, more sensors for agents will have more chances to develop purely reactive controllers.

5.2.5 Woods14 Problem

Woods14 is a Markovian environment designed by Cliff and Ross [1995] with eight sensors available on the agents. It has a simple linear path of empty cells and agents need to go through a field of trees. At each cell an agent experiences a different sensation and only one appropriate action among eight directional moves should be taken to reach the goal. Cliff and Ross [1995]



(a) (b)

Figure 5.11: Woods14 environment (a) grid world (b) memory analysis (*: 100 generations, ▷: 500 generations, ◇: 1000 generations, ○: 5000 generations)

showed that ZCS with internal memory cannot succeed in developing desirable control systems in the woods14 problem. Woods14, shown in Figure 5.11(a), is a very difficult problem for ZCS to solve because it requires long chains of actions before the reward is given. Also ZCS tends to create conflicting, overgeneral classifiers producing inappropriate actions. A conflicting overgeneral classifier is one that matches multiple sensations and does not cover appropriate actions for each sensation. This happens when the rule for its sense vectors are overgeneralized with the don't care term '#'.

An FSM with eight sensors, allowing 256 distinct sensor states, was evolved to solve the woods14 problem with a perfect score, average 10 time steps. Without difficulty, evolutionary computation succeeded in finding an appropriate strategy. When only four sensors are used as in the woods101 problem, this woods14 problem generates complex situations. Most empty cells have sensor aliases because of the sensor limitation. The problem becomes a non-Markovian environment.

EMO analysis for four-sensor agents shows that the memoryless policy fails to find solutions to the woods14 problem. Even two or three states could not succeed in solving the problem; there are starting cells from which the agent cannot reach the goal. As shown in Figure 5.11(b), there is a hierarchy of complexity with memory size. For two or three internal states, their performance is located between the memoryless strategy and the four state strategy. It depends on how many empty cells were successful to find the path to the goal. An optimal memoryless policy has 5 empty cells that can be connected to the goal. The other 14 cells failed to reach the goal. With two states, the goal is reachable from only 11 empty cells. Three states gives 17

cell no.	trajectory	length
0	0(2), 0(3)	2
1	4(2), 0(3)	2
2	7(1), 0(2), 0(3)	3
3	3(0), 7(1), 0(2), 0(3)	4
4	3(1), 3(0), 7(1), 0(2), 0(3)	5
5	0(2), 2(2), 6(0), 3(1), 3(0), 7(1), 0(2), 0(3)	8
6	2(2), 6(0), 3(1), 3(0), 7(1), 0(2), 0(3)	7
7	4(2), 6(1), 2(2), 2(2), 6(0), 3(1), 3(0), 7(1), 0(2), 0(3)	10
8	7(1), 6(1), 2(2), 2(2), 6(0), 3(1), 3(0), 7(1), 0(2), 0(3)	10
9	0(0), 4(2), 7(0), 3(1), 6(1), 2(2), 2(2), 6(0), 3(1), 3(0), 7(1), 0(2), 0(3)	13
10	4(2), 7(0), 3(1), 6(1), 2(2), 2(2), 6(0), 3(1), 3(0), 7(1), 0(2), 0(3)	12
11	3(1), 6(1), 2(2), 2(2), 6(0), 3(1), 3(0), 7(1), 0(2), 0(3)	10
12	3(1), 0(0), 4(2), 7(0), 3(1), 6(1), 2(2), 2(2), 6(0), 3(1), 3(0), 7(1), 0(2), 0(3)	14
13	7(1), 7(0), 3(1), 6(1), 2(2), 2(2), 6(0), 3(1), 3(0), 7(1), 0(2), 0(3)	12
14	7(1), 7(0), 4(2), 7(0), 3(1), 6(1), 2(2), 2(2), 6(0), 3(1), 3(0), 7(1), 0(2), 0(3)	14
15	7(1), 6(0), 7(1), 7(0), 4(2), 7(0), 3(1), 6(1), 2(2), 2(2), 6(0), 3(1), 3(0), 7(1), 0(2), 0(3)	16
16	7(1), 6(0), 7(1), 6(0), 7(1), 7(0), 4(2), 7(0), 3(1), 6(1), 2(2), 2(2), 6(0), 3(1), 3(0), 7(1), 0(2), 0(3)	18
17	0(2), 0(3), 7(3), 6(1), 6(0), 7(1), 7(0), 4(2), 7(0), 3(1), 6(1), 2(2), 2(2), 6(0), 3(1), 3(0), 7(1), 0(2), 0(3)	19
18	4(2), 0(3), 7(3), 6(1), 6(0), 7(1), 7(0), 4(2), 7(0), 3(1), 6(1), 2(2), 2(2), 6(0), 3(1), 3(0), 7(1), 0(2), 0(3)	19

Table 5.6: An evolved solution of woods14 problem (each pair of numbers in the trajectory represents motor action and state number in FSM, 0: down, 1: right, 2: up, 3: left, 4: lower right, 5: upper right, 6: upper left, 7: lower left)

successful cells among the total of 19 cells. The diagram in Figure 5.11(b) shows how closely a given number of states can achieve solutions.

One of the best solutions with four state machines, as in Table 5.6, shows how to handle perceptual aliasing. Cell 13, 14, 15 and 16 have the same sensory pattern. They are surrounded by trees in all four directions. Cells 13 and 14 need the same action of moving to the lower left, while both cell 15 and cell 16 need an action of moving to the upper left. The strategy first tries to move lower-left and if the situation is unchanged, then it tries to move upper-left. The two actions are sequentialized with internal states, depending on sensations. Cell 0, 5 and 17 have sensor aliases. Cells 0 and 17 prefer to take an action of moving down, but cell 5 needs an action of moving to the upper-left. These kinds of sensor aliases require internal states to escape cul-de-sacs. Internal states place marks on each sensor alias state and then block taking the same action. For instance, an agent first moves down at cell 5, changes its internal state, returns to cell 5 and then moves to the upper-left. The agent does not memorize sensation values and puts milestones at particular sensations. In this way internal memory can be easily represented by Boolean logical values or finite states.

If the number of sensors is decreased, more sensor aliases may occur, but not necessarily. Instead of four or eight sensors, two sensors are selected among eight sensors to see the influence of sensor limitation. First, FSMs with two sensors, front and right sensors, are evolved and the EMO analysis in Figure 5.12(a) shows that three internal states are not sufficient to solve the

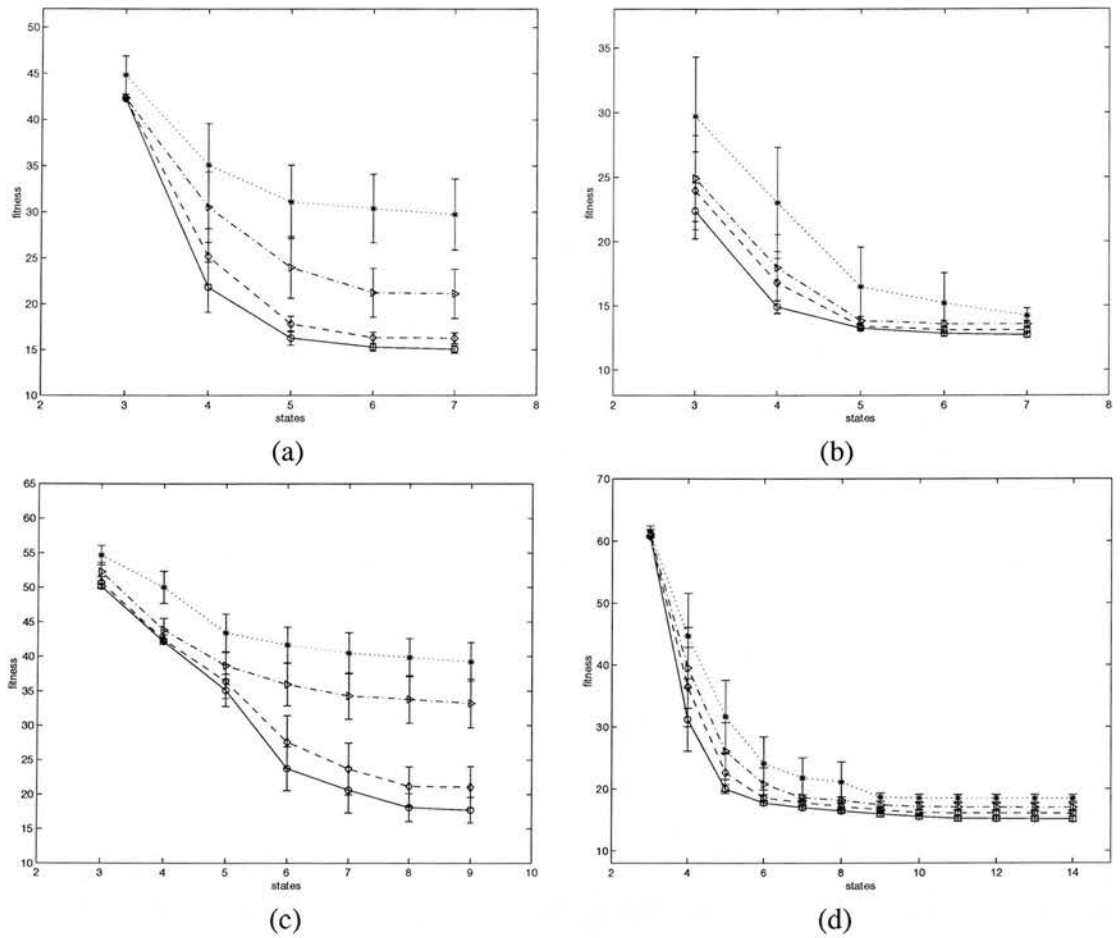


Figure 5.12: Memory analysis over the woods14 environment with one or two sensors (a) two sensors, front and right (b) two sensors, front and left (c) two sensors, left and right (d) one sensor, front (*: 500 generations, ▷: 1000 generations, ◇: 2000 generations, ○: 5000 generations)

problem[7] ; there are 9 initial cells which cannot reach goal for the best three-state machine. In the experiments with one or two sensors, penalty 80 is used instead of 40 in equation 5.2 to see a clear distinction of solvability for each state machine. In this case four state machines can solve the problem with a total 301 time steps to reach the goal from all empty cells while four-state agents with four sensors have a score of total 198 time steps[8]. When two sensors in front and left are selected, it still required four states to solve the problem and the performance is improved with a total of 245 time steps to the goal as shown in Figure 5.12(a)-(b). For three state machines, there are two initial cells which cannot reach the goal. However, the selection of two

[7]We assume that the problem cannot be solved with a specified control structure when no experimental run among 25 runs is successful to solve the problem

[8]These score solutions may not be optimal, and they are obtained from 5,000 generations

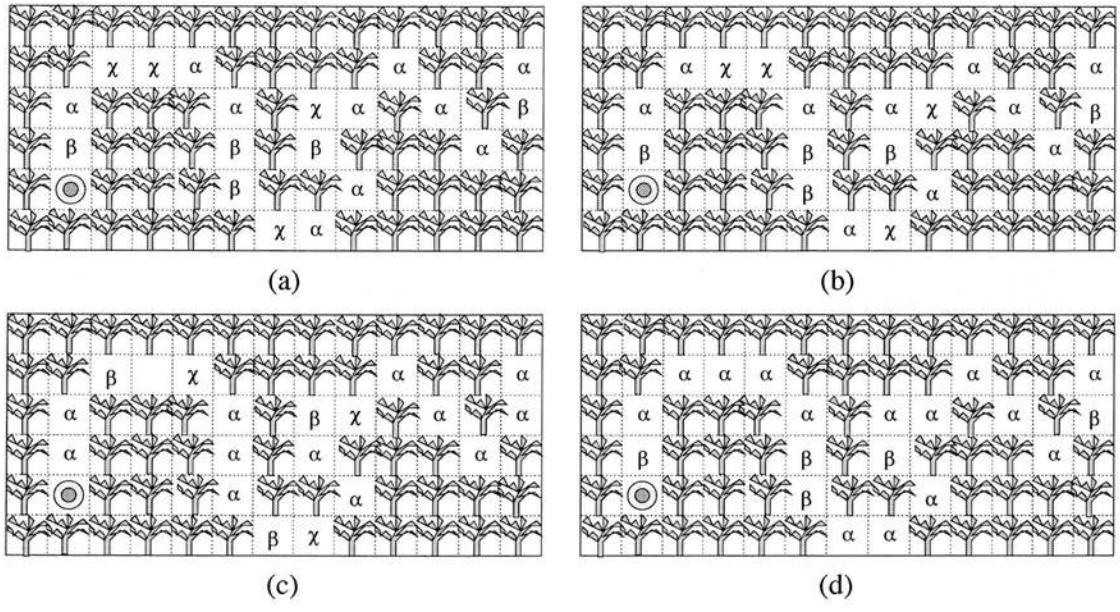


Figure 5.13: Perceptual aliasing in the Woods14 problem (a) two sensors, front and right (b) two sensors, front and left (c) two sensors, left and right (d) one sensor, front (each Greek symbol represents sensor aliases)

sensors on the left and right makes the wood14 problem more difficult. Even with five internal states, there are still three cells from which the agent cannot reach the goal. Six or more states could solve the problem. Figure 5.12(c) shows a significant change of performance between five states and six states. Even if there is a limitation on the number of sensors, what sensors are selected will be an important issue to solve problems and produce good performance.

When only one sensor, the front sensor, is used, four states are sufficient to solve the problem with a total of 371 time steps to the goal. Thus it unexpectedly outperformed the result of two sensors on the left and right. Three state machines had 14 cells which failed to reach the goal. Restricting sensors generally degrades the performance, but it does not say definitely that more internal states are required, considering results of two sensors on the left and right as well as only one sensor in front. It is notable that two sensor results including the front sensor are better than the result with only one front sensor, even though all of them require four states to solve the wood14 problem. In Figure 5.12(d), the best performance of the one sensor experiment even with many states is significantly worse than the performance of two sensor experiments including the front sensor.

Figure 5.13 shows perceptual aliases for a different set of sensors. The required amount of memory is not related to the number of total aliases. One can say that a sequence of perceptual aliases, that is, how ordered they are, is a more important factor in estimating memory amount

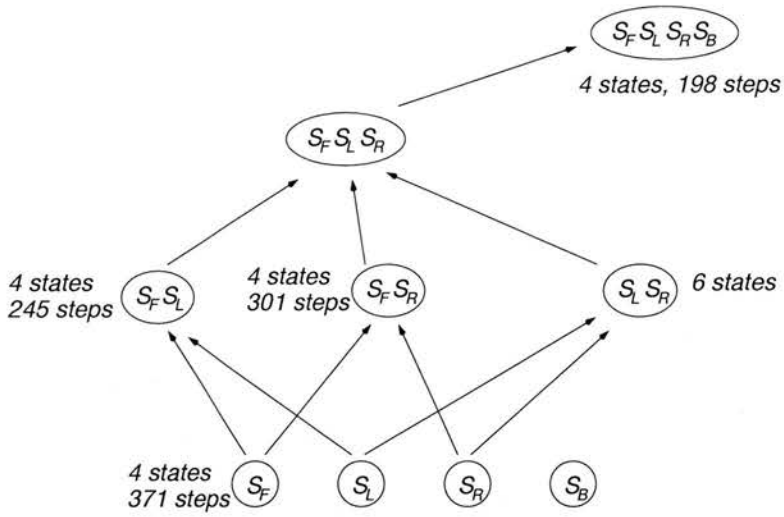


Figure 5.14: Partial order relation on performance with sensors in the woods14 problem (S_F, S_L, S_R, S_B are sensors in front, left, right and back)

needed to achieve a task.

We can build a partial ordering relation on the performance for a various number of sensors. For $S_A \subseteq S_B$, where S_A, S_B are a set of sensors, the performance G keeps the partial order for the same amount of memory, M.

$$G(S_A, M) \leq G(S_B, M)$$

More sensors can see the environment better. Also for $M_1 \leq M_2$, the partial order relation holds with a given set of sensors.

$$G(S_A, M_1) \leq G(S_A, M_2)$$

Using the above property, the following partial order relation will be inferred for the performance of robotic tasks or agent problems.

$$G(S_A, M_1) \leq G(S_B, M_2)$$

where $S_A \subseteq S_B$, and $M_1 \leq M_2$.

When a pair of sensors and memory is not ordered with each other, for example, there is no subset relation between S_A and S_B, or $S_A \subseteq S_B$ and $M_1 > M_2$, the performance order cannot be predictable. It will depend on tasks and environmental situations. The partial order relation in the woods14 environment is displayed in Figure 5.14. If the front sensor is included, at

most four memory states are required to solve the problem. When the number of sensors is increased, the performance becomes better. Thus, it is presumed that agents with front, left and right sensors will solve the problem with four states and have performance in the range ($198 \leq G \leq 245$).

5.3 Summary

In this chapter, two kinds of problems and their memory analysis are presented.

The Tartarus problem is more complex than the Santa Fe trail. It has many random environments and also eight sensors are used for an agent. When only three sensors among eight sensors are selected with the FSM approach, it shows similar results to other methods with eight sensors. Also the fitness and effort tests determine how many random samples are effective. Evolutionary multiobjective optimization (EMO) is applied to find the minimal structure with the best fitness. From this memory analysis, the Tartarus problem requires six or seven states to process sensor information appropriately.

Woods problems are goal search problems with perceptual aliasing. Five woods problems, woods101, woods102, maze10, Sutton's gridworld and woods14, were solved with EMO analysis. The woods101 environment requires 2 states to solve the problem, and woods102 also needs two states to reach the goal from any initial cell but four states for an optimal solution. The maze10 problem was solved with three states for four-sensor agents and Sutton's gridworld can be solved with two states. Reducing the number of sensors to four sensors requires four states for an optimal solution. Woods14 requires four states for an optimal solution with four sensors. As observed in the maze10 problem, the number of sensors influences the necessary memory size. Reducing the number of sensors or sensor range has the potential of creating more sensor aliases. The memory required to solve a problem is not necessarily proportional to the number of aliases. Instead it is more concerned with what kind of sensor aliases are experienced in the path. In the case where the same patterns are serially placed in the path, it may increase the memory size requirement. Memory plays an important role to solve sensor aliases and improves the performance. The EMO approach was effective to find the minimum number of memory elements and see a hierarchy of performance depending on memory size.

All the problems described above show that purely reactive systems are inferior to memory-based systems. However, they prove that a large number of memory elements are not necessarily required. A small number of bits are sufficient to evolve desirable behaviors. Many

complex control designs have been suggested for agent problems. The problem set provided in this chapter may be simple, but the results indirectly imply that simple reactive systems with small-sized memory are effective in agent problems.

Chapter 6

Simple Robot Model and Tasks

In this chapter, several robotic tasks are explored to see the effect of memory in reactive systems. In many respects, robotic tasks are different from the grid world problems described in the previous chapter. Robots can have a variety of sensor ranges and motor actions and they interact dynamically with environments. Reactive systems in mobile robots are often seen as a direct coupling between perception and action without any intermediate processing. However, some behavior-based approaches have used a small amount of memory in their reactive robot controllers. Thus, a variety of robotic tasks will be investigated with internal states through evolutionary computation techniques. They will explore the effect of adding known amounts of memory to purely reactive systems in a variety of tasks. The memory elements will be represented in the form of a set of rules with states, equivalent to a Boolean logic network with a fixed number of 1-bit registers.

It is assumed that robots have simple binary sensors to detect environments. Sensors consist of whisker-like sensors and light vector sensors to say if there is any object or light ahead. A simulated robot for five tasks–obstacle avoidance, wall following, exploration, box pushing, and corridor following–with two sensor or four sensor configurations is applied to compare the performance of stateless and memory-based controllers. Its controller is a rule-based state machine, which will be evolved with genetic algorithm. The rule-based state machine will not only make it easy to quantify the size of memory but also overcome the shortcomings of finite state machines.

Sensor models and control structures for robots are first described and then each robotic task and its fitness function is explained. Each robotic task and behavior performance are discussed with the effect of memory on the resulting evolved robot behaviors. The bulk of this chapter is

based on the paper [Kim and Hallam, 2001].

6.1 Methods

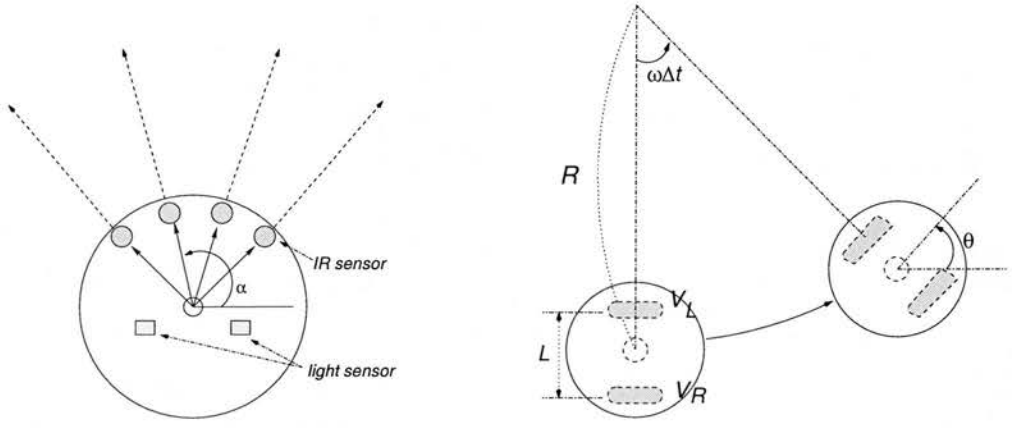
Some evolutionary robotics experiments [Lee, 1998] have shown that purely reactive systems without memory can implement various primitive behaviors: for example, object avoidance and exploration behaviors, and box-pushing to a target signalled by an environmental marker need only a purely reactive controller with binary sensory inputs, and more complex combinations of such behaviors can be realized using stateless switchers to arbitrate between the primitives.

Rather than using neural networks, Lee et al. [1997] evolve a Boolean logic network using genetic programming (GP). In this approach, each behavior is a purely reactive controller with no internal state [Lee, 1998; Lee et al., 1997]. Its control structure is a combinational logic circuit, evolved using GP, acting on sensor values converted to binary by thresholding or comparison (the thresholds and comparisons are also determined by the GP evolution). Such an approach offers the possibility of adding memory in an easily quantified way.

In this chapter, we will investigate the effects of memory on Boolean logic circuits by evolving state machines. A simulation program is used to explore robotic tasks but without noise. Even though it is different from reality, we can simulate some of the important properties of robot control systems and easily test how robotic tasks are related to internal memory. The effect of adding known amounts of memory to purely reactive systems in a variety of robotic tasks will be explored. Robotic tasks in noisy environments will be studied in the following chapters. Each section will describe a set of experiments that reveal behavioral and performance differences between a purely reactive controller and a memory-based reactive controller evolved for various particular tasks, and will explain the effect of memory on the resulting robot behaviors.

The experiments on a variety of tasks use simple robot models where robot kinematics are simulated but sensor readings are binarized. The robot model is a Khepera-style robot [Mondada et al., 1993], with binary sensors in front to detect any wall or obstacle. The sensor detects the intersection of an object with a fixed length ray projected along the sensor's pointing direction and computes the distance to the nearest such intersection along the ray[1]. The sensor values are binary, derived by thresholding the distance provided by the sensor, and sensor readings just say if any object is near the robot within some limit in the direction of the sensor. These binary

[1]Those sensors are ray-based distance sensors without any explicit acceptance cone, rather than modelled on infrared sensors.



(a) (b)

Figure 6.1: Robot model and kinematics in simulation (a) robot model (b) robot kinematics geometry

sensors do not give any quantitative value of how close to any obstacle the robot is. They are whisker-like sensors. In the experiments, robots have two sensor configurations: two infrared sensors with angle $45°, 135°$, or four infrared sensors with angle $45°, 80°, 100°, 135°$ as shown in Figure 6.1(a). Two light sensors are used in a box pushing task.

The Khepera-like robots consist of two drive wheels on a common axis controlled by two different motors. The motor actions for the left and right wheel vary the rotational velocity of the two wheels, that is, the robot's motion and trajectories. Kinematics on the left and right wheels will lead to the following path at the same angular speed ω.

$$(R - \frac{l}{2})\omega = v_l, \qquad (R + \frac{l}{2})\omega = v_r$$

where l is the wheel base length, and v_r, v_l are the right and left wheel speeds, respectively. Thus,

$$R = \frac{l}{2}\frac{(v_l + v_r)}{v_r - v_l}, \qquad \omega = \frac{(v_r - v_l)}{l}$$

When $v_l = v_r$, the radius R becomes infinite and the robot moves straightforward. If two wheel speeds are different, the robot will show a curved motion with radius R as shown in Figure 6.1(b). As a result, the robot will change its direction smoothly. From the robot kinematics, the following trajectory equation can be derived [Dudek and Jenkin, 2000].

$$\begin{pmatrix} x' \\ y' \\ \theta' \end{pmatrix} = \begin{pmatrix} \cos(\omega\Delta t) & -\sin(\omega\Delta t) & 0 \\ \sin(\omega\Delta t) & \cos(\omega\Delta t) & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} R\sin(\theta)) \\ -R\cos(\theta)) \\ \theta \end{pmatrix} + \begin{pmatrix} x - R\sin(\theta) \\ y + R\cos(\theta) \\ \omega\Delta t \end{pmatrix}$$

where θ is the robot heading direction, (x, y) is the current robot position and (x', y') is the next robot position. In simulation, 3 bits, eight possible values, {-16.0, -11.4, -6.9, -2.3, 2.3, 6.9, 11.4, 16.0} were allowed for each wheel action; each motor action controls the forward and backward speed such that robots can navigate even in narrow environments.

A finite state machine proved to be useful to quantify internal states in the grid world problem. However, it has shortcomings of encoding chromosomes for many sensors and several states, since the gene encoding size is exponentially increasing in terms of the number of sensors. An alternative representation is a rule-based state machine described in chapter 3 (section 3.1.3). It consists of a set of rules where each rule has preconditions and actions as well as state information. Each rule is fired only when sensor values match the preconditions of the rule. Also the state of the firing rule should necessarily match the current state. The sensor preconditions have fixed-length bit encoding with values of 0, 1, #(don't care), since sensors are binary-valued. We used the best matching rule policy to determine the next state and motor actions; choose the best matching rule among many rules to satisfy sensor conditions and the current state status, and then take the next state transition and motor actions of the rule.

Each rule has its own matching score. The matching score of a rule can be calculated by summing matching scores of each sensor. Each sensor receives a positive score +1 when a sensor value is exactly the same as the corresponding precondition of a rule, or a penalty score -10 when it is different from the precondition. If the precondition is '#', then it receives a score 0. As a result, the matching process will prefer the exact match with sensor values or the default('#') match. Figure 6.2 shows an example of scoring rules to find the best matching rule. A rule with the highest matching score is selected as the best matching rule. Also the rule should necessarily match the current state to be activated. State transition information can then be generated from the best rule to match the current sensor readings. In the rule structure, we can divide a set of state rules into several classes depending on the controller state. The state information can be seen as a medium to activate a set of rules, switching the controller between different sensorimotor couplings. In the experiments, 20 rules are used for each genome controller. At each time step, the robot reads sensor values from the modeled sensors and the control action determined by the chosen rule is applied to the motors.

Genetic operators, crossover and mutation, are used for a set of rules encoded by integer strings; especially motor actions and state number are encoded by integer strings. The rule-based state machines have a similar form to classifier systems, but they do not employ reinforcement learning such as the bucket brigade algorithm [Holland, 1975]. The reinforcement learning will have more potential of representation power than the evolutionary computation with a set of rules. In

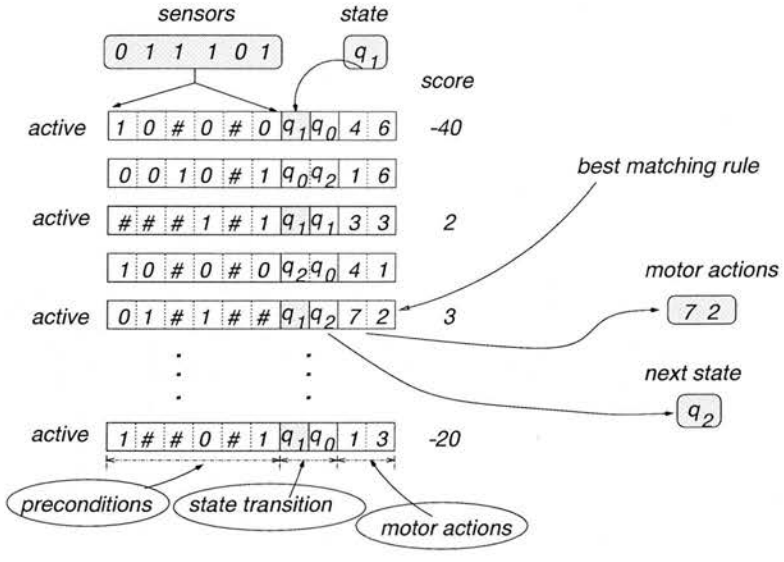


Figure 6.2: Matching process in a rule-based state machine

our experiments, the reinforcement learning will not be considered, since the dynamic weight change is beyond the power of Boolean logic networks.

In the rule-based state machines, memory amount is encoded with maximally allowed number of states for state transitions. We will describe a set of experiments that reveal behavioral and performance differences between a purely reactive controller and a memory-based reactive controller evolved for various particular tasks, and where appropriate we discuss the effect of memory on the resulting robot behaviors.

6.2 Experiments

We tested five different tasks—obstacle avoidance, wall following, exploration, box pushing, and corridor following—with two or four infrared sensors. Khepera-like kinematics simulation was used with the genetic algorithm over rule-based state machines. Crossover rate was 0.6 and mutation rate was 0.01 for evolutionary computation.

Each evolutionary run proceeds thus: N different initial positions for the robot are randomly selected in the arena in Fig. 6.3, to provide robustness in the evolved controller; evolutionary computation chooses K positions randomly without replacement[2] from the N initial positions at each generation and evaluates the new control strategies at those positions and uses the fitnesses

[2]Selection without replacement will give more chances to experience various environmental situations.

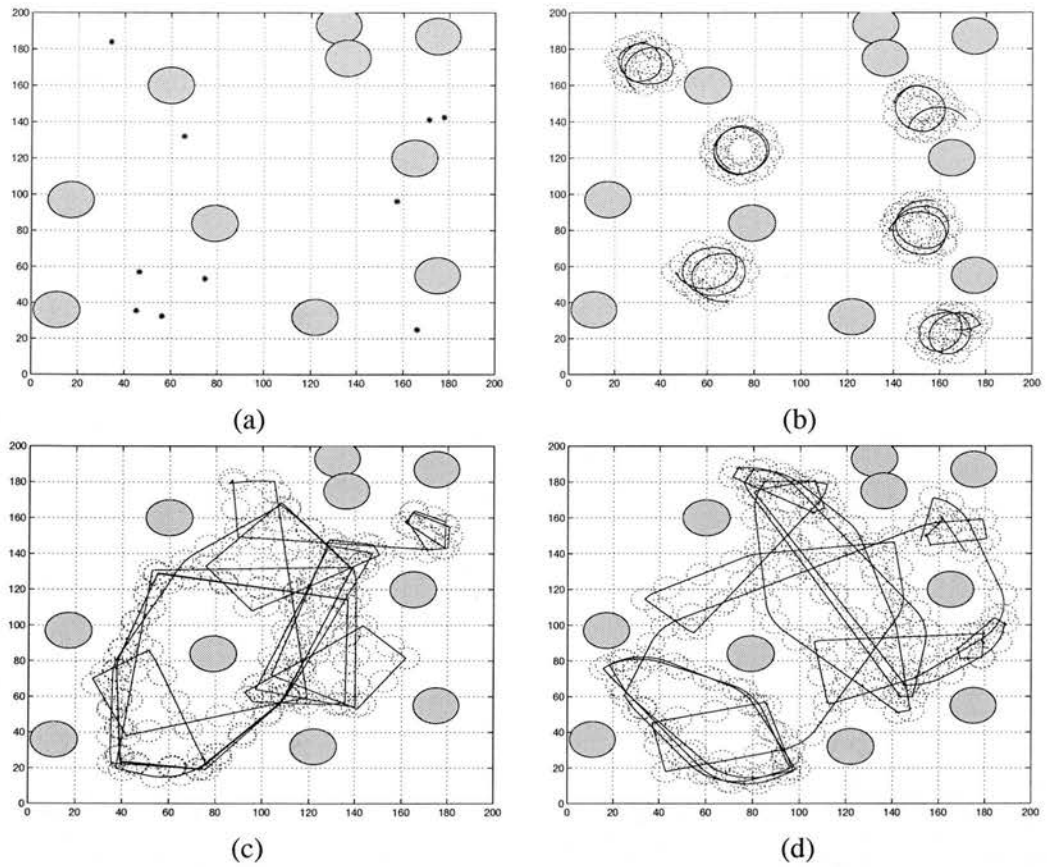


Figure 6.3: Obstacle avoidance behavior (a) arena with initial positions (b) reactive controller with 2 sensors (c) reactive controller with 4 sensors (d) 2 state machine controller with 4 sensors

thereby computed for selection of parents in the genetic algorithm; every 10 generations controllers are re-evaluated at all N positions. This N-K sample method[3] reduces the computation time, but also it has the effect of running over all initial positions regularly in the evolutionary run. We used $K = 5$ and $N = 10$ in our experiments and elitism strategy was not applied. For significance statistics, we run 10 tests for each configuration of controller structures for each of five behaviors, respectively.

6.2.1 Obstacle Avoidance

For obstacle avoidance behavior, we put cylinder-shaped boxes as obstacles randomly in the arena as shown in Figure 6.3. The fitness function F_O is defined as a penalty function as follows:

[3]This method is similar to the random subset selection by Gathercole [1998], and it will be discussed in detail in chapter 7.

$$F_O = P - \Sigma_{t=1}^{max}(\alpha(M_L(t) + M_R(t)) + \beta|M_L(t) - M_R(t)|/(2 * L) + \gamma C(t))$$

where P is the base of penalty, L is the robot wheelbase, $M_L(t)(M_R(t)$ the left(right) wheel motor output at time t, $C(t)$ is a collision detection function, and $\alpha = 0.1$, $\beta = 0.5$, $\gamma = 5.0$ are scaling coefficients for each term. If a robot collides with any obstacle, $C(t)$ will be set to one from that time onwards; otherwise it is zero. This collision factor encourages the controller to escape obstacles without collision. Usually γ is set to a high value, and when the robot experiences a collision the controller will have a very low fitness. The fitness calculates the difference between forward speed and rotation speed, to encourage motion as straight as possible. This experiment comprised controller evaluations lasting 2000 time steps, a population of 100 controllers and 4000 generations of the genetic algorithm.

For this task, memory-based controllers give a little better performance than reactive, but not significantly so. The number of sensors is more important as shown in Figure 6.3. When a robot has only two sensors, it cannot move straight forward due to collision with obstacles. It just staggers around a local area.

6.2.2 Wall Following

To evolve wall following behavior, we assumed there are many "energy tanks" in the arena the robot is exploring. If the robot finds one of these energy tanks, it acquires energy and can explore more area before its energy is exhausted. We assumed the robot loses its energy when moving. By placing the energy tanks near the walls of the environment and rewarding the agent for visiting them, we encourage the evolution of wall-following behavior: the fitness can be roughly calculated as the number of visited energy tanks and the energy level the robot has.

Thus, a fitness function F_W is a penalty function defined as follows:

$$\begin{aligned} F_W &= P - (1 - 0.7C)E \\ E &= \alpha\Sigma_i^N V(i) - (1 - C)(B - \gamma T) \\ V(i) &= \begin{cases} 1 & \text{if energy tank } i \text{ is visited} \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

where P is the base of the penalty, C is a collision detection flag, N is the number of energy tanks in the environment, E is the remaining energy after exploration, and T is the number of time steps it takes the robot to visit all energy tanks. B is the initial energy which was set to 50,

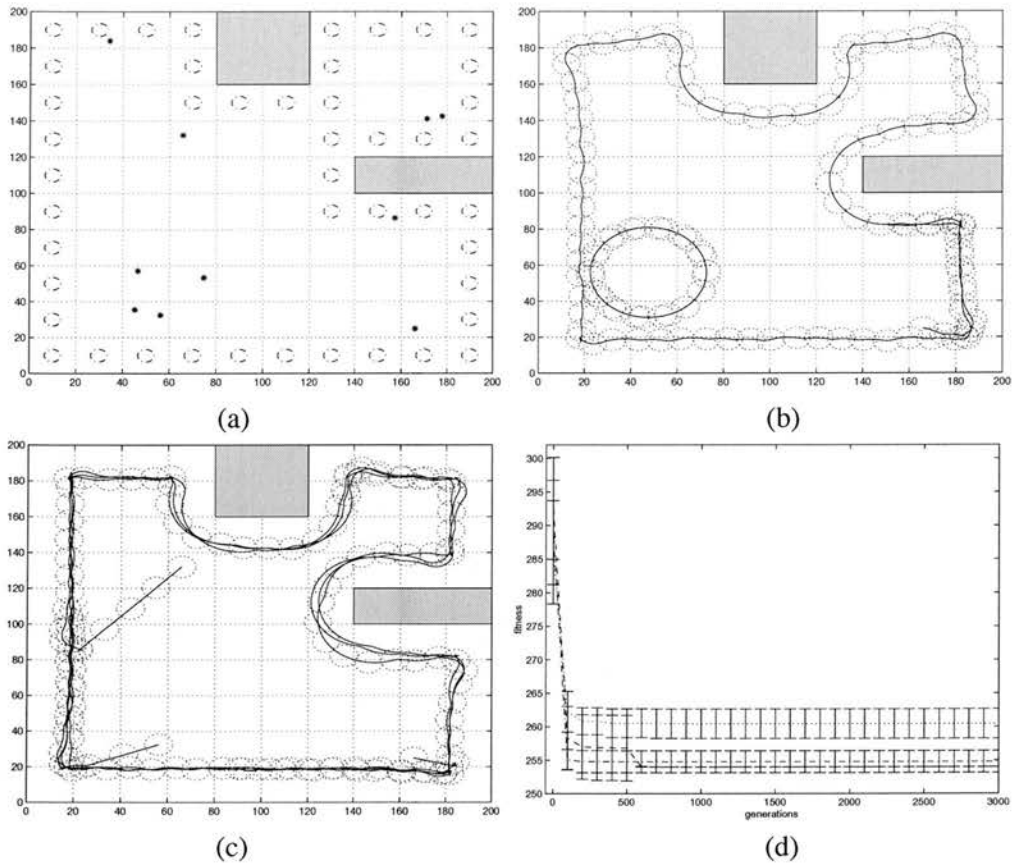


Figure 6.4: Wall following behavior (a) arena with initial positions and energy tanks (b) reactive controller (c) 2 state machine controller (d) 10 runs test

and $\alpha = 5, \beta = 1, \gamma = 0.05$ are scaling factors. We assume energy tanks are located at divisions around the wall and obstacles as shown in Figure 6.4 on a 20cm by 20cm grid (the whole arena was 200cm by 200cm).

If it collides during exploration, robot movement stops and calculates the fitness function F_W with $C = 1$, otherwise $C = 0$. Energy E is changed every time step, by losing a small proportion of energy with initial energy setting.

In this experiment each controller generated by the genetic algorithm was evaluated for 2000 time steps; the population size was 100 and the evolution ran for 4000 generations.

In this task, a 2-state machine is significantly better than the pure reactive controller—see Figure 6.4(d), where we run 10 tests for each configuration and the error bar shows 95% confidence interval. In some initial positions, the reactive controller cannot find the wall and it circles in the free space (see Fig.ure6.4(b)). The memory-based system is able to distinguish whether

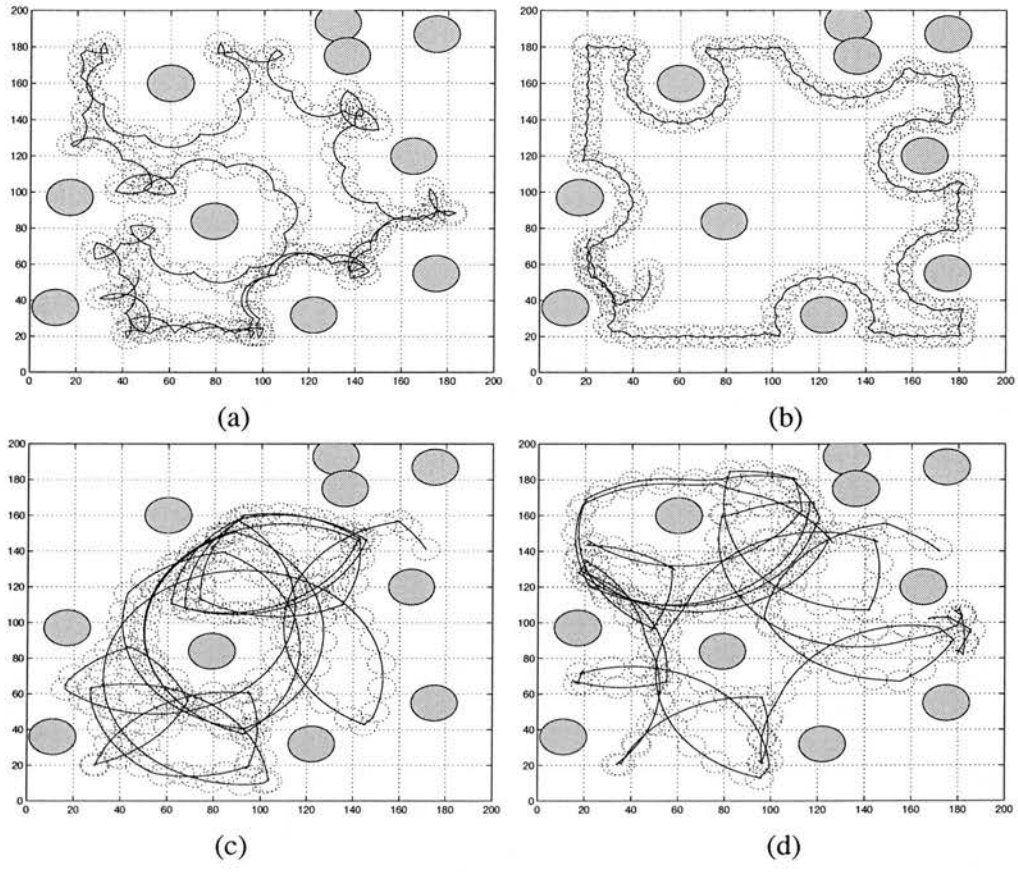


Figure 6.5: Exploration behavior (a) reactive controller with 2 sensors (b) 2 state machine controller with 2 sensors (c) reactive controller with 4 sensors (d) 2 state machine controller with 4 sensors

the robot has found a wall and to switch behavior when this occurs, as in Figure 6.4(c). The stateless controller adopts an appropriately curved path to follow a wall, but in free space such a path results in circling motions. With the 2-state controller, if the sensors read clear, the robot moves straight forward until it finds a wall; after detecting a wall the robot switches to a curved path strategy. For this task, the number of sensors does not improve the behavior, and the robot with 4 sensors shows similar movements to that with 2 sensors.

6.2.3 Exploration

For exploration behavior, we used the same environment as that in the test of obstacle avoidance behavior as shown in Figure 6.5. For this behavior, we can apply the above energy distribution idea over the whole environment for the fitness function, but we used a different calculation

method. The fitness function is defined as follows:

$$F_E = -\Sigma_i^N p(i) \log p(i)$$

where N is the number of small divisions, $p(i)$ is the probability that the robot has stayed in the i-th section and can be calculated dividing the number of time steps the robot has moved in the i-th area by the total time steps the robot runs.

Using this entropy measurement rewards the robot for visiting all areas of the environment uniformly, while the energy tank mechanism encourages the robot to visit the whole environment. As before, population size 100 with 4000 generations with 2000 time steps for each controller were used in each experiment.

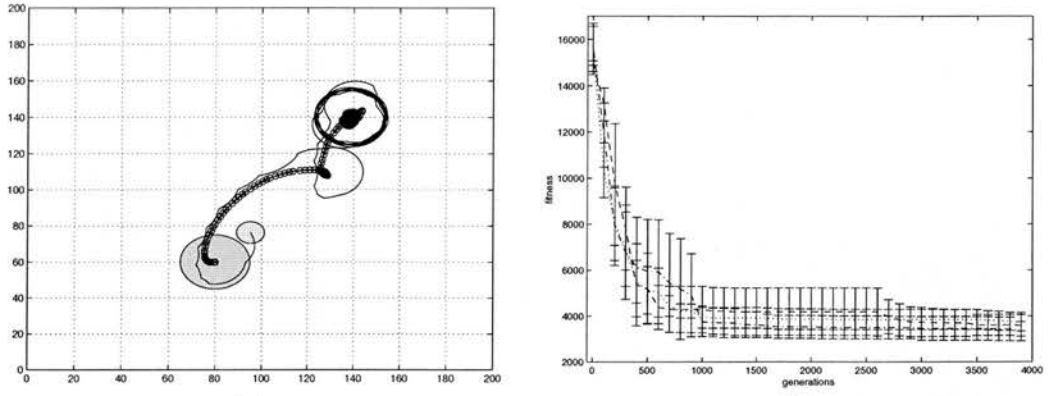
There was no significant difference between reactive and memory-based controller in terms of fitness, but the state machine was a little better than a reactive controller on inspection of its behavior—see Figure 6.5. When a robot has two sensors, it shows wall following behavior. 3-state machine and 4-state machine behavior was similar to 2-state machine and in many cases 3-state and 4-state machines evolved into a 2-state machine controller. It appears that the 2-state machine is most desirable in this behavior. The 2-state machine controller shows a very delicate wall following behavior in Figure 6.5(b). Increasing the number of sensors allows the robot to move fast around the environment, because it reduces the possibility of collision by seeing the environment well.

6.2.4 Box Pushing

We assume a circular box is located in the arena and a robot moves the box into a target position indicated by a light. The robot uses its obstacle sensors to detect the circular box and ambient light sensors to detect the light. We simplified the light sensor value to binary values to say whether the light is close to the left or right sensor, or the light is positioned behind the robot. Thus, the information of two light sensors can represent the direction of light. Control structures with a different number of memories were tested to see if the memory causes differences in the robot's performance. The fitness function F_B is defined as a penalty function as follows:

$$F_B = \Sigma_{t=1}^{max}(\alpha W(t) + \beta D(t) + \gamma C(t))$$

where $W(t)$ is the distance between robot and box, $D(t)$ is the distance between box and light, and $C(t)$ is the collision detection function. In the experiment, we set $\alpha = 0.1$, $\beta = 1.0$, and



(a) (b)

Figure 6.6: Box pushing behavior (a) 2 state machine controller with 2 sensors (b) 10 runs test with angle 45° (dotted : reactive, dashdot : 2 states, solid : 3 states)

$\gamma = 50.0$.

It was hard to evolve box pushing behavior with random initial positions—in a real application it would be combined with exploration behavior. Robots are initially positioned near box. 1500 time steps for each random gene controller were used with initial robot positions around the circular box. Robot controllers with a different number of sensors have shown similar fitness values and behaviors. We run 10 tests for each configuration and the error bar shows 95% confidence interval. The fitnesses of reactive, 2-state, 3-state, and 4-state machines are not significantly different as shown in Figure 6.6.

6.2.5 Corridor Following

For this task, we use a similar fitness method tested in wall following behavior. We assume energy tanks are located on every 20cm by 20cm grid division without obstacle. We want to design a robot controller to explore all empty places and pass through even narrow corridors. Thus, energy-tank resource search method will encourage robots to visit all empty grid areas. The fitness function F_C is a penalty function defined as follows:

$$\begin{aligned} F_W &= P - (1 - 0.7C)\alpha\Sigma_{i=1}^{N} V(i) \\ V(i) &= \begin{cases} 1 & \text{if energy tank } i \text{ is visited} \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

where P is the base of the penalty, C is a collision detection flag, and N is the number of energy tanks in the environment.

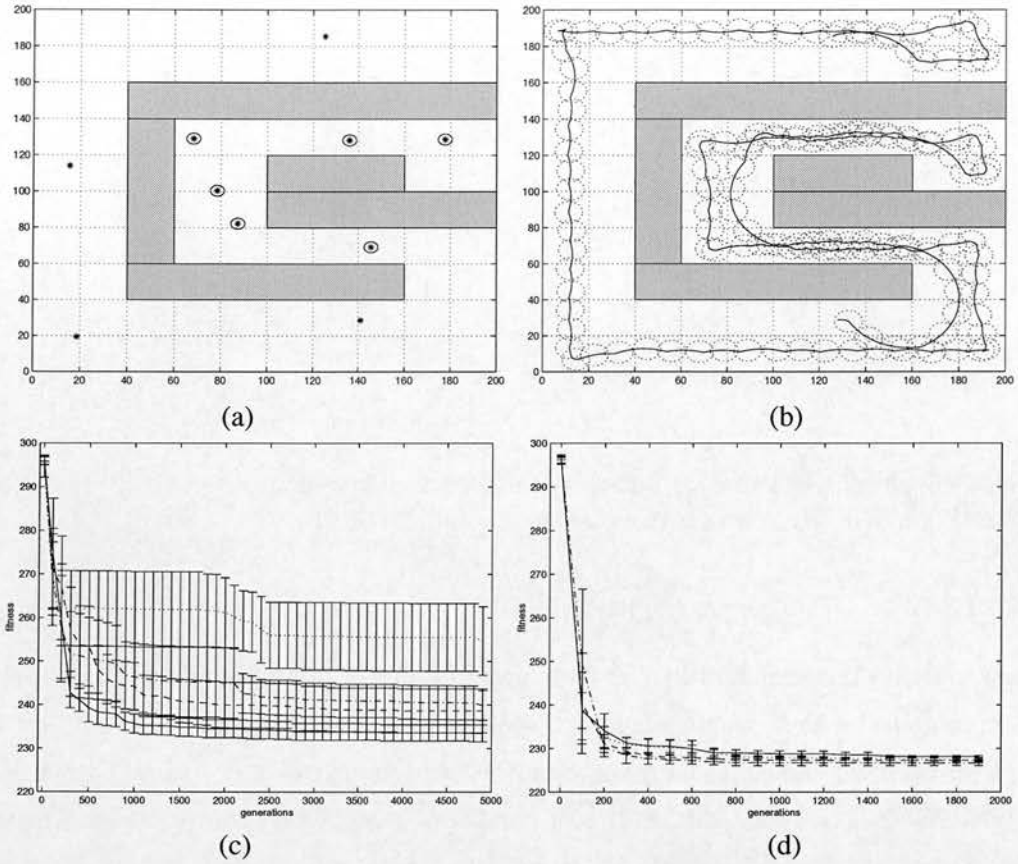


Figure 6.7: Corridor following behavior (a) corridor environment with 10 initial positions (b) 2 state machine controller with 2 sensors (c) 10 runs test with 2 sensors (d) 10 run test with 4 sensors (dotted : reactive, dos-dashed : 2 states, dashed : 3 states, solid : 4 states

In this experiment each controller generated by the genetic algorithm was evaluated for 1000 time steps; evolution ran for 5000 generations with the population size 100.

When we run tests with 10 initial positions, they tend to have local minima of fitnesses. Tests with 4 initial positions outside the narrow corridor shown in Figure 6.7 can easily get the low fitness values; robot wanders around the wall with those initial positions but are reluctant to move inside the narrow corridor, because it has more probability to collide with the wall. *N-K* sampling method for starting positions produced few results that tests with all 10 initial positions identified as good controllers.

Instead we used six difficult positions located inside corridor among 10 initial positions. In this corridor following task, a 2-state, 3-state, 4-state machine is significantly better than the pure reactive controller when only 2 sensors are used— see Figure 6.7(c). All six positions are checked for each controller configuration and the error bar shows 95% confidence interval.

In some initial positions, the reactive controller cannot pass all grid areas or it experiences collision inside corridor. The memory-based system use its memory bit very efficiently to adjust their turning angle carefully. It shows more careful movement in a narrow corridor than reactive controller. As a result, it reduces the probability of collision with the wall and the robot can pass through narrow corridor with more safety.

For this task, the number of sensors improves the behavior, and the robot with 4 sensors can see the environment well and reduce the collision probability. To evolve controllers with 4 sensors shows better performance than those with 2 sensors as shown in Figure 6.7(c)-(d). Its convergence speed was faster and test fitnesses were improved as well. In case with 4 sensors, the fitnesses of reactive, 2-state, 3-state, and 4-state machines are not significantly different.

6.3 Summary

In this chapter, robotic tasks were tested to see the memory effect. A simulation program is developed to model a Khepera-like robot. It has two or four infrared sensors, two light sensors and two wheel motor actions. Instead of allowing continuous-valued sensors, the sensor values were binarized to obtain Boolean logic networks for robot controllers.

Using the representation of binary sensor readings, rule-based state machines were applied to control structures. In rule-based state machines, a fixed-memory controller is represented as a collection of rules conditioned on the sensory state and on the controller's internal state. The rules thus implement the transition function of a finite state machine, and are similar to a classifier system [Goldberg, 1989]; they provide an economical way of representing the controller for evolution using a genetic algorithm. The controller structure originates from finite state machines and the genetic representation is a multivalued string instead of a binary string. This scheme reduces the length of genes. In the experiments, the chromosome representation follows an integer encoding for output action and state number.

In the experiments, we have seen the potential of memory-based systems with simplified robot models. Infrared sensor readings are similar to whisker sensors. Some robotic tasks such as wall following behavior utilize internal states to decompose them into subtasks or sequentialize motor actions. The internal states are also influenced by the number of sensors in a robot. When the number of sensors is increased, it has a tendency of reducing the necessary memory size for tasks.

For robotic tasks, experience of various environmental states is necessary for evolving con-

trollers. There exist many situations in an environment or several environments that robots should handle. The computation time for evolving is proportional to the number of such sample situations. Thus, N random possible positions are collected and during the evolutionary process, K samples are selected among N possible positions for evaluating one genome. This N-K sampling is an efficient method in obtaining desirable controllers using evolutionary algorithms – this will be discussed in detail in the next chapter.

In reactive systems, mobile robots work efficiently through a tight state-free coupling between perception and action. We investigated the effect of allowing the controller small amounts of memory in five different behaviors, obstacle avoidance, exploration, wall following, box pushing, and corridor following. In simple behaviors such as obstacle avoidance and exploration, a reactive system without memory elements is sufficient to control a mobile robot. The number of sensors has more influence on performance of such behaviors than the amount of state available.

In some behaviors, such as wall following and corridor following, the memory element plays an important role by allowing the controller to specialize its behavior depending on context—for instance, free-space versus close to a wall. The amount of memory available is not significant for these tasks as tested here, because the tasks themselves only require a couple of states to be distinguished. Adding more memory does not improve performance.

In this chapter we explored the effect of adding known amounts of memory to purely reactive systems in a variety of tasks. Using a rule-based state machine approach, we construct controllers for a simulated robot for five tasks—obstacle avoidance, wall following, exploration, box pushing, and corridor following—with two sensor configurations using evolutionary computation techniques, and compare the performance of stateless and memory-based controllers. For obstacle avoidance and exploration no significant difference is observed; for wall-following and corridor following, stateless controllers are significantly worse than memory-based but increasing amounts of memory give no significant increase in performance for wall following and corridor following. The need for memory in these cases reflects a need to discriminate sensorimotor contexts to effectively perform the task.

Chapter 7

Robot Model and Sample Selection Method in Noisy Environments

In this chapter, realistic simulation methods and the evolutionary techniques are investigated. The study suggests an efficient methdology which will be applied to the next chapter. Sensor readings and motor actions in the simulation are subject to noise, since real robotic environments cannot avoid noise. Noise normally makes robotic tasks more difficult and so evolving controllers in noisy environments requires much effort to find robust controllers.

A real Khepera robot is first modeled with the same configurations in a simulation program for evolutionary algorithms. The sensor readings in a real robot are sampled to support realistic simulation. Unlike whisker-like binary sensors tested in chapter 6, infrared sensor readings have a continuous range with cone bearing. Also random noises are added to sensor readings and motor actions.

This chapter also discusses several other issues, including sample selection, the design of robust controllers and elitism strategies in evolutionary computation. The sample selection issue concerns how samples in the N-K sample method can be selected. Two methods, random selection and dynamic selection [Gathercole, 1998; Lee, 1998], are applied to evolve robotic controllers. Multiple evaluations on a genome controller are considered to avoid noise-sensitive results, and the average or the worst case of evaluations are suggested to produce robust controllers in noisy environments. The issue of the N-K sample method to support robustness is also related to the elitism and it will be explained how the elitism can be applied in the N-K sample method.

As an example robotic task, wall following behavior is tested with various reproduction selec-

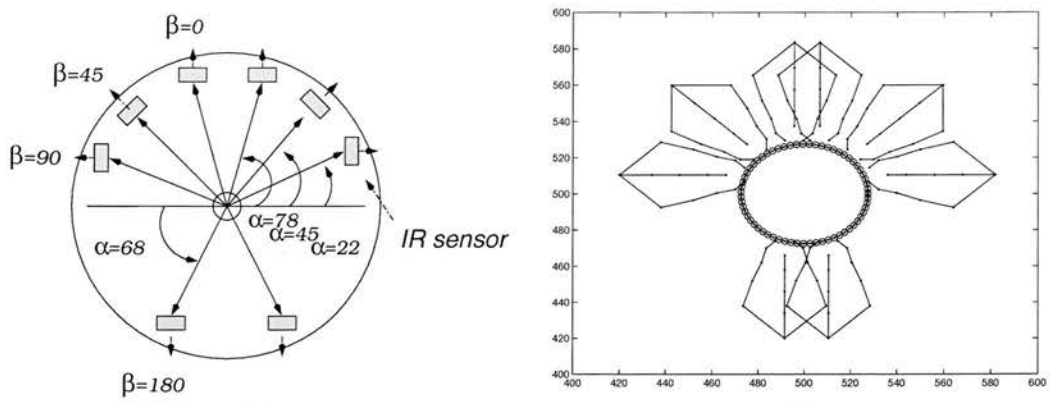


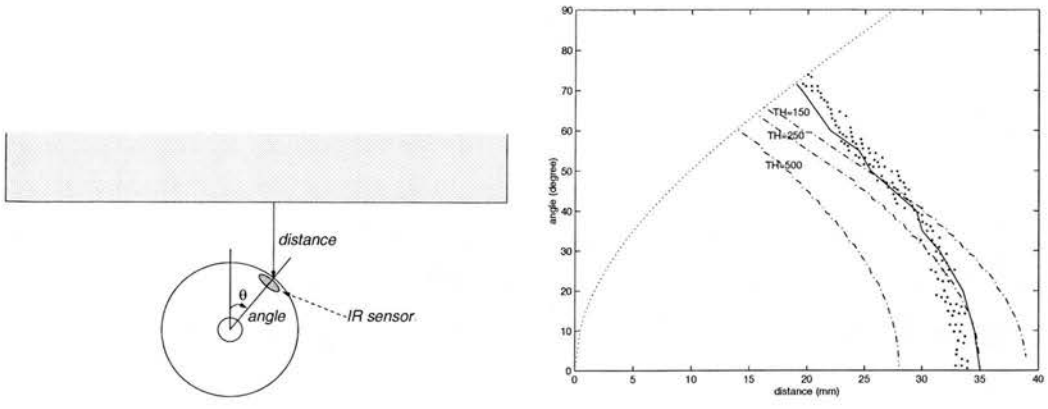
Figure 7.1: Khepera robot simulation model (a) robot sensor configurations (α: angular position of each sensor on the circular board, β: angular direction of each sensor) (b) cone-bearing area of each infrared sensor

tion mechanisms, sample selection methods, robust fitness methods and elitism. The desirable methods suggested in the result will be applied to various robotic behaviors in the next chapter.

7.1 Robot Models

In the previous chapter, binary sensors like whiskers are modeled for infrared sensors of a robot. A more realistic simulation method is required to see real robot behaviors. Instead of whisker-like sensors, the sensor range for each infrared sensor will be shaped with a cone-bearing, because the real infrared sensors cover almost cone-shaped areas. Figure 7.1(a) shows the detection ranges and angles of eight infrared sensors in a real robot. The developed simulation program will follow such real robot configurations. An example of sensor ranges with cone-bearing angles in a simulation program is shown in Figure 7.1(b). It models 40 degrees bearing angles with a set of 30 sensor points in a cone area. The sensor values are computed by estimating the detection point and its appropriate sensor value with the 30 sensor points. A random noise $\pm 10\%$ is added to the sensor values. The wheel dynamics are also subjected to random noise. A random noise of $\pm 10\%$ is added to the motor speeds for the two wheels, and the direction of the robot is influenced by $\pm 5\%$ random noise.

In simulation, the world environment has a binary map to help sensor processing detect any object easily. Instead of calculating each sensor detection area directly, objects and walls are masked with one in the environment array. If one of the 30 points which belong to an infrared sensor is overlapped with any masked area, its proper sensor value is recorded; each of the



(a) (b)

Figure 7.2: Infrared sensor readings of Khepera robot (a) diagram of measurement (b) sensor readings in simulation and a real robot (dotdashed line: whisker-like sensor readings, solid line: sensor readings in real robot, dotted line: limit boundary (sensors cannot be tested above the boundary, since a robot collides with the wall), points: simulated cone-bearing sensor readings with noise)

sensor points has the magnitude of the proximity sensor at the corresponding distance[1]. From that point, a more exact sensor value is estimated by measuring the interpolated detection point with any object.

To observe sensor values in realistic simulation, the sensor reading method by whisker-like sensors is compared with that by cone-bearing infrared sensors. We assume that a robot is positioned near a wall and then the direction and distance of each sensor to the wall is measured with sensor readings as shown in Figure 7.2(a). First, whisker-like infrared sensors are assumed to have one ray reflection to detect any obstacle and their obstacle detection will depend on its cutting threshold to binarize its sensor readings. The binary values will say whether or not an object is detected with the sensors. If one sensor value is greater than the given threshold, then we can say that an object is detected with the sensor. Otherwise, it is recognized as there is no obstacle in front of the sensor. Figure 7.2(b) shows three dashdot curves for detecting the wall with threshold 150, 250 and 500; infrared sensor values range from 0 to 1023.

The sensor readings on a real Khepera robot are measured by changing the distance and angle of the robot to the wall. The cutting threshold 500 is assumed to detect walls and its detection range is displayed in Figure 7.2(b) with a solid line. Simulated cone-bearing sensors are shown together with many dot points. It is assumed that they have $\pm 10\%$ noisy signals and its cutting threshold for detection is still 500. The dot points are very close to real sensor readings by

[1]This point scanning method was used with 15 sensor vectors in Khepera simulation package 2.0 [Michel, 1996]

forming a cloud around the detection line of a real sensor, while whisker-like sensors cannot approximate the real sensors by even changing the cutting threshold.

The motor actions for both left and right wheels are restricted to the set $\{-8, -6, -4, -2, 2, 4, 6, 8\}$ and only eight possible actions are allowed for each wheel. They thus have 64 possible combinations for two wheel motors. For neural network controllers, the motor actions will have integer values ranging from -8 to 8; neural network outputs are continuous but separated into integer values, since the real Khepera robot control uses integer values instead of real values.

7.2 Experimental Issues in Realistic Simulation

In realistic simulation, noise is involved in the processing of sensor readings and motor actions. A random noise significantly influences the performance and even the same controller may produce remarkably different results on different occasions. Unless a controller can handle noisy signals appropriately, it increases the possibility of experiencing collisions with obstacles. The evolutionary robotics approach may evolve robust controllers with rare collisions. The performance will be compared for various controller structures in the noisy world. As shown in grid world problems in the previous chapter, it will be investigated if memory-based controllers are superior to purely reactive systems. The potential of controllers evolved in noisy world environments is to produce desirable controllers indirectly for the real world application.

We define a *sample* set over starting positions as a set of trial positions used while evolving. It can be positions over several environments depending on how many environments are used for robot experiences. In evolutionary computation, we cannot test all possible positions and directions of robots even in one environment. Instead we assume randomly chosen situations will represent the whole set of environmental situations. Thus N random positions are used on behalf of the whole environment. In evolutionary robotics, robots need to experience various environmental states. For every generation each chromosome can be tested for its fitness over all N positions. If N is large, the evolutionary process will need much time, since the computing time is proportional to the number of trials, N as seen in equation 4.2. In the previous experiments in chapter 6, the N-K sample method was applied to reduce computation time and also cover the whole set of positions. In the experiments K samples were randomly selected among N positions for each generation and the fitness of each chromosome was measured over K sample positions. It is supposed that the random selection of K positions will most likely give an equal probability to each of the N positions.

In the robotic experiments, a robot can be placed at any position and direction to test its robustness in the environment. Many tasks, for example, wall following, obstacle avoidance and corridor following, will require such robustness. Some evolved controllers have shown they are successful only at a few positions but they easily collide with obstacles at difficult situations. In evolutionary experiments, we assume that the robot should be successful, if possible, at any random starting position to obtain robust controllers for a robot behavior. Thus the design of such desirable controllers should be linked with fitness in the evolutionary approach.

The above problems are related to the following two issues:

- How to handle robust controllers in the noisy world environments
- How to select sample positions among noisy environmental situations

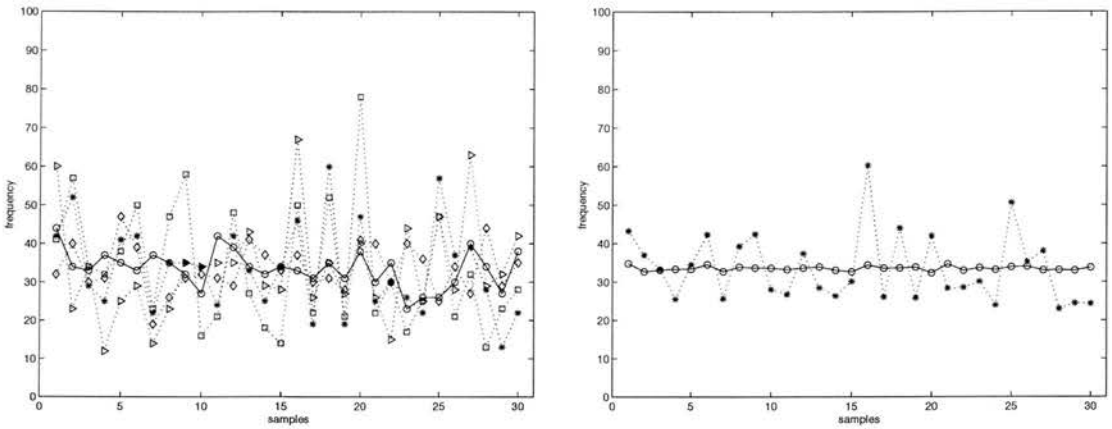
The issues ultimately determine how to design robust controllers efficiently. Before several tasks are evolved for memory analysis, the efficiency of evolutionary mechanisms for the selection of trial samples (sample positions) is studied. This selection problem has been mentioned as *subset selection* in the application of evolutionary algorithms to classification problems [Gathercole and Ross, 1994; Gathercole, 1998]. Lee [1998] did some systematic investigation of the subset size using the effort measure in his robotic experiments. For evolutionary robotic experiments in this thesis, we will call it *sample selection* or *trial selection.*

7.2.1 Sample selection

In the previous chapters, the N-K sample method was applied to the Tartarus problem and the robotic tasks, instead of evolving controllers with the whole N positions. Those tasks have noisy-free environments and the random subset selection scheme [Gathercole, 1998] has been used; samples with the subset size K are randomly selected among N positions.

There is an issue of how to select K samples among N positions for every generation. Gathercole and Ross [1994] developed a dynamic subset selection method for evolutionary algorithms, which can be used to choose training examples probabilistically in classification problems. The method gives more priority to the training samples which are difficult to classify or have not been selected. At each generation, the subset size, difficulty and age terms are handled to compute the appropriate probability of selection for training cases. They showed that the dynamic subset selection method was better than random subset selection.

Motivated by their work, a dynamic sample selection method is suggested for evolutionary



(a) (b)

Figure 7.3: Distribution of samples using sample selection method (a) examples of frequency distribution (dotted: dynamic selection, solid: random selection) (b) mean of frequency distribution over 25 runs (dotted: dynamic selection, solid: random selection)

robotics problems in addition to random sample selection. While the dynamic subset selection [Gathercole and Ross, 1994] uses a mixture of difficulty and age to generate a selection, the dynamic selection in the thesis will only consider the difficulty level of each sample. Every generation the chromosome with the best fitness over the K positions is collected. This chromosome is then evaluated over N positions and it will show a spectrum of fitness from difficult positions to easy positions to be evolved. From the fitness distribution, sample positions are selected by giving more chances to difficult positions. Those selected sample positions will be applied to a population of chromosomes in the next generation. In this selection scheme, worse fitness will have more probability to be selected as a sample position, using the roulette wheel method over fitness. If a few positions have bad performance, the evolutionary process will focus on those positions so as to improve the performance on the N positions. The roulette-wheel selection in terms of fitness will choose samples with replacement and thus it dynamically decides K samples. We will call this selection method *dynamic sample selection.*

While evolutionary process progresses with the dynamic selection, each sample of N positions has been selected many times, depending only on fitness distribution of samples. The dynamic sample selection does not include the age term of how long samples have not been selected in the evolutionary process. It might happen never to select some subset of the N samples. However, it is presumed that it will depend on what kind of N samples are collected. Some sample may be unselected with dynamic sample selection, when the sample is quite similar to another and it is easy to evolve on.

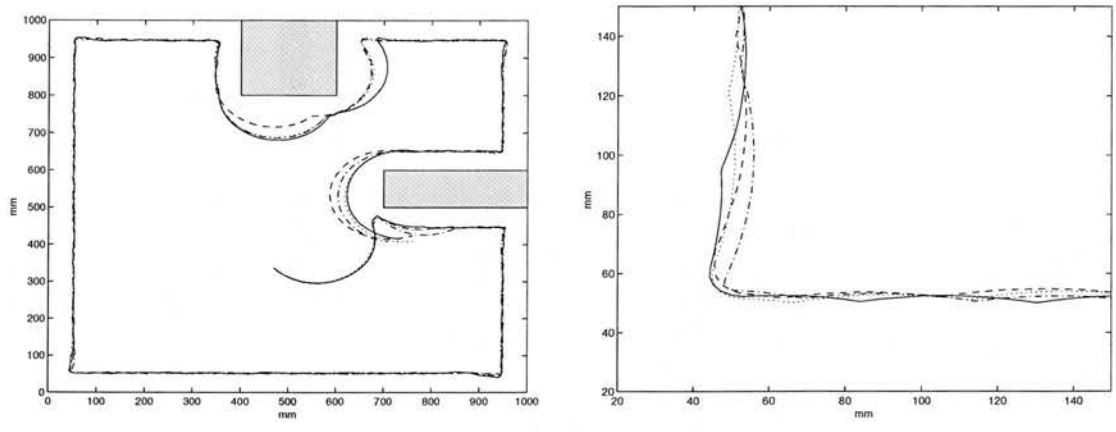


Figure 7.4: An example of different trace results on the same controller in a noisy world

In the experiments of wall following in noisy world, N sample positions were randomly chosen in the environment. Each sample received many chances to be selected without considering the age term, even though the frequency distribution of the dynamic selection was not so uniform as the random selection. Figure 7.3 shows the frequency distribution of wall following experiments on $N = 30$ sample positions with $K = 5$.

7.2.2 Robustness in the noisy world

The performance for one genome over N positions is measured as follows:

$$F = \frac{1}{\sum_{k=1}^{N} w_k} \sum_{j=1}^{N} w_j f_j$$

where f_j is a fitness for the robot starting at the j-th sample position and w_j is the weight parameter depending on the level of importance. F will give the weighted average performance at any position among N starting positions. We will assume the weight $w_j = 1$, for $j = 1,...,N$ in the experiments.

The world environment is so noisy that every run of the same controller has different trace results (see Figure 7.4). In this situation, an evolutionary result for one run is not reliable. Even the best controller cannot guarantee that it will work correctly with the same performance. To avoid this problem, one controller should be tested over all starting positions with many runs. If it shows stable results, it is reliably accepted as a desirable controller in a noisy world. However, such a test with many repeated runs will be computationally infeasible.

The controller chromosomes in a population are evolved with the average fitness over sampling

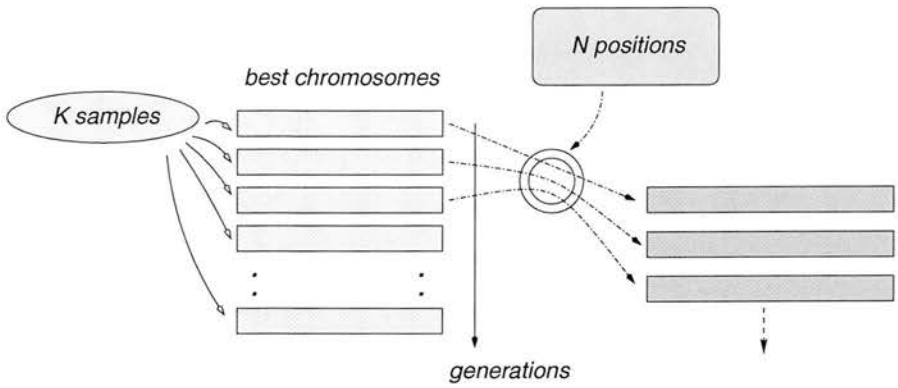


Figure 7.5: N-K sample method

positions. They are not evolved for the whole set of robot situations. In the N-K sampling approach, the best chromosome for the K positions is not exactly the same as the best chromosome for the whole N positions. Every generation the best chromosome for the K positions is collected and then tested for the whole N positions. Figure 7.5 illustrates an example of the N-K sample method in the noisy world. In the previous N-K sample experiments without noise, chromosomes are evaluated for the N positions intermittently instead of every generation; for example, for the Tartarus problem[2], the best chromosome for K positions is evaluated over N positions every 50 generations and for robotic tasks in chapter 6, all chromosomes in a population are evaluated over N positions every 10 generations and then the best chromosome is recorded.

To build robust controllers, we cover three issues, robust fitness, robust N-K sample method and elitism for robust controllers.

- Robust fitness

If the fitness result for the N positions is the best case so far, then it keeps the best chromosome. For a noisy world, every run of experiments will have a different fitness result. The best chromosome in a population is evaluated several times and its average score will represent the fitness of the chromosome.

$$F_{average} = \frac{1}{m}\sum_{i=1}^{m}\left(\frac{1}{\sum_{k=1}^{N} w_k}\sum_{j=1}^{N} w_j f_{ij}\right) \tag{7.1}$$

where m is the number of runs for the same controller and f_{ij} is a single run fitness for the

[2]The Tartarus problem has a large $N = 10,000$ and the whole population was not evaluated on the N positions.

no.	computing cost	method
1	$pK+Nm$	Genomes are evaluated on K positions once
2	$pN+Nm$	Genomes are evaluated on N positions once
3	$pKm+Nm$	Genomes are evaluated on K positions multiple times
4	pNm	Genomes are evaluated on N positions multiple times

Table 7.1: Four possible schemes in the N-K sample method (p: population size, m: number of trial runs)

j-th sample. For noisy world, $|f_{aj} - f_{bj}| \geq 0$ for the a-th and b-th run, $1 \leq a,b \leq m$, while $|f_{aj} - f_{bj}| = 0$ without noise.

The other way is to record the worst case performance instead of the average score of m runs. Among m runs, the worst case score will consider the worst scenario which may happen when the controller is taken. It chooses the worst case result with the minimum fitness[3].

$$F_{worst} = \min(\sum_{j=1}^{N} r_j f_{1j}, \quad \sum_{j=1}^{N} r_j f_{2j}, \quad \ldots, \quad \sum_{j=1}^{N} r_j f_{mj}) \tag{7.2}$$

where $r_j = \frac{w_j}{\sum_{k=1}^{N} w_k}$. This measure will play a role of avoiding noise-sensitive controllers.

- Robust N-K sample method

In the robotic experiments, five runs are applied for one single controller on N positions and the robust test with multiple evaluations is restricted to the best controller in a population[4]. The average or worst score of five runs[5] for the best controller is recorded. If the average or worst fitness over five runs is bad, then the controller is regarded as sensitive to noise and it is automatically discarded.

The following schemes can be applied for the N-K sample method to find robust genome controllers.

- Genomes in a population are evaluated on K positions once, but the best chromosome is evaluated again on N positions multiple times.
- Genomes in a population are evaluated on N positions once, but the best chromosome is evaluated again on N positions multiple times.
- Genomes in a population are evaluated on K positions with multiple trials, but the best

[3]If the penalty fitness is assumed, the run with maximum fitness will be the worst case.

[4]All the chromosomes in a population can be evaluated with multiple trials, but the number of trials significantly influences the computation time.

[5]Five runs may not be enough to obtain robust controllers, but experimentally, it shows a reasonable result, even though it cannot perfectly catch noisy-sensitive controllers in some cases.

chromosome is evaluated again on N positions with multiple trials.

- Genomes in a population are evaluated on N positions multiple times and find the best chromosome.

The computing cost for each scheme is displayed in Table 7.1. The best strategy for robustness is the fourth method, but it requires many evaluations (pNm) for one generation; strictly speaking, the second and the fourth method may not be seen as sample selection schemes. The first strategy is most economical way of reducing the computing cost, even though it may neglect the chance of some chromosome to be the best chromosome for the N positions. In our experiments, the first strategy shown in Figure 7.5 was mainly taken to find robust controllers[6].

- Elitism in the robust N-K method

One important factor in evolutionary robotics is related to elitism. The effect of elitism has been emphasized for evolutionary strategies in chapter 4 (section 4.4.2). In the N-K sample approach, the best genome in a population, which has the best fitness for the K sample positions, is re-evaluated for the whole N positions with multiple trials, using equations 7.1-7.2. With this process, the best chromosome for the N positions and its best fitness can be checked every generation for elitism. The elite is then inserted into a new population of the next generation. This has a tendency to accelerate the convergence speed to the optimal control structures. The experiments in this chapter will examine the effect of such elitism in a robotic task.

The Evolutionary Multiobjective Optimization (EMO) approach has produced feasible solutions to analyze fitness performance in terms of memory size (see chapter 4-5). EMO concurrently searches for the best controllers for each number of states. If the N-K sample method is used with the EMO approach, the chromosomes in a genetic population are evaluated once over K positions[7]. There are variable state machines in a population of chromosomes. The best chromosomes for each number of states are collected and then they are evaluated *multiple times* over N positions to support robustness. If they are better than the previous best chromosomes, they are stored. Otherwise, they are discarded. Through this process, the best chromosomes for each number of states will be updated every generation and thus a pool of the best chromosomes will be maintained. For every generation, one of the best chromosomes in the pool will be selected for the elitism where a state number is randomly chosen.

[6] In most experiments coming, only the best chromosome for the K positions receives multiple trial evaluations. For the T-maze problem in the next chapter, every genome in a population is evaluated multiple times.

[7] This can be modified with multiple evaluations over K positions or N positions. It corresponds to one of the methods in Table 7.1.

7.3 Experiments with Wall Following

The robust N-K method is studied with a robotic task, wall following behavior. It will investigate desirable methods for elitism, reproduction mechanisms and sample selection methods. The choice of methodology will be applied to various robotic behaviors in the noisy world in the following chapter.

When wall following behavior is tested in the previous chapter, energy tanks are placed near walls and robots are evolved to visit as many tanks as possible. In the same way, it is assumed that energy tanks are placed around walls and robot agents are trying to visit energy tanks.

The fitness is estimated with the number of energy tanks that a robot agent has visited. If a robot agent can visit all energy tanks within a limited amount of time, we can say that the agent is successful for the wall following behavior. Random noise will increase the possibility of colliding with walls. The collision will have a high penalty and thus the fitness is defined as a penalty function as follows:

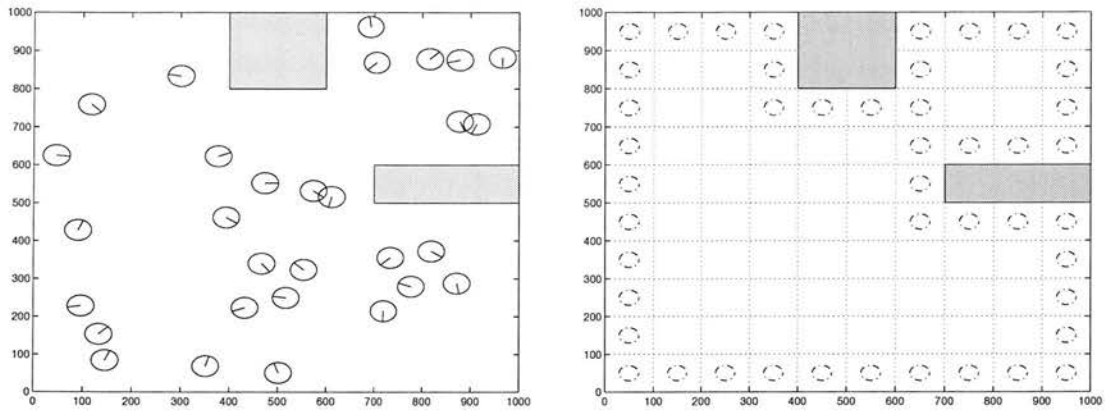
$$f_W = P - V + C(T - t_c) \tag{7.3}$$

where P is the base of the penalty ($P = 100$ is set), V is the number of energy tanks that a robot has visited, and T is the maximum time steps for exploration, which is set to 1200 time steps in the experiments. If collision occurs, the exploration stops and the fitness is calculated with the collision time t_c. C is the binary collision flag to say if a robot has collided with any obstacle. The arena size is 100 cm by 100 cm. The environmental configurations are similar to Figure 6.4 and the arena has 46 energy tanks. The optimal fitness will be 54 (=100-46) as the least lower bound.

7.3.1 N-K Sample Experiments

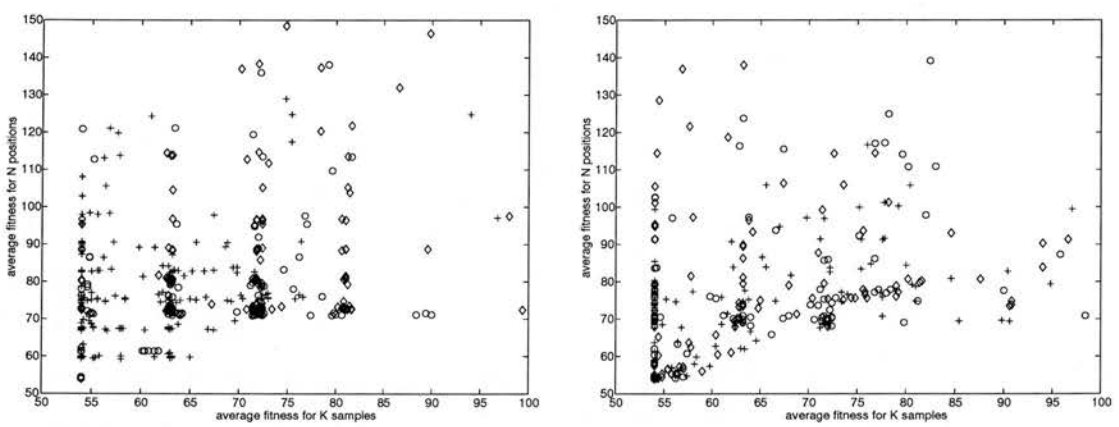
In this experiment, 30 random positions are selected as starting positions as shown in Figure 7.6. Using realistic simulation, sensor readings and motor actions are influenced with $\pm 10\%$ random noise. 200 generations and population size 50 were used for evolutionary experiments. Desirable robot controllers are required to have correct wall following behavior for every starting position. We applied the N-K sample method with $N = 30, K = 5$ to this problem.

In the initial generations, the robots were successful at following walls at only a few starting positions. As the evolutionary process continues, successful controllers were found to follow



(a) (b)

Figure 7.6: World environment for wall following (a) 30 random positions and directions (b) energy tanks

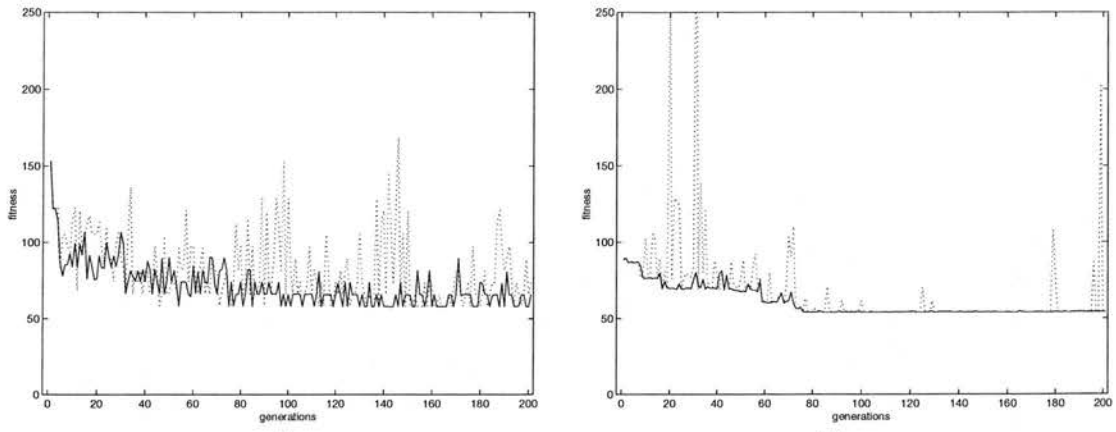


(a) (b)

Figure 7.7: Examples of average fitness of the same controller over $N = 30$ positions and $K = 5$ samples (each symbol represents an evolutionary run)

walls, starting at all positions. In some cases, the evolutionary computation had locally optimal controllers, following walls at almost all positions except a few difficult positions. The optimal controllers robustly follow walls at every starting position even with noise. To support robustness, the best chromosome in a population is evaluated over $N(= 30)$ positions five times. It is often observed that controllers produced very different fitness results, starting even at the same positions; they may correctly follow walls or collide with obstacles depending on noise states. This indirectly indicates that five runs may be insufficient to detect noise-sensitive controllers.

Figure 7.7 shows the fitness distribution of the best controllers for each generation. They are evaluated five times over K samples and N positions to see the correlation. Normally the best



(a) (b)

Figure 7.8: Examples of fitness curves in noisy environments (a) bad controller (b) good controller (dotted: the best chromosomes for K samples, solid: the best chromosome for N positions)

chromosome for the K positions is not exactly the same as the best chromosome for the whole N positions and also the fitness performance between two evaluations may not be proportionally correlated as shown in Figure 7.7. The correlation, of the average fitnesses over K samples and N positions, is very low in many cases; however, performances with better fitness show a little higher correlation. Figure 7.8 also shows the gap of performance over the K samples and N positions. The best chromosome over K samples does not guarantee to have optimal solutions over N positions, even though it has potential. This problem can be overcome by using the elitism for robust controllers.

The best performance over N positions is checked for every generation and the best chromosome over N positions with multiple runs will be used for the elitism. In Figure 7.8, the best chromosome over K samples for each generation is evaluated over N positions five times. The average score in equation 7.1 is drawn with dotted lines and shows much fluctuation. The chromosome with the best value of the average score is stored as the elite while the evolutionary runs continue. The elite chromosome is separately evaluated over N positions five times; it is assumed that this chromosome is the best for N positions by keeping the best value chromosome using equation 7.1. The fitness evaluation of the elite is shown in Figure 7.8 with solid lines and the average of five evaluations over N positions with the elite genome is not constant, either. The fitness curves for the best chromosomes over N positions still have a fluctuation of fitness, but it is supposed that desirable controllers should have a small fluctuation of fitness, regardless of noise as shown in Figure 7.8(b). Using the idea, good controllers can be chosen by checking whether the fitness curves are stable without much fluctuation.

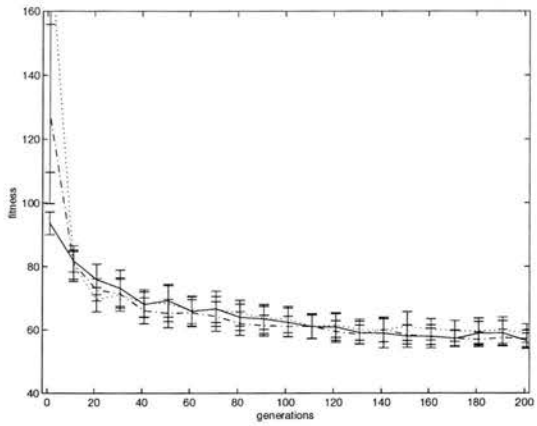


Figure 7.9: Fitness curves with different sample size (dotted: $K = 5$, dashdot: $K = 10$, solid: $K = 30$)

sample size	# success	effort cost
$K = 5$	15	$[1.30C_5, 2.46C_5]$
$K = 10$	17	$[1.20C_{10}, 2.07C_{10}] = [2.42C_5, 4.15C_5]$
$K = 30$	19	$[1.21C_{30}, 1.77C_{30}] = [6.79C_5, 10.65C_5]$

Table 7.2: Sample size and the effort cost interval in N-K sample method (C_i is a unit computing cost for a single run, $C_{30} = 6C_5, C_{10} = 2C_5$)

Figure 7.9 shows the average fitness result over N positions for three different sample sizes $K = 5, 10, 30$. Two state machines were tested for the experiments with fitness-proportional selection and elitism. For each experiment, 25 runs were used to evolve controllers. In terms of fitness test, they were not significantly different. Thus, the effort test can be checked to see the performance. If a success of a single run is defined as the case in which the average fitness is 60 or below, $K = 5, 10, 30$ have 15, 17, 19 successes, respectively. Table 7.2 shows that increasing sample size tends to increase the computing cost with a small improvement over the number of successes. When the computing cost of each experiment is transformed into the cost for 5 samples, the samples with $K = 10, 30$ are not as effective as $K = 5$. Thus, the experiments with N-K sample method will use $K = 5$ by default.

7.3.2 Sample Selection and Robustness Experiments

Fitness-proportional selection and tournament-based selection are popular reproduction methods in evolutionary algorithms. In the experiments, we are interested in observing the effect of dynamic sample selections. It is also an important matter in evolutionary robotics as well as in classification problems. As mentioned above, two sample selections are applied: one is the

chromosome reproduction	sample selection	robustness	elitism	# success	effort cost
tournament-based	random selection	average case	no	12	$[1.50C, 3.34C]$
			yes	11	$[1.59C, 3.76C]$
		worst case	no	12	$[1.50C, 3.34C]$
			yes	8	$[1.93C, 5.81C]$
	dynamic selection	average case	no	14	$[1.36C, 2.71C]$
			yes	11	$[1.59C, 3.76C]$
		worst case	no	13	$[1.43C, 3.00C]$
			yes	13	$[1.43C, 3.00C]$
fitness-proportional	random selection	average case	no	9	$[1.79C, 4.94C]$
			yes	19	$[1.13C, 1.77C]$
		worst case	no	12	$[1.50C, 3.34C]$
			yes	17	$[1.21C, 2.07C]$
	dynamic selection	average case	no	13	$[1.42C, 3.00C]$
			yes	22	$[1.05C, 1.43C]$
		worst case	no	9	$[1.79C, 4.94C]$
			yes	20	$[1.10C, 1.65C]$

Table 7.3: Evolutionary tests on chromosome selection schemes, sample selection methods, robustness and elitism

random selection of samples and the other is dynamic (fitness-proportional) selection. In noisy environments, a controller should be repeatedly tested and two different measures, the average case and the worst case measure, can be considered. Table 7.3 shows the possible combinations over the evolutionary measures and their performance.

The control structures in the experiments were finite state machines with two internal states; wall following behavior requires two internal states with binary sensors, from the experiments in the previous chapter (see section 6.2.2). The performance evaluation is based on effort costs using 25 runs on each configuration. It will estimate how much effort should be spent to achieve desirable solutions. The baseline of performance is fitness 60 (equation 7.3) which corresponds to visiting more than 40 energy tanks without collision, on the average at every starting position. Two sensors at angles $45°, 135°$ were used with random noise $\pm 10\%$. Crossover rate 0.6 and mutation rate 0.02 were used with 200 generations in the experiments.

From Table 7.3, it can be seen that the fitness-proportional method is better than the tournament-based method in reproduction mechanism for wall following behavior, when elitism is applied. The elitism of the chromosome with the best fitness for N positions in the N-K sample method is very effective in the fitness proportional method. The evaluations of multiple runs with the average case and the worst case performance in equations 7.1-7.2 are not significantly different. Both measures are useful to remove noise effects in evolving controllers in noisy environments.

The random selection and dynamic selection methods are not significantly different in their performance, although dynamic selection method is slightly better than random selection in tournament-based reproduction. Also in fitness-proportional reproduction the dynamic selec-

tion method shows a little better performance than random selection, except the worst-case fitness experiments of multiple evaluations without elitism. In terms of the effort cost, one cannot assert that dynamic selection is superior to random selection. However, it shows its potential for better results.

It seems that accelerating the performance with elitism is more influential in performance than screening out bad chromosomes sensitive to noise. Also dynamic selection with the N-K sample method can be recommended when elitism is applied.

7.4 Summary

The robotic experiments in this chapter are distinguished from grid world problems. They allow noisy sensor readings and dynamic robotic movement by motor actions. Most importantly noisy sensor readings create the following issues:

- How to model sensors
- How to choose samples in the N-K sample method
- How to obtain robustness and elitism in the N-K sample method

A realistic sensor reading is obtained with sampling sensor values in a real Khepera robot. Infrared sensors have 40 degrees bearing cones; 30 points in the cone area are examined to determine whether each point touches any object. Then random noises are added to sensor readings and motor actions as observed in real robots. Such realistic robot models imitate robots working in real environments. This method requires much computational time as the coordinates of sensor points for infrared sensors need to be calculated.

Some researchers used an efficient tabular method for all sensor readings and motor actions [Miglino et al., 1995; Lee, 1998]. If distance and angle between a robot and an object are given, they are looked up in the sensor table to find the appropriate values. It can reduce the computation time of the evolutionary process. Instead the method requires careful efforts to measure a sensor value corresponding to each distance and angle for an object. For future work, we need to find an efficient method with less effort to obtain sensor samples in real robots. Thus, the following method can be proposed. After taking sensor points and their values for each infrared sensor, more fine-grained values can be collected by testing them at many possible distances and angles and these can then be saved into sensor tables. If the sensor

tables are employed in evolutionary computation, it will reduce the computing cost and closely match real sensor values.

The *N-K* sample method was examined and applied in this chapter. This method was applied for the Tartarus problem in chapter 5 and for robotic tasks in chapter 6. It was based on random subset selection method [Gathercole, 1998]. Normally *K* samples can be randomly selected from *N* positions for every generation. In some cases, when difficult positions (hard to evolve) receive more attention and time, it can accelerate the convergence to solutions. Thus, a variation of dynamic sample selection [Gathercole and Ross, 1994] based on fitness-apportionate probability is suggested instead of uniform probability on sample selection; the higher penalty fitness, the more chance to be selected as *K* samples. This method can easily see what positions are hard to evolve by checking how many times each sample is selected during evolutionary runs.

Realistic simulation with noise needs a variation of the *N-K* sample method. Noise in sensor readings may produce different behavior results on the same genome controller. A single fitness test does not guarantee to have robust controllers. Thus, multiple evaluations are applied on a chromosome and then the average score or the worst score is considered to represent its fitness. This will have the effect of removing noise-sensitive controllers. If multiple evaluations are applied to every chromosome in a population, it will increase the evolutionary time in proportion to the number of evaluations. When the *N-K* sample method is used for many starting positions, the best chromosome in a population will receive multiple evaluations on *N* positions. The best scoring chromosomes that also have stable fitness curves will be selected as desirable controllers.

Various methods of reproduction, sample selection and multiple evaluation schemes were tested for wall following behavior. Tournament-based and fitness-proportional reproduction methods were tested for reproduction, random selection and dynamic selection for the *N-K* sample method, and the average and the worst fitness for multiple robust evaluations. Elitism was also tested to see its significant effect. For wall following behavior, fitness-proportional method was better than tournament-based method. The average case and the worst case performance were not significantly different. Dynamic selection was a little better than random selection, though they were not significantly different. Elitism was a most influential factor to improve performance even in noisy environments. The chromosome with the best multiple evaluation score over *N* positions was selected as the elite and it was then inserted into a new generation. This accelerated convergence speed to the design of optimal robust controllers. Increasing sample size to a large extent had a tendency of increasing the computing cost with

a small improvement over the performance.

The methodology suggested in this chapter will be applied to several robotic tasks in the next chapter. The average score of multiple evaluations and dynamic selection for the N-K sample method will be used with $(N = 30, K = 5)$. The method is quite useful to design robust controllers to cover many sample positions in noisy environments. Various robotic behaviors will be demonstrated with memory analysis.

Chapter 8

Robotic Tasks in Noisy Environments

In this chapter, robotic tasks are investigated with realistic simulation involving noisy signals in sensor readings. The effect of memory is examined in a variety of tasks. Robotic kinematics and sensor readings simulate real robotic tasks closely. The simulation results will indicate what real robot performance looks like.

Some robotic tasks tested in chapter 6 – wall following, corridor following and box pushing – will be studied again to see the difference between whisker-like sensors and realistic sensors as well as to cover how to design robust controllers and how to find desirable controllers with high performance in test environments. Also, T-maze decision behavior is explored as an example of a difficult case for purely reactive controllers.

The evolutionary multiobjective optimization (EMO) method will be applied to see significance statistics results on variable state machines in noisy environments, as well as fitness tests and effort tests. Experiments will test various control structures including feedforward neural networks and rule-based state machines from two sensors to six sensors in addition to finite state machines. The controllers are evolved to see memory effects depending on the number of sensors or sensor range.

To find successful controllers, several test environments are introduced for each task and evolved controllers will be evaluated for the test environments. Two criteria, a collision factor and a fitness factor, are considered for choosing the best controllers in the experiments. The process of selecting desirable controllers will be demonstrated with their behaviors.

8.1 Methods

The experiments on a variety of tasks will use realistic Khepera models where robot kinematics (see section 6.1) are simulated and sensor readings are sampled in a real Khepera robot. Infrared sensors have 40 degrees bearing cones where sensor values are estimated with 30 points distributed in the cone area (see section 7.1). The method used in the simulation program is not so efficient as a tabular method [Miglino et al., 1995; Lee, 1998] to record sensor readings and motor actions in a real robot. However, it requires less efforts to measure sensor values at possible distances and angles to every object. In experiments, random noises are added to the sensor readings and motor actions. Sensor readings ranges from 0 to 1023 and will be binarized with threshold 500 for state machines. To see the effect of sensor range, state machine controllers will be compared with neural network controllers evolved over real-value sensor range.

There are ambient light sensors at the same angle positions as infrared sensors. The light sensors can detect light intensity in a given environment. In the simulation experiments, two light sensors at angle 22° and 158° are used. The light intensity is estimated with two factors, the distance of the light sensor from a lamp, and the lamp's angular deviation from the centreline of the sensor.

Wall following, corridor following, box pushing and T-maze behavior will be investigated to see memory effects on the behavior performance. For wall following, corridor following and box pushing, N-K sampling will be applied with $N = 30, K = 5$ since they have relatively many sample positions. Test environments have 100 sample positions, respectively. Evolved controllers will be evaluated for each environment 10 times and its average score will be recorded. The average of multiple evaluations with dynamic selection method and N-K sampling are used in evolutionary computation.

For T-maze decision behavior, multiple evaluations will be applied to every chromosome in a population, while for other behaviors multiple evaluations are used for only the best chromosome in a population. For each task, memory-based controllers and purely reactive controllers are compared. The evolutionary multiobjective optimization (EMO) method is applied to quantify memory amount for significant differences in performance. Also finite state machines, rule-based state machines and neural networks are applied to the tasks. Evolving feedforward neural networks can produce the best solution for purely reactive controllers with a continuous sensor range.

8.2 Wall Following Behavior

8.2.1 Experimental Issues

Wall following behavior with realistic simulation was tested in the previous chapter to see N-K sampling and sample selection issues in noisy environments. The same fitness function in equation 7.3 is used with 1200 time steps for a single run. The environment is shown in Figure 7.6 with 30 random starting positions. Thus, N-K sampling with $N = 30, K = 5$ is used.

For wall following behavior, the following issues will be addressed:

- Are memory-based controllers better than purely reactive controllers for wall following behavior? For significance statistics, are the fitness test and effort test useful to determine how many memory states are required when realistic noisy simulation is applied?
- Is the EMO approach better than the fitness test for memory analysis?
- Feedforward neural networks are one of the purely reactive controllers. Is the neural network control structure more powerful than state machines with binarized sensor readings?
- How can we choose desirable controllers with several test environments?

8.2.2 Purely Reactive Controllers vs. Memory-based Controllers

In the previous experiments with whisker-like binary sensors (chapter 6), two internal states were required to achieve wall following behavior. Now we investigate if more realistic sensor modeling will influence the memory requirement. Evolutionary computation will find the best control structure even in the noisy world environments. Tournament selection of group size four is used for the test. The two best chromosomes in a group will be copied to a new population and they will also be reproduced with genetic operators. Mutation rate is set to 2 over chromosome length, crossover rate is 0.6 and population size is 50. To see the memory effect, finite state machines are applied to control structures. 25 runs are tested to validate the performance and each single run uses 200 generations.

Only two infrared sensors at 45° and 135° in front are first used and the threshold for binarizing sensor values is 500. Two different random noises over sensor readings and motor actions were used to see the effect of noise: $\pm 5\%$ and $\pm 10\%$ noise. The results of random noise $\pm 5\%$ and $\pm 10\%$ were not significantly different in performance and $\pm 10\%$ noise had a slightly

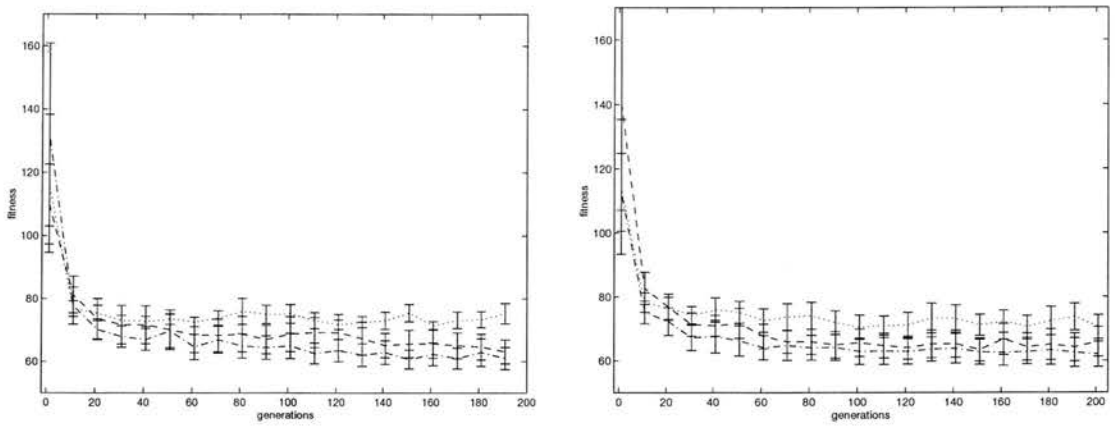


Figure 8.1: Fitness curves with 25 runs for wall following behavior (a) $\pm 5\%$ noise (b) $\pm 10\%$ noise (dotted: reactive, dashdot: 2 states, dashed: 3 states)

larger standard deviation (error bars) than $\pm 5\%$ as shown in Figure 8.1. Considering the t-distribution over fitness data from 25 runs, memory-based controllers were significantly better than purely reactive controllers in the case of random noise $\pm 5\%$ with 95% confidence intervals as shown in Figure 8.1(a). There was no remarkable difference between two and three states. Two state machines were a little better than three state machines in the average performance, but the difference was not significant. It is partly because three state machines have a longer size in their chromosome representation and the mutation rate is dependent on the chromosome length. The realistic model experiments have similar results to simple model experiments in Figure 6.4.

When $\pm 10\%$ noise is added to sensor readings and motor actions, the performance of purely reactive controllers is close to that of memory-based controllers, as shown in Figure 8.1(b). Thus, the effort test was applied to the experimental results. When we assume that desirable controllers have the penalty fitness 60 or below (equation 7.3), purely reactive controllers had no such performance at any run, while two or three state machines had 12 successful controllers among 25 runs. The best performance among 25 runs with purely reactive systems was 68.5 on the average for 30 positions, which means that 31.5 energy tanks were visited among the 46 tanks in total. The best performance with memory-based controllers was 54 which corresponds to visiting all energy tanks in the arena. We can say that the performance is significantly different in terms of the effort cost; the effort cost interval (95% confidence) for 12 success is $[1.50C, 3.34C]$ while that for no success is $[7.57C, 1030.93C]$ where C is a computing cost for a single run. Using the effort test, we can see a clear difference between purely reactive

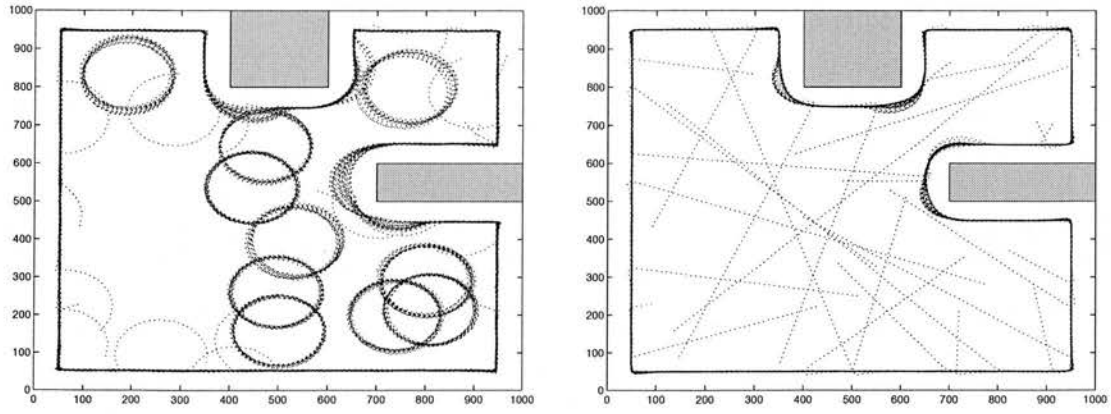
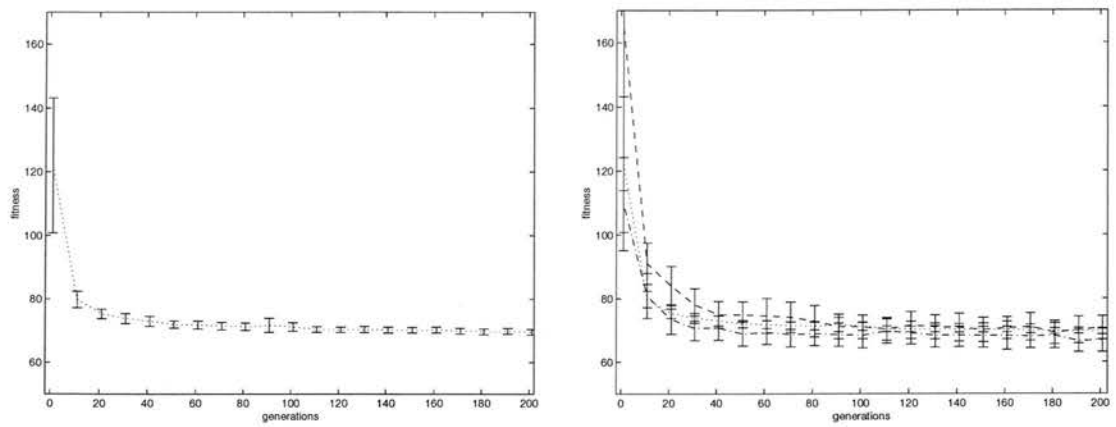


Figure 8.2: Comparison of the best performance between memoryless and memory-based systems for wall following behavior (a) purely reactive system (b) 2 state machines

controllers and memory-based controllers. The results support the conjecture that two states are sufficient for wall following behavior regardless of sensor modeling.

Figure 8.2 shows wall following behaviors by the best chromosomes for purely reactive controllers and two state controllers. In Figure 8.2(a), the memoryless reactive system has circular movements to detect any wall and begins to follow walls when any sensation is recognized. On 10 starting positions among 30 positions, the robot shows circular movements and the performance becomes degraded since the robot cannot scan other areas. Once any wall is detected, the controller shows good performance to follow walls and visit all energy tanks. Wall following behavior with two state machines is demonstrated in Figure 8.2(b). The behavior is decomposed into two simple behaviors: one is to go straight to reach any wall when the robot sees no obstacle and the other is to follow walls by repeating two actions of turning right and moving forward. There is a conflict of perceptions in the two primitive behaviors. The sensation of no obstacle in front should be distinguished between the two behaviors and internal states can solve the conflict by memorizing each distinct situation, but purely reactive controllers have a limitation on the problem because they cannot separate the perceptual aliases.

The experiments have used two sensors so far and two more infrared sensors at 78° and 102° were added to see the effect of more sensors. Four sensors can reduce fluctuation of fitness curves and prevent collision with obstacles by perceiving better the surrounding environment, as shown in Figure 8.3. The fitness curves of 25 run experiments for purely reactive controllers with 4 sensors have smaller standard deviations over the average performance than those with 2 sensors, when $\pm 10\%$ noise are used. However, adding sensors does not improve fitness



(a) (b)

Figure 8.3: Fitness curves with four sensors (a) purely reactive control (b) comparison over different internal states (dotted: reactive, dashdot: 2 states, dashed: 3 states)

performance. Purely reactive control systems still show circular behavior in the arena if robots are placed at positions far from walls. Their trace results are almost the same as the results shown in Figure 8.2(a); at 10 places among 30 positions, the robot has circular behavior and cannot follow walls.

Figure 8.3(a) shows the fitness curve of purely reactive control with 4 sensors. Compared with 2 sensors, it shows very small-sized error bars. The surrounding environment can be perceived better and collisions can therefore be prevented. Collisions often occur because of wrong perceptions from noisy sensor signals. Increasing the number of sensors will reduce the probability of misleading perceptions and thus decrease the collision rate. Figure 8.3(b) shows that purely reactive systems and memory-based systems have no significant difference over fitness distribution. Considering the effort test, memory-based systems have 4 successes (fitness is below 60) while purely reactive systems have no success. Their difference is not statistically significant.

Memory-based systems with four sensors have a larger size of chromosomes and worse performance than those with two sensors, and thus it is more difficult to evolve optimal control structures at the same computing cost. It will require more generations to see the gap between purely reactive systems and memory-based systems. The significance test can be improved with the following memory analysis on variable state machines.

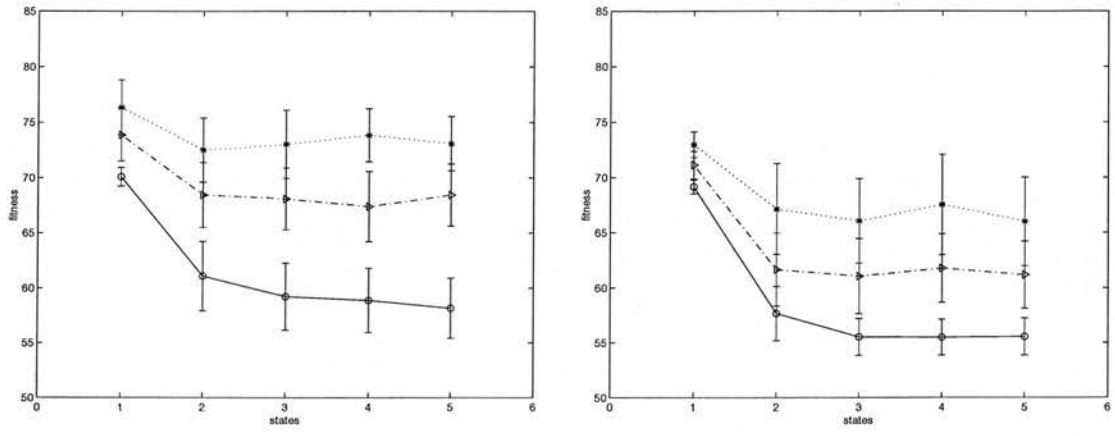


Figure 8.4: Memory analysis over wall following behavior with 2 sensors (a) population size 50 (b) population size 100 (dotted: 5 generations, dashdot: 100 generations, solid: 300 generations)

8.2.3 Memory Analysis with the EMO Approach

The evolutionary multiobjective optimization (EMO) approach is applied to variable state machines as control architecture in order to analyze the best performance over quantified memory states. Two different population sizes were tested to see the effect of memory for wall following behavior with 2 sensors, as shown in Figure 8.4. For significance statistics, 25 runs are tested and their error bars are displayed together at the same generations. When the number of generations is increased, the performance becomes close to optimal, visiting all energy tanks (fitness 54). There is a significant difference between memoryless (one state) approach and memory-based (more than one state) approach. However, two states are not significantly different from more than two states, though more than two states are a little better in the average performance. From the figures, one can just say that more than two states are easily leading to a little faster convergence. When 1000 generations were tested, the performance of two states became close to that of other multi-states with a small gap between them. In Figure 8.4(a)-(b), larger populations show better performance, as expected. Increasing the population size in evolutionary computation requires the effect of more computing effort to reach solutions.

Using the fitness test over various states with four sensors, it was hard to see the difference between memoryless controllers and memory-based controllers as shown in Figure 8.3. The EMO approach was also applied over variable state machines with four sensors. The result in Figure 8.5 shows that memory-based controllers are significantly better than purely reactive controllers and two or more states are not statistically different. From the above results, it is

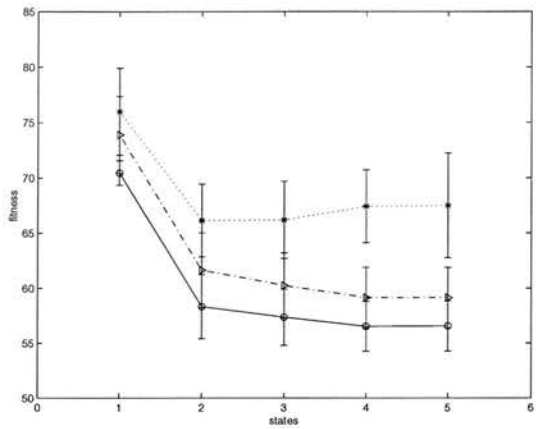


Figure 8.5: Memory analysis over wall following behavior with 4 sensors, population size 100 (dotted: 5 generations, dashdot: 100 generations, solid: 300 generations)

inferred that the wall following behavior requires two states, or 1-bit memory regardless of the number of sensors. The EMO approach proved it was very effective for taking the significance test on fitness distribution as well as for obtaining solutions for each number of states. Also it can generate solutions of strictly increasing performance for the number of states, unlike independent runs for each number of states.

8.2.4 Neural Network Approach

The next experiments for memory analysis involve the evolution of neural networks. Feed-forward neural networks can be one of the best control architecture in purely reactive control systems, since they can in principle represent any mapping from sensor readings to motor actions. For neural network controllers, the motor actions will have integer values ranging from -8 to 8, while previous experiments have eight possible motor actions. The neural network outputs are discretized into integer values from real continuous values. Neural networks used six infrared sensors in the front of the robot, each of which produces a continuous integer value ranging from 0 to 1023.

In neural networks weight parameters are encoded into integer values from -128 to 127 with 8 register bit representation. One hidden layer and four nodes in the hidden layer are used for control structures. Then the chromosome representation will be a series of weight parameters from input layer to hidden layer and from hidden layer to output layer. When genetic algorithms were applied to neural networks, successful neural network controllers with four hidden nodes were found with 200 generations and population size 50. Crossover rate was 0.6 and mutation

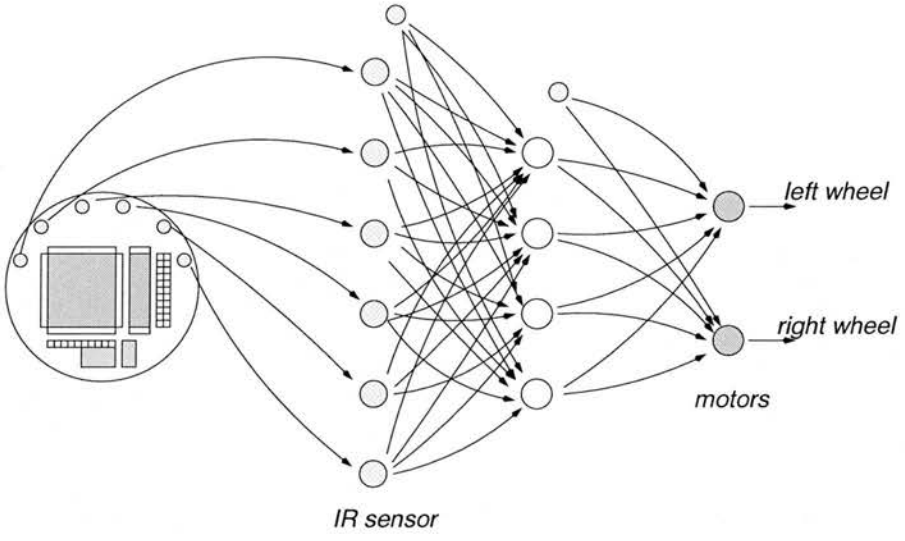


Figure 8.6: Neural network diagram for purely reactive controllers

rate was 2 over chromosome length. Figure 8.6 shows the neural network structure as the controller.

Figure 8.7 shows the successful wall following behaviors of some of the best chromosomes. The result in Figure 8.7(a) is quite similar to the behavior with two state machines in Figure 8.2(b). The robot first goes straight until it reaches any wall or obstacle and then follows walls by changing motor actions depending on continuous-valued sensor readings. Figure 8.7(b) displays a different style of neural network controller which has a circular movement to detect walls. The circular radius is larger than that in Figure 8.2(a). Neural network controllers have a finer scale of motor actions which can produce the circular movement with a larger radius; its motor actions are 7 and 8 for left and right wheels, respectively. The radius of circular movement depends on the difference between motor actions for left and right wheels. Purely reactive controllers with binarized sensors, shown in Figure 8.2(a), had motor actions 6 and 8 for left and right wheels, respectively. If real values are allowed for motor actions, evolutionary computation will be able to evolve neural networks into circular movements with much larger radii.

Figure 8.7 shows neural network controller with six sensors. A different number of sensors was tested to see if the number of sensors influences performance with neural networks. Figure 8.8 is the fitness distribution with two sensors $(45°, 135°)$, four sensors $(45°, 78°, 102°, 135°)$ and six sensors $(22°, 45°, 78°, 102°, 135°, 158°)$. Their performance is not significantly different, which indirectly implies that two sensors are sufficient to evolve wall following behavior for a given environment. Neural networks have continuous ranges on sensors and motor actions.

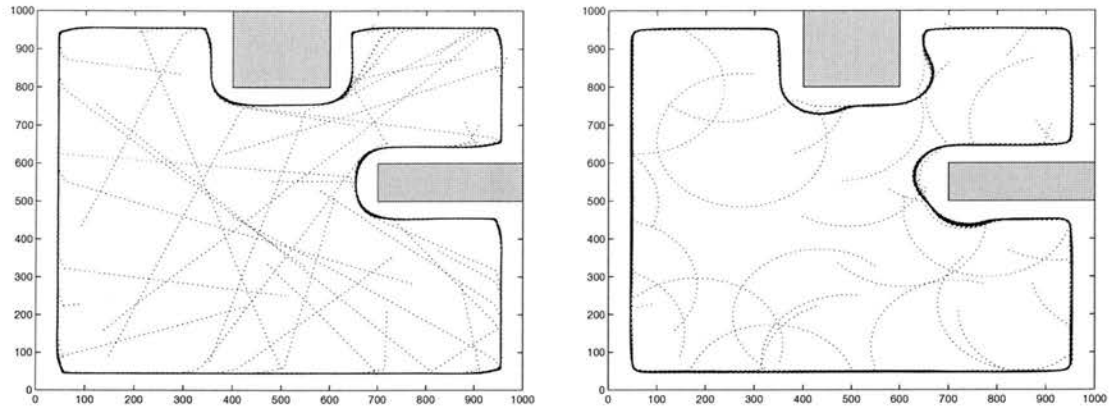


Figure 8.7: Examples of neural network controllers for wall following behavior

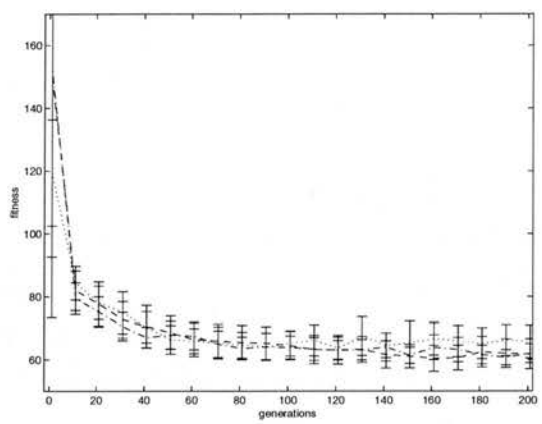


Figure 8.8: Fitness curve of neural network controller for wall following behavior (dotted: 2 sensors, dashdot: 4 sensors, dashed: 6 sensors)

From the experiments on neural network controllers, it is inferred that the performance of purely reactive systems can be influenced by their sensor range and variety of motor actions.

8.2.5 Testing Controllers in Various Environments

Robot controllers were evolved over 30 random positions and directions in one environment. To prove the usefulness and robustness of evolved controllers, the controllers were tested in several environments. The purpose of testing controllers is to see what kind of controllers are robust and applicable to various environments. The testing will check whether memory structure and sensor configurations are important to adapt controllers to environments. Also,

it will show whether controllers adapted to one environment can be easily applied to other different environments.

Various control structures are investigated in the experiments. To see the relationship between memory and sensor configurations, different numbers of sensors (two, four and six) were tested with variable states. Also, feedforward neural networks were evolved to be compared with finite state machines. The neural networks had a fixed number of hidden nodes (four), and a different number of sensors were tested. The neural network structure is used to investigate the limitations of purely reactive systems.

The testing environments consist of four environments, `env0, env1, env2, env3`. The first environment (`env0`) is the same as the evolved environment in Figure 7.6 and has 30 starting positions. The second environment (`env1`) has the same arena with the same objects but 100 new random positions are generated as shown in Figure 8.9(a). This environment is used to test how well the evolved controller will work when robots start at different positions and directions. The third environment (`env2`) consists of various objects including sharp corners, as shown in Figure 8.10(a). The environment has 100 random starting positions. It is an environment with many difficult positions and it is easy to collide with objects. For successful behaviors, robots should move carefully while following walls. The fourth environment (`env3`) includes cylinder-shaped objects instead of polygon-shaped boxes, as shown in Figure 8.11(a) and 100 positions and directions are randomly selected for starting positions.

The chromosomes with the best performance are first collected by evolving robots in the environment `env0` shown in Figure 7.6. Each evolved controller is evaluated 10 times for each of the four environments and its performance is recorded in Tables E.1-E.3 (see appendix E). They show the average performance and standard deviation over 10 evaluations. Data from the two best controllers with desirable performance in four environments are collected into Table 8.1.

Table 8.1 shows the results of two components of performance measurement. One is to check how many collisions occur for each environment by testing controllers at all starting positions. The other is to see the fitness which is related with how many energy tanks are visited. The fitness is averaged over starting positions that experience no collision. The optimal fitness (penalty fitness) is 54 for `env0` and `env1`, 37 for `env2`, and 76 for `env3`. The two component measures, collision and fitness, are estimated with 10 runs to see the distribution. Desirable controllers should have the least collisions and the best fitness. In the experiments, there was no perfect controller; no collision and the optimal fitness. When a controller is evolved for the

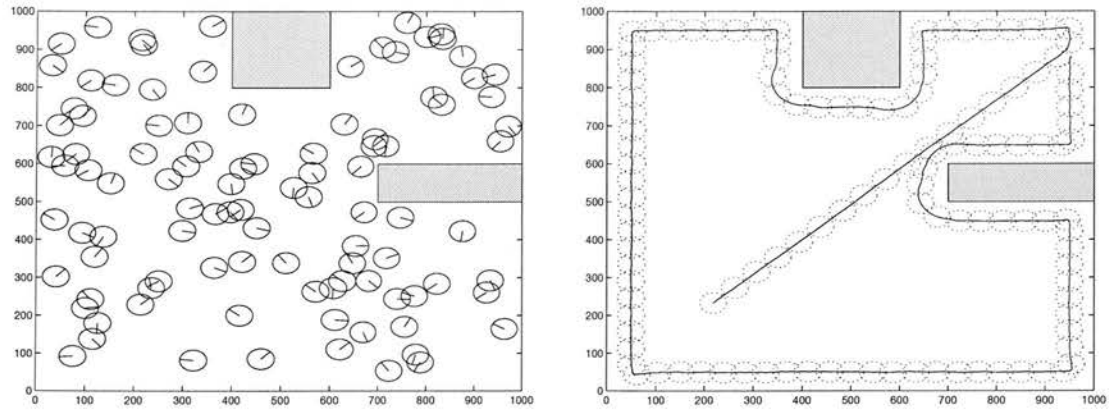


Figure 8.9: Test environment `env1` for wall following (a) 100 random positions and directions (b) an example of testing the best two-state machine at one position

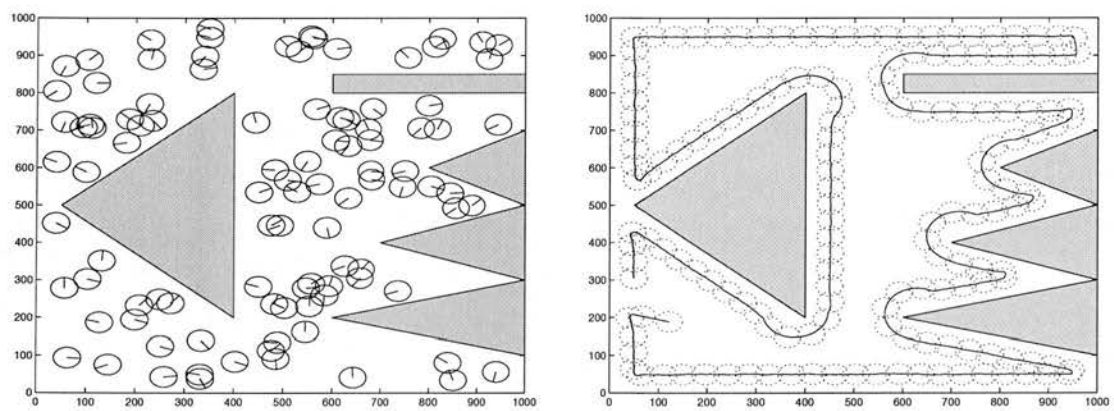


Figure 8.10: Test environment `env2` for wall following (a) 100 random positions and directions (b) an example of testing the best two-state machine at one position

environment `env0`, the controller normally has a good performance in the environment `env1` but it shows unstable performance in the environments `env2` and `env3`. The evolved controller can be adapted to the given environment but it is not guaranteed to have a desirable performance to unknown environments.

Regardless of control structures or the number of sensors, the best controller evolved in `env0` was normally successful in `env1` at nearly all 100 random positions, exceptions at one or two positions. The difficult positions were so near a wall, and the robot collided with the wall within the first few time steps. However, if the robot is placed a little distance further from the wall to look around the environment, it can follow walls without collision. When the best controller is applied to `env2`, it has a higher probability of colliding with obstacles, since the

control structure		env0	env1	env2	env3
2 sensors, 1 state	c_5*	0.1±0.3	0.7±0.7	3.3±0.6	2.6±1.6
	c_6*	0.0±0.0	0.6±0.7	2.9±0.9	0.1±0.3
2 sensors, 2 states	c_3*	0.0±0.0	1.2±0.8	4.6±0.7	0.5±0.7
	c_9*	0.0±0.0	2.0±0.0	3.8±0.7	0.0±0.0
2 sensors, 3 states	c_0*	0.0±0.0	0.0±0.0	4.4±0.5	0.0±0.0
	c_5*	0.0±0.0	0.0±0.0	4.1±0.9	0.0±0.0
2 sensors, 4 states	c_1*	0.0±0.0	0.0±0.0	4.7±0.9	0.1±0.3
	c_8*	0.0±0.0	0.5±0.5	4.0±0.0	0.6±0.7
4 sensors, 1 state	c_0*	0.0±0.0	0.0±0.0	0.0±0.0	0.3±0.5
	c_3*	0.0±0.0	0.5±0.5	0.0±0.0	1.7±0.9
4 sensors, 2 states	c_0*	0.0±0.0	0.6±0.5	1.4±0.5	0.6±0.8
	c_5*	0.0±0.0	1.0±0.0	0.0±0.0	1.0±0.0
4 sensors, 3 states	c_2*	0.0±0.0	0.6±0.8	0.8±0.9	0.3±0.5
	c_7*	0.0±0.0	4.0±0.0	2.1±0.9	1.6±0.8
4 sensors, 4 states	c_2*	0.0±0.0	0.5±0.5	0.7±0.9	0.7±0.6
	c_5*	0.0±0.0	0.5±0.7	3.0±1.2	0.5±0.7
2 sensors, neural	c_0*	0.0±0.0	1.9±0.3	34.2±2.8	7.0±0.0
	c_9*	0.0±0.0	1.3±0.6	54.8±3.8	0.1±0.3
4 sensors, neural	c_2*	0.0±0.0	0.0±0.0	2.4±1.5	0.8±0.4
	c_5*	0.0±0.0	1.0±0.0	6.6±1.0	4.9±0.3
6 sensors, neural	c_1*	0.0±0.0	0.0±0.0	2.4±0.7	0.1±0.3
	c_5*	0.0±0.0	0.0±0.0	1.0±0.0	0.1±0.3

(a) collision test

control structure		env0	env1	env2	env3
2 sensors, 1 state	c_5*	68.18±0.72	69.92±0.33	40.00±0.19	82.98±0.53
	c_6*	68.21±0.62	69.83±0.29	37.07±0.13	82.03±0.30
2 sensors, 2 states	c_3*	54.00±0.00	54.00±0.00	46.76±0.19	77.98±0.11
	c_9*	54.00±0.00	54.00±0.00	38.96±0.06	76.84±0.24
2 sensors, 3 states	c_0*	54.00±0.00	54.00±0.00	44.58±0.10	78.91±0.04
	c_5*	54.00±0.00	54.00±0.00	39.27±0.12	76.89±0.33
2 sensors, 4 states	c_1*	54.00±0.00	54.00±0.00	39.82±0.22	76.80±0.19
	c_8*	54.00±0.00	54.00±0.00	41.83±0.08	78.04±0.07
4 sensors, 1 state	c_0*	67.71±0.94	69.67±0.37	38.43±0.14	81.61±0.41
	c_3*	68.27±0.91	69.90±0.32	45.74±0.40	83.67±0.36
4 sensors, 2 states	c_0*	54.36±0.03	54.37±0.02	49.27±0.04	78.92±0.29
	c_5*	54.00±0.00	54.00±0.00	38.29±0.30	76.10±0.31
4 sensors, 3 states	c_2*	54.00±0.00	59.19±0.59	49.46±0.10	81.67±0.28
	c_7*	54.00±0.09	54.00±0.00	37.45±0.24	77.88±0.42
4 sensors, 4 states	c_2*	54.00±0.01	59.15±0.27	49.62±0.12	81.81±0.21
	c_5*	54.01±0.02	54.16±0.20	48.01±0.38	79.11±0.11
2 sensors, neural	c_0*	56.46±0.59	60.25±0.64	50.84±0.32	77.92±0.44
	c_9*	55.66±1.15	59.70±0.67	49.62±0.65	77.78±0.21
4 sensors, neural	c_2*	55.57±0.94	59.54±0.67	49.57±0.43	77.05±0.55
	c_5*	55.71±1.13	59.20±0.67	56.81±0.94	77.84±0.31
6 sensors, neural	c_1*	54.00±0.00	57.66±0.24	41.18±0.09	78.91±0.09
	c_5*	54.61±0.60	58.86±0.63	46.89±0.12	77.19±0.60

(b) fitness performance

Table 8.1: Comparison of performance with various control structures for wall following (each data shows the average fitness and standard deviation over 10 evaluations)

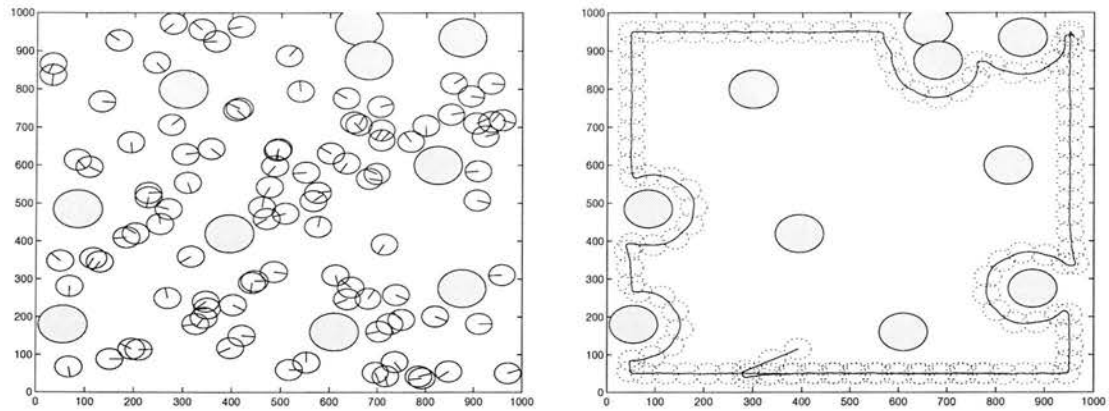


Figure 8.11: Test environment `env3` for wall following (a) 100 random positions and directions (b) an example of testing the best two-state machine at one position

environment has many sharp corners and it is difficult to perceive the corners with a small number of sensors.

Neural networks with two sensors (at $45°, 135°$) have bad performance with many collisions in `env2`; they have more than 30% failures. The evolved neural network controllers have circular movements with large radii and the behavior improves fitness in `env0` by visiting almost all energy tanks around the wall. However, it makes robots liable to collide with obstacles in `env2`. Moreover, two sensors are insufficient for looking around the surrounding environment. In contrast, finite state machines with two sensors are more careful at following walls, making smaller circular movements, since a restricted set of motor actions produces a small-radius circling.

The evolved neural networks with six sensors have two kinds of controllers as shown in Figure 8.7. Two neural network controllers (c_0, c_1 in Table E.3 – see appendix E) were tested in different environments. The controller with circular movements (c_1) had almost no collision in the test environment `env3`, while the go-straight controller (c_0) experienced collisions at a rate of roughly 20%. The go-straight controller was not successful at wall following from many positions and thus it had worse fitness. It has a very weak strategy at concave corners and it does not know how to escape difficult situations. It seems that the go-straight controller needs internal states to handle each difficult situation or complex sensorimotor mapping, and more hidden nodes may be required.

Noisy sensors and motor actions often influence the perceptual states on robots with a few sensors. Increasing the number of sensors gives more potential to obtain robust controllers.

Adding sensors can help robots recognize perceptual situations better even with noisy sensor readings. It will prevent misunderstanding environmental states due to noise and can solve perceptual aliases partly. The collision rate for neural network controllers can be reduced by increasing the number of sensors from two to six as shown in Table 8.1(a). This also happens for state machine controllers; four sensors have smaller collision rates than two sensors (see test environment `env2`).

Internal states play an important role in achieving desirable performance. Two state machines with two sensors or four sensors are better in fitness performance than even neural networks (see Table 8.1(b)). Purely reactive systems with continuous sensor range and expanded motor actions still have restrictions in wall following behavior, even though they improve the performance. Thus, one can say that wall following behavior requires at least two internal states.

From Table 8.1, the best control architecture is the controller c_5 with 4 sensors and 2 internal states. It has almost no collision and uniformly good fitness performance in every test environment. Figure 8.9(b), 8.10(b), and 8.11(b) show its behavior. Without the continuous sensor range used in neural networks, the behaviours demonstrate that only binarized sensor information is sufficient to process perceptions appropriately. In the experiments, two criteria, collision and fitness, were used to select desirable controllers. The criteria for selection of desirable controllers will be more complex if characteristics of behaviors are added such as how safely robots move and how properly robots escape difficult situations. These are therefore multi-criterion selection problems.

According to the results in Tables E.1-E.3 (see appendix E), evolved controllers are so diverse that they often produce different performance in the same environment. Evolving robot controllers requires much environmental experience in order to obtain robust controllers working in diverse situations. Generally evolved controllers in an environment work correctly in the given environment. In evolutionary computation, sampling should be carefully used to produce the ability of handling difficult situations and to prevent unpredictable environmental states.

8.3 Corridor Following

Corridor following is similar to wall following behavior. The task is to follow narrow corridors from one end to the other end. Figure 8.12(a) shows the environment and 30 random positions selected in the arena. The environment has narrow corridors in the middle of the path and robots need to pass through corridors appropriately to enter a small room at the end of corridor.

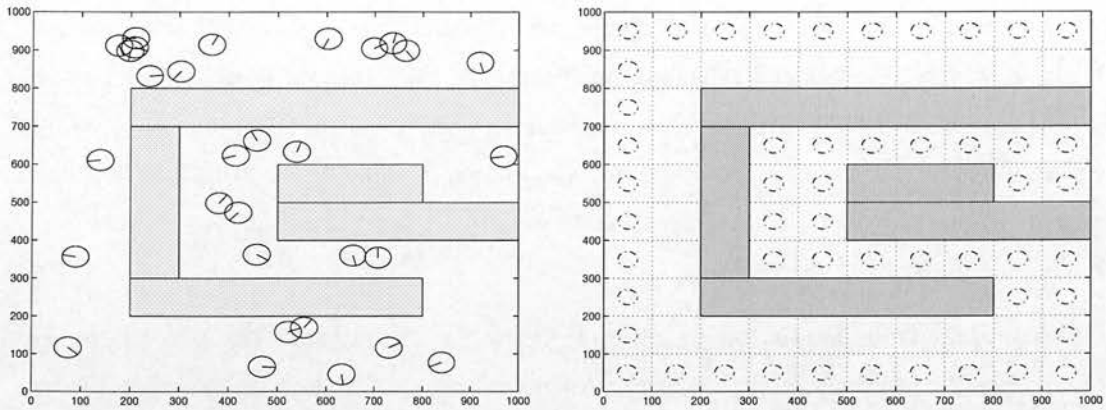
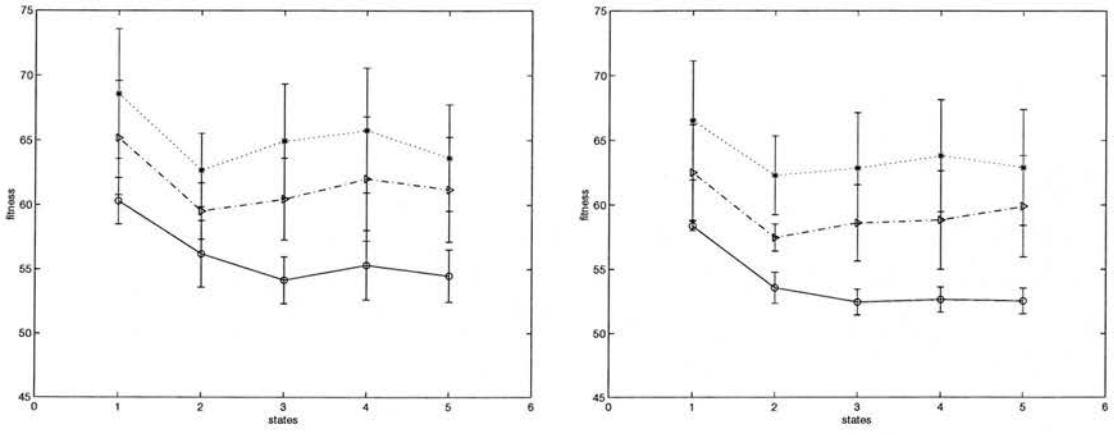


Figure 8.12: World environment for corridor following (a) 30 random positions and directions (b) energy tanks in the environment

Narrow corridors are not open fields, so robots tend to hesitate to go through them since there is a higher probability of collision. Random sample selection with the *N-K* sample method had difficulty in obtaining desirable controllers for narrow corridor following, as mentioned in section 5.2.5. Thus, the dynamic selection method is applied with five samples among 30 positions, instead of random selection. With realistic simulation, a random noise $\pm 10\%$ is employed for sensor readings and motor actions. Similar to wall following behavior, the same style of fitness calculation is used. For exploration time, 1200 time steps are assigned for each genome controller. The energy tanks are distributed around walls as shown in Figure 8.12(b), which will lead robots to follow corridors.

Evolutionary computation for corridor following uses the same fitness function given in equation 7.3. The experiments first investigate memory analysis for how many states are required for corridor following. The basic control structure is a finite state machine tested in wall following behavior. Memory and its corresponding performance will be analyzed with the number of sensors. Neural networks are also compared with single-threshold state machines. The sensor range and the number of sensors are relevant factors related to perceptual aliases. It is believed that the aliases influence the need of memory. Various control structures and their performance will elucidate the relationship between sensors and memory.

The best chromosomes with stable and robust performance are selected and applied to several corridor environments. The testing environments consist of three mazes with 100 random robot positions and directions as well as the environment used for evolving. The fitness and the number of collisions are considered as criteria to choose desirable controllers. Two performance



(a) (b)

Figure 8.13: Memory analysis with 2 sensors for corridor following (a) 25 runs without removing bad runs (b) 25 runs with removing bad runs (dotted: 60 generations, dashdot: 100 generations, solid: 300 generations)

measures will be useful to select safe and robust controllers.

8.3.1 Purely Reactive Controllers vs. Memory-based Controllers

In this environment, robots are always close to corridors and so they will not need the effort of exploration to reach a corridor; for wall following, robots need to go straight or make a circular movement to approach a wall. We still have the question of whether or not corridor following requires internal memory. The EMO approach was quite effective to investigate the memory hierarchy and fitness performance, instead of running each state machine controller independently. Thus, the EMO approach was first applied to two-sensor robots. For significance tests, 25 experiments were run with 300 generations and a population size of 100. The average fitness was estimated with 95% confidence intervals as shown in Figure 8.13. Purely reactive systems and memory-based systems with more than two states are significantly different as in Figure 8.13(a). The EMO that runs on two-sensor robots had a wide performance distribution, and in some runs the fitness was stuck with a high penalty value even after 300 generations. Those runs were removed and new experimental runs with better performance were inserted into the pool of 25 run data. Figure 8.13(b) shows the new diagram of memory analysis. In this figure, memory-based systems with two states or more are significantly better than purely reactive systems.

In many cases, two sensors are sufficient to follow corridors within 1200 time steps. If more than 1200 time steps is given for exploration, robots can succeed in visiting all energy tanks.

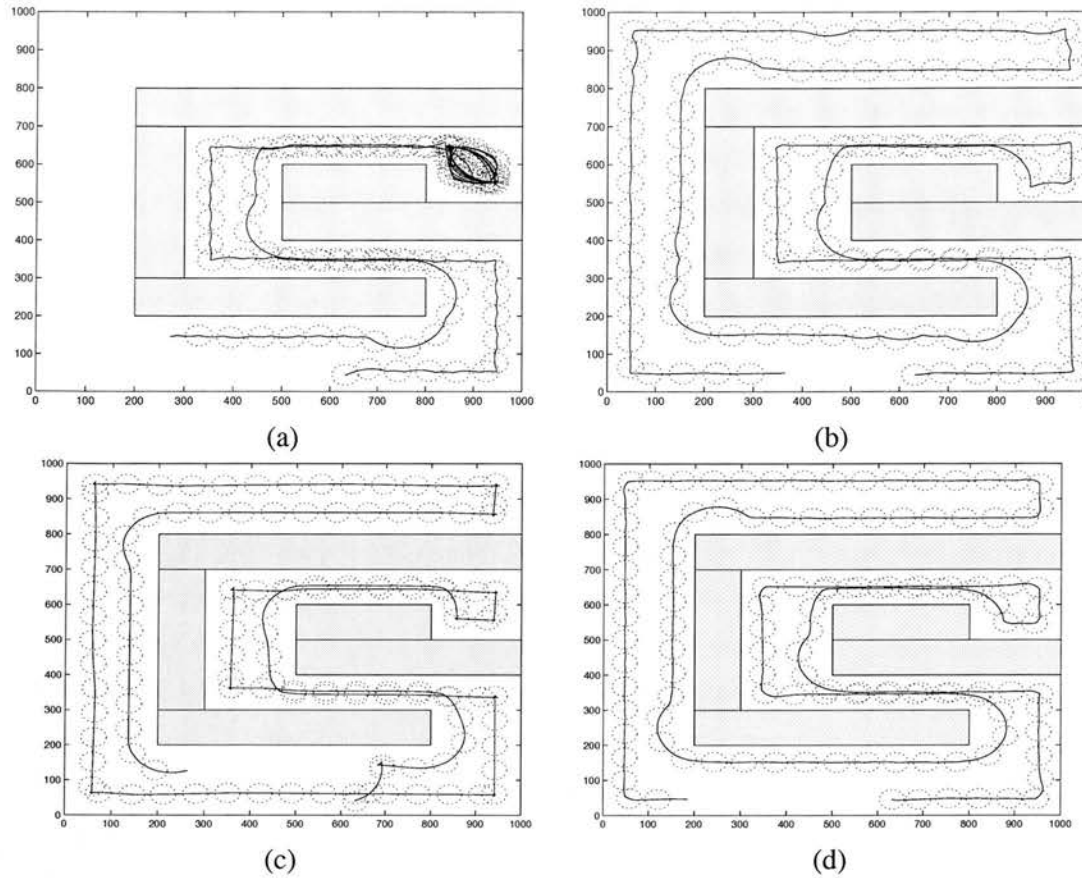
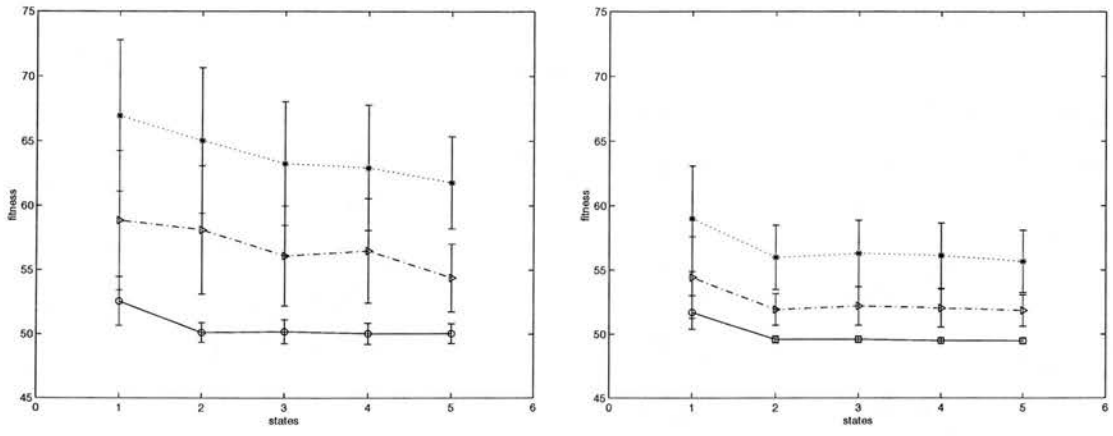


Figure 8.14: An example of corridor following behavior (a) purely reactive controller with two sensors (b) two state machine controller with two sensors (c) neural network with two sensors (d) purely reactive controller with four sensors

The maze-style arena has a room with a small exit at the end of the corridor in the middle section. Purely reactive control systems have difficulty in finding the exit directly and repeat moving around inside the room as shown in Figure 8.14(a). This behavior tends to be sensitive to noise. In some cases they easily find the exit by accident and in others they stagger around the room for a long time. They normally get a low score for visiting energy tanks within a limited time amount, because they spend much time in the room and cannot cover other areas. Memory-based systems with at least two states follow walls closely and so efficiently find the exit. Figure 8.14(b) shows their different behaviors. In two state controllers, for instance, state information is used to sequentialize two kinds of motor actions: move forward quickly, and move forward in the right direction when there is no sensation. This helps speedy movement in a narrow corridor. The memory analysis in Figure 8.13 shows the difference in performance efficiency rather than the outcome of whether robots can succeed in following corridors without collision; for wall following behavior, internal memory is used to achieve the task in open areas.



(a) (b)

Figure 8.15: Memory analysis with 4 sensors for corridor following (a) 25 runs without removing bad runs (b) 25 runs with removing bad runs (dotted: 60 generations, dashdot: 100 generations, solid: 300 generations)

When the EMO approach was applied to four-sensor robots, purely reactive systems and memory-based systems were not so different in performance with 95% confidence intervals as shown in Figure 8.15(a). By trial and error, bad performance runs were discarded and 25 good running experiments were collected to see the EMO result. Figure 8.15(b) shows significant difference in performance between purely reactive systems and memory-based systems. In a similar way to two-sensor robots, memory sequentializes two kinds of motor actions to speed up when no sensation is detected. When 1200 time steps are assigned, the performance of memoryless systems is lower than that of memory-based systems. However, purely reactive systems with four sensors can achieve perfect scores at every position with 2000 exploration time steps. More than two sensors can easily detect environmental situations and their corridor following is smoothly working. The two sensors in front $(78°, 102°)$ among four sensors can detect how close to walls the robots are, and they play a significant role of avoiding collision. If robots feel too close to walls with the front sensors, they first move backwards in a biased direction and then move forward. When two-sensor performance is compared with four-sensor performance (see Figure 8.13 and Figure 8.15), four-sensor robots show much better performance than two-sensor robots with the same size of memory. Furthermore, memoryless controllers with four sensors are slightly better in average fitness than multi-state controllers with two sensors, though it is not a significant difference (this will be checked in detail while testing controllers).

Neural networks, as examples of purely reactive systems were evolved with two and four sensors. One hidden layer with four hidden nodes were used for every experiment. The corridor

environment can have an optimal fitness of 49 (51 energy tanks are placed in the arena) and neural networks achieved this optimal fitness. Figure 8.14(c) shows neural network control behavior with two sensors. The controller utilizes the information of continuous sensor range and constructs local features in sensor readings. For instance, when the neural network recognizes the sensation that the robot is very close to a wall, it produces motor actions for moving backward in the left direction until the right sensor is free of sensation. Then it continues to follow corridors. This kind of behavior has often been observed with two binary sensors and memory states. Neural networks can process a continuous range of sensor readings and effectively use a quantitative information over sensors. However, relatively many weight parameters are used in neural networks. It is notable that controllers with a few states and binarized sensors do almost as well as feedforward neural network controllers with continuous sensor ranges.

8.3.2 Testing Controllers in Various Environments

The arena in Figure 8.12 is used for evolving controllers. The environment, called `env0`, has 30 random starting positions. Various controller structures are evolved to find the best fitness in the environment. The same arena, but with 100 new random positions, will be tested to see how correctly evolved controllers work in the given environment. This environment, named `env1`, is shown in Figure 8.16(a). An environment, `env2` is created with long corridors in the zigzag path. The corridor is so narrow that only one robot can move in the path, and 100 positions are randomly selected in the arena, as displayed in Figure 8.17(a). Another environment, `env3`, with a maze-style arena and narrow corridors, shown in Figure 8.18(a), will be tested with 100 random positions. For the environments `env0` and `env1`, 51 energy tanks are distributed and for the environments `env2` and `env3`, 55 and 60 energy tanks are placed at blank spaces (one space is assumed as a 10 cm by 10 cm square space). The optimal fitness for `env0`, `env1`, `env2` and `env3` is 49, 49, 45 and 40, respectively. The best controllers were obtained via the evolutionary process with N-K sample method and then they were tested over four environments. Each controller chromosome was evaluated 10 times. Two sensors and four sensors were tested with states ranging from one to three, respectively. Also, feedforward neural networks were evaluated with two and four sensors. The average number of collisions and the average fitness are listed with standard deviations in Tables E.4-E.6 (see appendix E). The best evolved behaviours with minimal collisions and the best fitness are shown in Table 8.2.

For two sensors, purely reactive controllers with binarized sensors are worse than two state

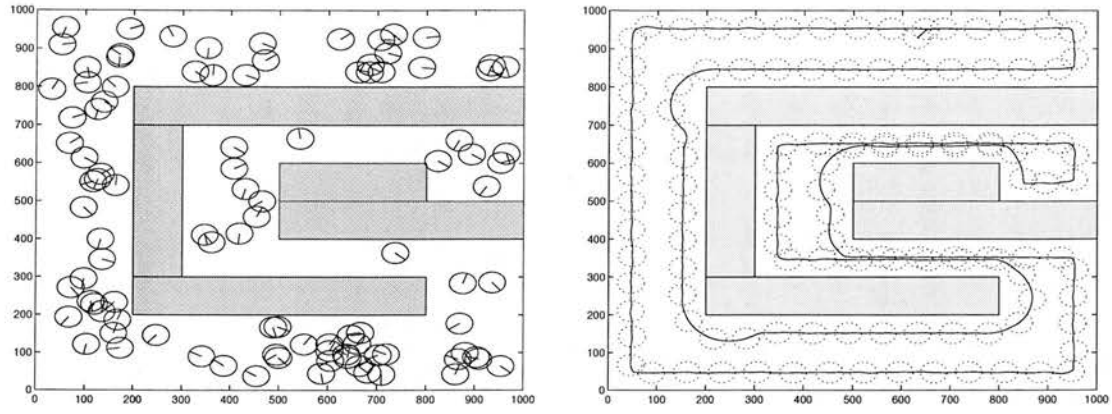


Figure 8.16: Test environment `env1` for corridor following (a) 100 random positions and directions (b) an example of testing the best two-state machine at one position

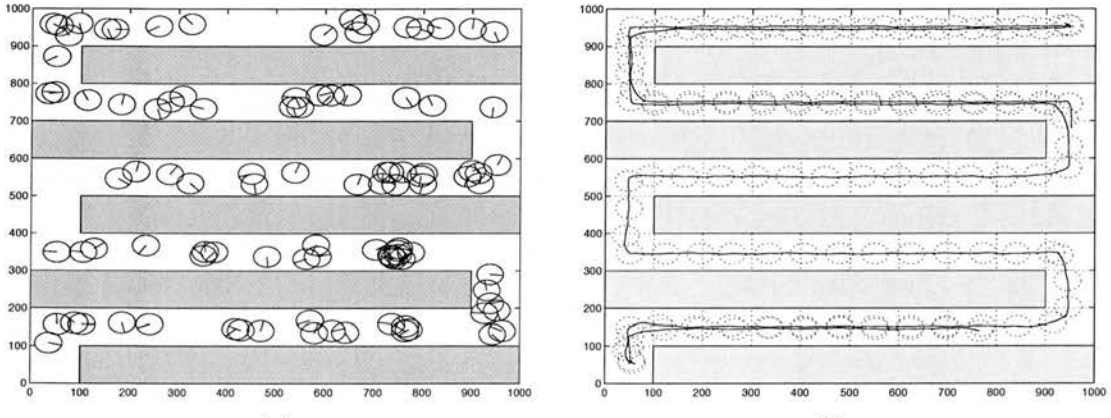


Figure 8.17: Test environment `env2` for corridor following (a) 100 random positions and directions (b) an example of testing the best two-state machine at one position

controllers in fitness performance, while the former is better in collision performance than the latter. Purely reactive systems tend to take safe and slow actions. There is a trade-off between the two measures, collision and fitness performance. When the fitness performance becomes better, the number of collisions is increased (see c_7, c_9 performance with 2 sensors, 1 state for test environments). Two state controllers much improved the fitness performance for every environment. Three state controllers have similar performance to two state controllers but reduced the number of collisions.

Adding two more sensors in front remarkably reduced the number of collisions for every state machine, while it did not improve the fitness performance to a large degree. More sensors means more information about the surrounding environment and thus the number of sensors

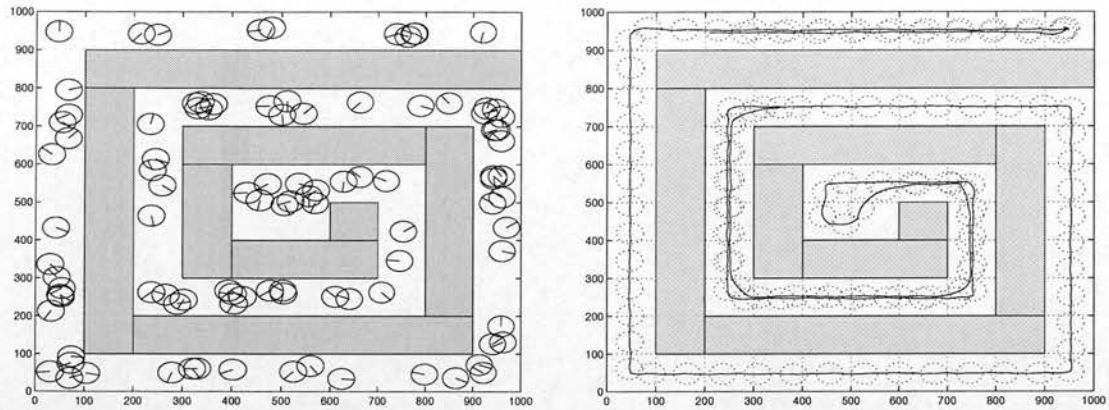


Figure 8.18: Test environment `env3` for corridor following (a) 100 random positions and directions (b) an example of testing the best two-state machine at one position

can significantly influence the design of robust controllers. Neural networks with only two sensors have good fitness performance and rare collisions. They have a finer scale of sensor readings than binarized sensor modeling and they easily adapt themselves to find local features for sensors. Even in neural networks, four sensors improved the performance of collision and fitness.

Two state machines with two sensors have several collisions in test environments. Two-sensor robots cannot easily detect sharp corners in front. It rarely happened that robots collide with sharp corners during their initial movements. However, once robots began to follow corridors, no collision occurred. The corridor following behavior shows the potential of purely reactive systems. Increasing the number of sensors is better for recognizing environments than increasing memory. Table 8.2 shows that memoryless systems with four sensors are better in test environments than memory-based systems with two sensors in terms of collision and fitness performance. When the number of sensors or sensor range is restricted, for example, only two sensors are used, or binarized sensor modeling is used, internal memory significantly influences the performance. In corridor following behavior, internal states can link two different motor actions to speed up robot movements.

From Table 8.2, the best control structure for corridor following is the neural network controller with four sensors (c_9) or two state machines with four sensors (c_4). The corridor following behaviors with the best control system (two state machine) are displayed in Figure 8.16(b), Figure 8.17(b) and Figure 8.18(b).

control structure		env0	env1	env2	env3
2 sensors, 1 state	c_7*	0.0 ± 0.0	1.2 ± 0.4	2.0 ± 0.4	5.1 ± 1.3
	c_9*	0.0 ± 0.0	1.2 ± 0.4	2.0 ± 0.0	1.8 ± 1.0
2 sensors, 2 states	c_6*	0.0 ± 0.0	1.8 ± 0.4	4.5 ± 0.5	3.0 ± 0.9
	c_7*	0.0 ± 0.0	1.9 ± 0.5	4.6 ± 0.7	2.0 ± 1.0
2 sensors, 3 states	c_4*	0.0 ± 0.0	2.2 ± 0.4	2.1 ± 0.3	3.7 ± 1.0
	c_7*	0.0 ± 0.0	4.1 ± 0.9	2.4 ± 0.5	4.7 ± 1.1
4 sensors, 1 state	c_4*	0.0 ± 0.0	0.0 ± 0.0	1.0 ± 0.0	0.6 ± 0.5
	c_8*	0.0 ± 0.0	0.0 ± 0.0	1.0 ± 0.0	0.2 ± 0.4
4 sensors, 2 states	c_4*	0.0 ± 0.0	0.0 ± 0.0	0.3 ± 0.5	0.0 ± 0.0
	c_6*	0.0 ± 0.0	0.1 ± 0.3	1.1 ± 0.8	1.0 ± 0.0
4 sensors, 3 states	c_6*	0.0 ± 0.0	0.2 ± 0.4	0.5 ± 0.5	0.2 ± 0.4
	c_9*	0.0 ± 0.0	0.6 ± 0.8	1.0 ± 0.0	0.3 ± 0.5
2 sensors, neural	c_4*	0.0 ± 0.0	1.2 ± 0.4	0.7 ± 0.6	0.5 ± 0.5
	c_9*	0.0 ± 0.0	0.0 ± 0.0	1.5 ± 0.5	0.0 ± 0.0
4 sensors, neural	c_2*	0.0 ± 0.0	0.0 ± 0.0	0.7 ± 0.5	1.0 ± 0.0
	c_9*	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	1.0 ± 0.0

(a) collision test

control structure		env0	env1	env2	env3
2 sensors, 1 state	c_7*	59.66 ± 1.22	59.46 ± 0.34	54.68 ± 0.08	55.84 ± 0.54
	c_9*	59.05 ± 1.02	59.15 ± 0.37	61.15 ± 0.21	61.37 ± 0.35
2 sensors, 2 states	c_6*	50.03 ± 0.09	50.49 ± 0.07	49.03 ± 0.10	47.24 ± 0.09
	c_7*	50.11 ± 0.12	50.65 ± 0.22	49.02 ± 0.06	47.33 ± 0.21
2 sensors, 3 states	c_4*	50.92 ± 0.27	51.57 ± 0.07	49.78 ± 0.07	47.54 ± 0.12
	c_7*	50.16 ± 0.15	51.81 ± 0.07	49.98 ± 0.17	47.49 ± 0.15
4 sensors, 1 state	c_4*	50.19 ± 0.15	50.73 ± 0.09	48.96 ± 0.05	46.97 ± 0.16
	c_8*	50.19 ± 0.13	50.79 ± 0.09	49.33 ± 0.15	46.66 ± 0.22
4 sensors, 2 states	c_4*	49.14 ± 0.16	49.19 ± 0.09	48.02 ± 0.16	46.03 ± 0.14
	c_6*	49.09 ± 0.08	49.09 ± 0.10	47.79 ± 0.15	44.98 ± 0.06
4 sensors, 3 states	c_6*	49.10 ± 0.09	49.51 ± 0.06	48.80 ± 0.19	46.67 ± 0.29
	c_9*	49.02 ± 0.07	49.11 ± 0.05	47.62 ± 0.11	45.30 ± 0.22
2 sensors, neural	c_4*	49.54 ± 0.05	49.55 ± 0.06	47.96 ± 0.07	45.37 ± 0.18
	c_9*	49.76 ± 0.08	49.82 ± 0.06	48.36 ± 0.16	45.85 ± 0.15
4 sensors, neural	c_2*	49.00 ± 0.00	49.47 ± 0.00	46.61 ± 0.07	43.70 ± 0.04
	c_9*	49.00 ± 0.00	49.00 ± 0.01	47.50 ± 0.04	44.44 ± 0.39

(b) fitness performance

Table 8.2: Comparison of performance with various control structures for corridor following (each data shows the average fitness and standard deviation over 10 evaluations)

8.4 Box Pushing Behavior

In chapter 6, the box pushing task was tested with two whisker-like binary sensors. This would appear to be a complex task, but the evolutionary result proved that purely reactive systems are sufficient to achieve the behavior. The box pushing behavior will be tested again with realistic simulation. In noisy environments, sensor readings for a box are not stable and when the box is at a little distance but nearly outside the sensor range of the robot, the box may not be sensed. In this experiment, we will see how control systems handle the problem. There is still a question of whether purely reactive systems can achieve box pushing task in spite of noise. Realistic simulation and $\pm 10\%$ random noise were applied to sensor readings and motor actions in a

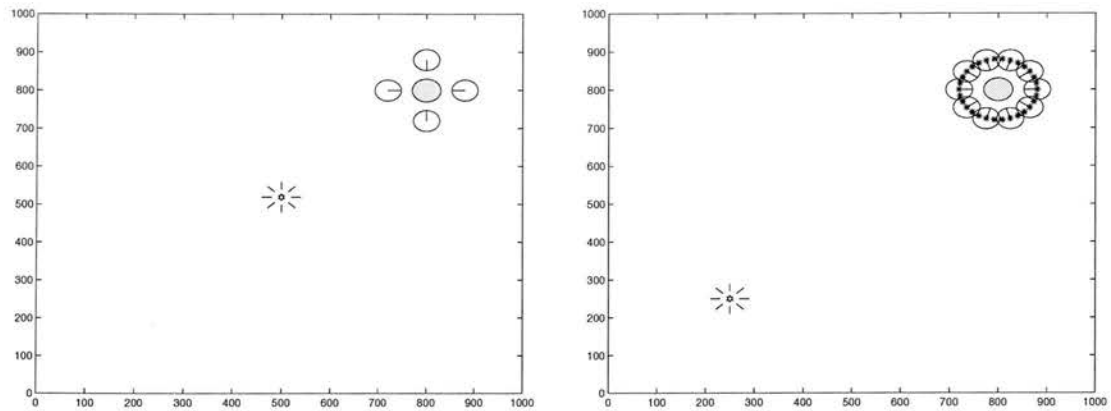


Figure 8.19: World environment for box pushing (a) evolving environment `env0` (b) test environment `env1`

similar way to the above experiments. Ambient light sensor values to represent light intensity were also influenced by a noise of $\pm 5\%$. Infrared sensor readings range from 0 to 1023, and the values were binarized with threshold 500.

Initially two infrared sensors $(45°, 135°)$ in front and two light sensors with the same angles are used. The box radius is 30 mm and it can be sensed by two infrared sensors when the robot is close enough to the box. In the previous experiments with whisker-like infrared sensors, the box radius was larger than 30 mm in order to detect the moment that robots are in contact with or very close to the box. In evolutionary experiments, robots have four sample positions as shown in Figure 8.19(a), and light is placed in the middle of the arena. At every sample position, robots face the circular box but are not close enough to the box so that they can sense it. This requires the robot to move forward at the initial step. Two light sensor vectors will say which sensor detects a higher intensity of light, instead of the intensity value. In other words, two vectors for ambient light sensors are assumed to indicate whether light is on the left or on the right, after two light sensors read and compare intensity values.

For evolved controllers, a test environment, which will be called `env1`, is given with 30 sample positions around the box, as shown in Figure 8.19(b). 30 samples are selected with uniformly distributed angles around the box at every 12 degrees. At each starting position, robots face the box with the same distance as in the evolving environment, but light is placed in the lower left corner. Light is at a further distance than experienced in the evolving environment. Successful robot controllers should push the box from the upper right corner to the light without losing contact with the box.

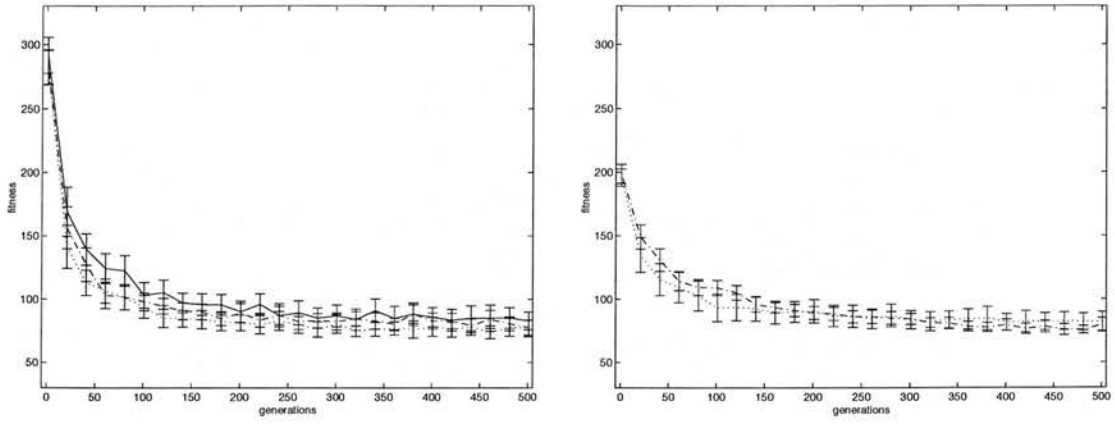


Figure 8.20: Average fitness distribution for box push task (a) two infrared sensors (b) four infrared sensors (dotted: 1 state, dashdot: 2 states, solid: 3 states)

In evolutionary experiments, the environment has 4 sample positions as in Figure 8.19(a). The fitness function for one position is defined as a penalty function as follows:

$$f_B = \Sigma_{t=1}^{T}(0.01W(t) + 0.1D(t))$$

where $W(t)$ is the distance between robot and box and $D(t)$ is the distance between box and light. In the experiments, 500 time steps are used for the exploration time T. The population size was 50, and fitness-proportional selection was applied for genome reproduction. Every genome was evaluated with 4 sample positions and the average score was taken as its fitness. The evolution process ran for 500 generations.

In these box pushing experiments, the following issues are addressed.

- How many memory elements (states) are required to push the box towards the light with realistic simulation?
- Does noise influence the memory requirement?
- Which control structures will be useful to evolve controllers, among FSMs, rule-based structures and neural networks?
- What would be the role of memory if memory states are necessary?
- If the box size is changed, does it influence the memory requirement and the behavior required to push the box?
- What are the difficult positions among the sample positions in the box pushing task?

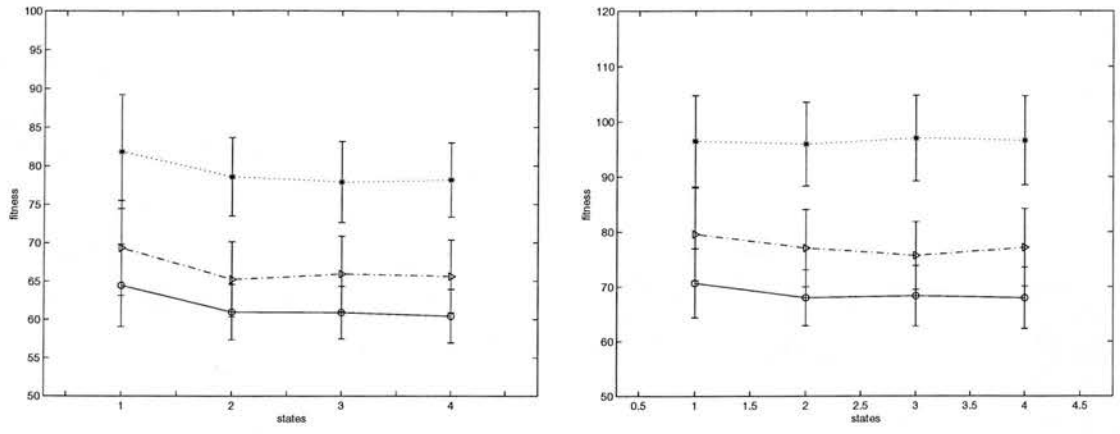
control structure		failure test		performance test	
(sensor, memory)	no.	env0	env1	env0	env1
2 infrared sensors, 1 state	c_0*	0.2 ± 0.4	1.8 ± 1.2	56.57 ± 7.08	127.31 ± 5.39
	c_5*	0.1 ± 0.3	1.8 ± 1.5	62.29 ± 2.91	134.71 ± 7.48
2 infrared sensors, 2 states	c_1*	0.0 ± 0.0	1.2 ± 1.1	67.66 ± 6.86	145.93 ± 6.55
	c_8*	0.0 ± 0.0	2.6 ± 1.3	55.52 ± 3.09	118.69 ± 6.35
2 infrared sensors, 3 states	c_6*	0.0 ± 0.0	2.2 ± 1.2	57.26 ± 1.90	118.08 ± 7.35
	c_8*	0.0 ± 0.0	1.8 ± 1.0	60.63 ± 2.77	132.92 ± 6.55
4 infrared sensors, 1 state	c_2*	0.0 ± 0.0	1.2 ± 0.6	62.36 ± 3.03	147.59 ± 4.87
	c_8*	0.0 ± 0.0	1.1 ± 0.8	59.45 ± 2.93	137.82 ± 8.26
4 infrared sensors, 2 states	c_2*	0.0 ± 0.0	0.0 ± 0.0	61.62 ± 2.32	154.33 ± 1.91
	c_4*	0.0 ± 0.0	0.4 ± 0.5	59.06 ± 6.19	138.13 ± 7.81
4 infrared sensors, 3 states	c_5*	0.1 ± 0.3	2.7 ± 1.7	86.31 ± 4.80	139.56 ± 3.76
rule-based	c_6*	0.0 ± 0.0	0.0 ± 0.0	75.05 ± 3.46	170.64 ± 1.60
4 infrared sensors,	c_0*	0.0 ± 0.0	3.2 ± 1.2	76.73 ± 11.34	154.66 ± 7.21
neural network	c_2*	0.0 ± 0.0	0.2 ± 0.4	77.94 ± 5.52	162.25 ± 1.53

Table 8.3: Comparison of performance with various control structures for box pushing (each data shows the average score and standard deviation over 10 evaluations)

8.4.1 Evolving Various Controllers

Two infrared sensor robots with memory states from one to three were evolved with the FSM structure. In a similar way, four infrared sensor robots, adding two more sensors $(78°, 102°)$ in front, were evolved. Each experiment was run 25 times to validate the fitness performance. By the average fitness from the fitness distributions (see Figure 8.20), purely reactive controllers are not significantly different from memory-based controllers in fitness performance. It is also shown that two infrared sensor robots have similar performance to four infrared sensor robots, comparing Figure 8.20(a)-(b). It is thus believed that two proximity sensors are sufficient to push the box towards the light. Four proximity sensors have smaller variances in the fitness distribution and more sensors tend to produce more stable results in noisy environments.

Table 8.3 shows the two best collections of desirable chromosomes (more chromosome tests are shown in Table E.7 and Table E.9 – see appendix E). Each of the best chromosomes is evaluated 10 times in the evolving environment, `env0`, and test environment, `env1`. Two criteria, the performance test and failure test, are applied to choose desirable controllers. Failure is defined as the case when robot controllers are not successful at pushing box towards light, for example, robots collide with walls or the distance between the final box position and light position is more than 20 cm. The failure test is to count how many failures occur in evaluating a given chromosome over all starting positions, and it is averaged over 10 evaluations. The performance test is to see the average fitness score over 10 evaluations. Each column in Table 8.3 shows the average score and standard deviation of both tests. The best collections of two-sensor robot controllers show that there is no significant difference depending on the number of



(a) (b)

Figure 8.21: Memory analysis for box pushing behavior (a) two infrared sensors (b) four infrared sensors (dotted: 100 generations, dashdot: 300 generations, solid: 700 generations)

memory states. When a controller has good performance in the evolving environment, it also shows good performance in the test environment.

There were a few failures with the controllers in the test environment `env1`. The failures happened because the robots continue to move around the box instead of pushing it, even though they are in contact with box. Robots have no information of light intensity from sensor vectors and so they evolved to respond to which side light is placed. In some situation robots may behave as they do when the light is very near the box. Increasing the number of infrared sensors to four can reduce the number of such failures by pushing the box in a more delicate way; the best controllers with two memory states and four infrared sensors have almost no failures.

Controllers with four binarized infrared sensors have similar performance to neural network controllers with continuous-valued infrared sensors; with neural network controllers, two light sensor vectors are used in the same way. Binarized information with one threshold is sufficient to evolve desirable controllers for box pushing. According to the performance of the best collections, memory states do not improve the performance from purely reactive control systems. This is consistent with the result of average fitness performance in Figure 8.20(b).

The evolutionary multiobjective optimization (EMO) is applied to box pushing behavior to see the effect of memory, as shown in Figure 8.21. There is a slight difference in the average fitness between purely reactive controllers and two state controllers, but their difference is not significant with a 95% confidence interval over 25 runs. It is confirmed again that memoryless controllers can achieve the box pushing task.

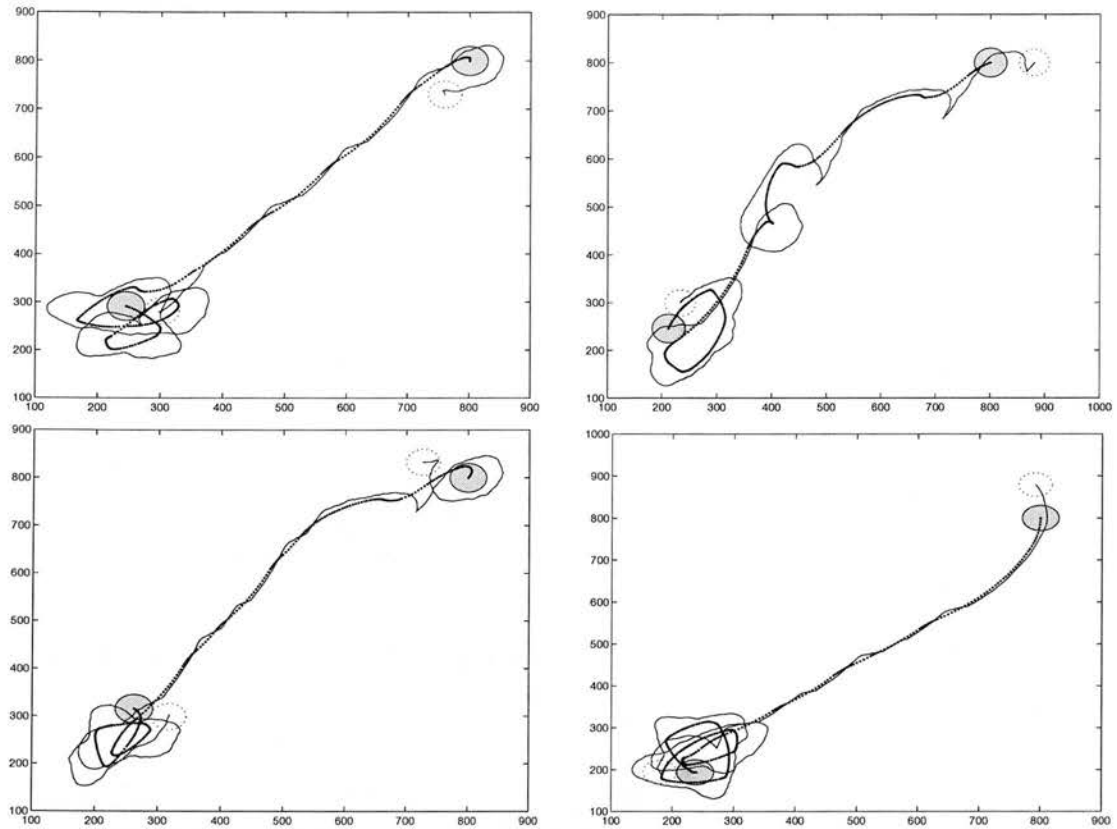


Figure 8.22: Box pushing behavior of purely reactive controller with two sensors

Box pushing behavior needs no internal state, if the environmental markers exist for light. Figure 8.20 and Figure 8.21 show the fitness performance with different memory states. Their performances are not significantly different and show similar fitness curves. The number of sensors is increased so that the robots can read the environmental information better. Adding two more infrared sensors in front can reduce the danger of losing contact with the box and produces a more stable fitness distribution; when robots lose the box, they stagger around the arena looking for the box and incur penalties. Four sensors will have more desirable solutions. Neural networks with two and four infrared sensors also show high fidelity solutions. More sensors or more fine-grained sensors are useful for robust controllers, but it is shown that simple binary sensors can work for the box pushing task.

No advantage of internal memory is displayed in the performance of average fitness even with two infrared sensors. The best chromosomes are tested at 30 samples positions in the test environment, as shown in Figure 8.19(b). We will observe the role of memory states in evolved controllers.

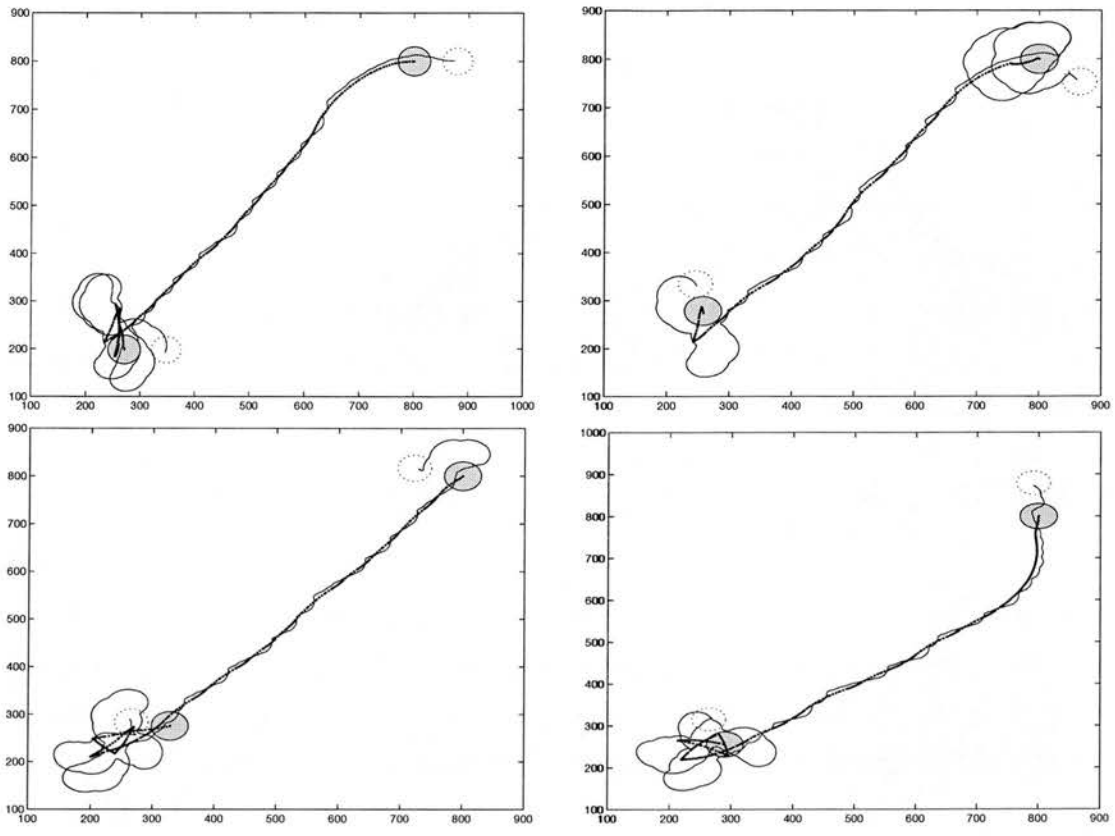


Figure 8.23: Box pushing behavior of two state controller with two sensors

8.4.2 Robot Behaviors

Figure 8.22 demonstrates the behavior of one of the best purely reactive controllers with two proximity sensors and two light sensors (c_5 in Table 8.3). Its strategy is simple as follows: the robot pushes the box diagonally towards the light when either one of the proximity sensors is active. With no sensation of any infrared sensor, it rotates counter-clockwise on the spot. Rotating in one direction can help to sense the box again when the robot loses contact with it. If both proximity sensors are active, the robot moves diagonally away from the light. Pushing the box at an angle, by the difference between left and right motor actions, depends on the light direction. If the box is sensed at one side with an infrared sensor and the light is detected at the other with the light sensors, the robot circles around the box by repeating moving forward and turning left; this is similar to a sequence of actions observed in wall following behavior. Otherwise, the robot pushes the box towards the light.

Figure 8.23 displays behavior examples of pushing the box with one of the best two state controllers (c_8 in Table 8.3). In this case, the robot efficiently uses its internal states. The strategy

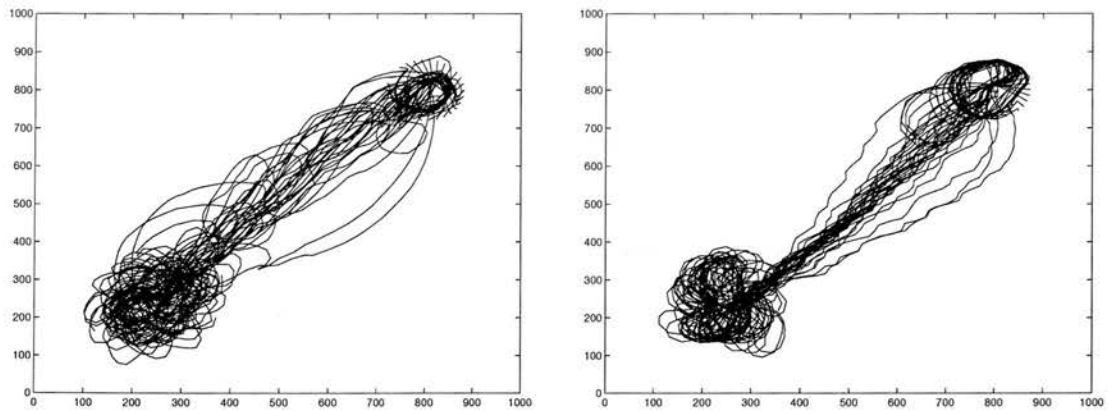


Figure 8.24: Trace of robot moving to push box with 30 sample positions (a) purely reactive controller (b) two state controller

is similar to the purely reactive controller as mentioned above. However, at the moment that the robot is almost at a position near the box with the light in front, the robot is ready to change its internal state. The internal state indicates that the robot can push the box straight forward. The internal state is marked when the direction of the two light sensor vectors is changed, or when only the left proximity sensor becomes active from the moment that both sensors are active. In this state, the robot rotates anticlockwise and checks whether both proximity sensors get switched on. If both sensors are active, then the robot moves straight forward. Otherwise, it returns to the previous internal state. This internal state does not greatly improve the global performance because such situations rarely occur. Yet the robot pushes the box in an elegant way as shown in Figure 8.23.

The evolutionary process found such an efficient and flexible controller with memory states. Thus, it can be said that internal states play a role in developing more intelligent controllers. Figure 8.24 shows the trace results of robots explained above. The memory-based controller has a tendency to push the box in a more direct path towards the light.

8.4.3 Experiments with Small-Sized Box

In the above experiments the box size is similar to the robot size, such that two proximity sensors easily detect the box. The following experiments investigate whether the box size can influence box pushing behavior and memory effects. A small-sized circular box with radius 17.5 mm will be tested at this time for evolving controllers using the same environment. This box is so small that two infrared sensors at angle 45° and 135° are rarely active together when

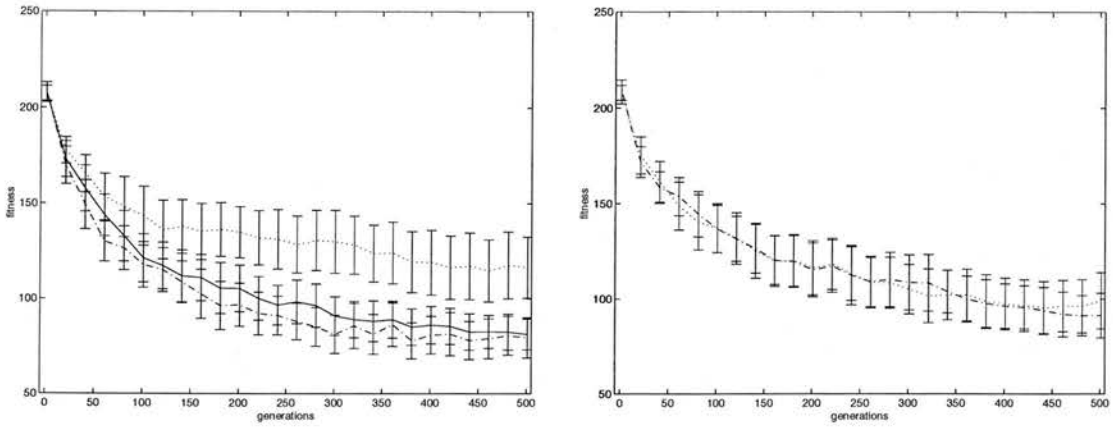


Figure 8.25: Average fitness distribution for pushing a small box (a) two infrared sensors (b) four infrared sensors (dotted: 1 state, dashdot: 2 states, solid: 3 states)

the box is placed in front of the robot.

The previous experiments were repeated with the small box. Different numbers of memory states with two proximity sensors were first tested to see how many memory states are required to push the small box towards light. Figure 8.25(a) shows that this small box pushing behaviour has a differentiated performance with memory states. Each given control structure has been evolved 25 times and its average fitness distribution is obtained with 95% confidence intervals. When only two proximity sensors are used, purely reactive controllers can achieve the task but using the fitness test, their performance is significantly worse than the performance of memory-based controllers. When the best case of purely reactive controllers is considered, its fitness reaches 60, while the best two state controllers reaches 50.

When we perform the effort test by assuming that the desirable performance is 80, purely reactive controllers obtained 4 successes among 25 runs. Two state controllers and three state controllers produced 15 and 14 successes among 25 runs, respectively. With 95% confidence intervals, the expected computing cost to reach the fitness 80 or below will be $[2.87C, 15.26C]$ for 4 successes, $[1.30C, 2.46C]$ for 15 successes, and $[1.36C, 2.71C]$ for 14 successes, where C is a unit computing cost for one single evolutionary run. In terms of the effort cost, the memoryless system is significantly different from memory-based systems. Purely reactive systems have difficulty in evolving desirable controllers with two proximity sensors within 500 generations. If their evolutionary runs continue for many generations, it is presumed that this may reduce the gap of fitness distributions between memoryless systems and memory-based systems for box pushing behavior (the fitness curve of purely reactive controllers in Figure 8.25(a)

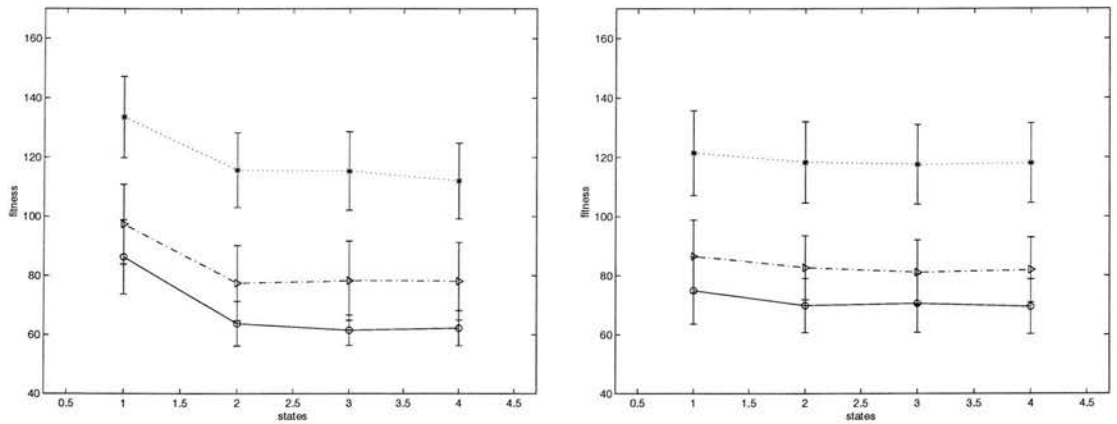


Figure 8.26: Memory analysis for box pushing behavior with small box (a) two infrared sensors (b) four infrared sensors (dotted: 100 generations, dashdot: 300 generations, solid: 700 generations)

is slowly progressing). However, the EMO application result for small box pushing experiments shows that memory-based systems are significantly better than purely reactive systems (see Figure 8.26(a)).

Four proximity sensors (two more proximity sensors in front) can improve the performance of purely reactive controllers as shown in Figure 8.25(b). There is no significant difference between memoryless system and memory-based systems. The EMO approach for memory analysis also produces the same result as shown in Figure 8.26(b).

Table 8.4 shows the performance results of the two best chromosomes for small box pushing experiments; they are selected from good controllers recorded in Table E.8 and Table E.10 (see appendix E). The results of two infrared sensors show that the best chromosome of purely reactive systems is worse than that of memory-based systems in both the failure test and the performance test, especially with test environment `env1`.

One of the best controllers with two sensors and two memory states (c_1 in Table 8.4) gets few failures and good fitness performance. Its behavior is demonstrated in Figure E.2 (see appendix E). The robot controller prefers to push the box at an angle with the left front portion of the robot's body. If the robot sees a change of light direction while continuing to push the box, it changes its internal state and then tries to circle around the box anti-clockwise. If it does not sense the box, then it suddenly rotates clockwise with maximum speed such that both the box and the light are seen on the left side of the robot. Then it continues to push the box again.

Figure 8.27 shows the fitness distribution of a rule-based system and neural network. The rule-

control structure		failure test		performance test	
(sensor, memory)	no.	env0	env1	env0	env1
2 infrared sensors, 1 state	c_6*	0.0 ± 0.0	2.1 ± 0.9	60.58 ± 10.66	148.09 ± 6.25
	c_7*	0.0 ± 0.0	1.1 ± 1.0	62.76 ± 8.59	145.94 ± 9.15
2 infrared sensors, 2 states	c_1*	0.0 ± 0.0	0.1 ± 0.3	50.85 ± 2.43	123.87 ± 3.07
	c_7*	0.1 ± 0.3	0.4 ± 0.7	51.30 ± 3.02	119.05 ± 2.72
2 infrared sensors, 3 states	c_4*	0.0 ± 0.0	0.3 ± 0.5	53.31 ± 1.94	122.00 ± 5.31
	c_8*	0.0 ± 0.0	0.0 ± 0.0	54.57 ± 1.92	138.83 ± 2.74
4 infrared sensors, 1 state	c_3*	0.1 ± 0.3	0.5 ± 0.9	60.45 ± 6.09	143.37 ± 4.66
	c_7*	0.0 ± 0.0	0.4 ± 0.5	51.90 ± 1.74	123.87 ± 4.98
4 infrared sensors, 2 states	c_0*	0.0 ± 0.0	0.0 ± 0.0	53.04 ± 2.40	121.18 ± 4.39
	c_6*	0.1 ± 0.3	0.2 ± 0.4	57.74 ± 2.11	128.54 ± 3.34
4 infrared sensors, 3 states rule-based	c_0*	0.0 ± 0.0	0.9 ± 0.9	68.13 ± 2.81	156.53 ± 1.63
	c_6*	0.0 ± 0.0	0.1 ± 0.3	54.35 ± 6.41	120.15 ± 2.30
4 infrared sensors, neural network	c_0*	0.0 ± 0.0	2.0 ± 1.5	71.03 ± 4.89	161.91 ± 4.85
	c_1*	0.0 ± 0.0	0.5 ± 0.5	67.58 ± 3.24	123.16 ± 4.14

Table 8.4: Comparison of performance with various control structures for pushing small box (each data shows the average fitness and standard deviation over 10 evaluations)

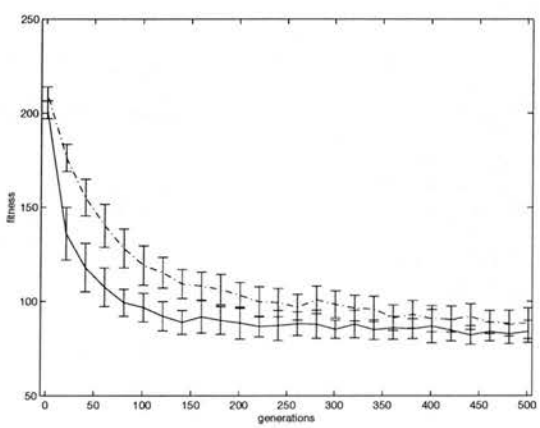


Figure 8.27: Fitness curve of rule-based structure and neural networks (dashdot: rule-based structure, solid: neural network controller)

based structure takes a little more time to reach stable fitness but it has a similar performance to the neural network structure. Rule-based structure allows maximally three states in memory and 30 rules are assumed for one controller design. Figure E.3 shows behavior examples of one of the best control structures (c_6 in Table 8.4). Its behavior has a winding path when pushing the box and the controller pushes the box with the left side of the robot when the light is on the left, and with the right when the light is on the right. As a result, it makes a zigzag movement.

8.4.4 *N-K* Sample Method for Difficult Positions

For the box pushing task, four sample positions were sufficient to evolve desirable controllers. However, another experiment was tested to see if more robot positions, for instance 30, can

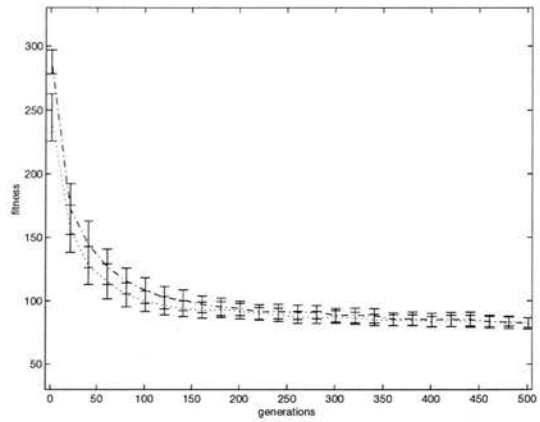


Figure 8.28: Average fitness in four infrared sensor robots with 30 sample positions (dotted: purely reactive controller, dashdot: two state controller)

accelerate the convergence speed of evolution for desirable controllers. In this case, the N-K sample method was applied with $N = 30, K = 4$. 30 sample positions help robots to experience more environmental situations than 4 samples. In Figure 8.28, the performance of $(N = 30, K = 4)$ is slightly better than that of $(N = 4, K = 4)$, although it is not significant. Box radius was 30 mm and population size was 50 and the evolutionary runs were repeated 25 times to see the fitness distribution. The error bars show 95% confidence intervals based on t-distribution. The error bars are smaller and more samples lead to a stable fitness distribution.

The ratio of computing costs in the N-K sample methods is follows as:

$$\frac{C_{30,4}}{C_{4,4}} = \frac{50 \cdot 4 \cdot c + 5 \cdot 30 \cdot c + \alpha}{50 \cdot 4 \cdot c + 5 \cdot 4 \cdot c + \alpha} = \frac{350 \cdot c + \alpha}{220 \cdot c + \alpha} \leq \frac{350}{220} \approx 1.59$$

where $C_{30,4}, C_{4,4}$ represent the computing costs for $(N = 30, K = 4)$ and $(N = 4, K = 4)$ experiments, respectively. Variable c is the computing cost for one evaluation at one sample position and α is the other computing cost including genome reproduction and selection. In the experiments, population size 50 was taken and the best chromosome in a population was evaluated five times to test robustness for each generation. With $(N = 30, K = 4)$, the best chromosome is evaluated at 30 sample positions five times and the best genome over 30 samples is inserted into a new population by elitism. The above experimental results say that a little more computing cost for 30 samples is worth while to achieve better and stable performance.

Figure 8.29 shows the frequency of each sample position tested for evolution when dynamic sample selection was applied with the fitness-proportional scheme. The data were obtained from the experiment with 25 runs over four-sensor reactive controllers. It indirectly indicates

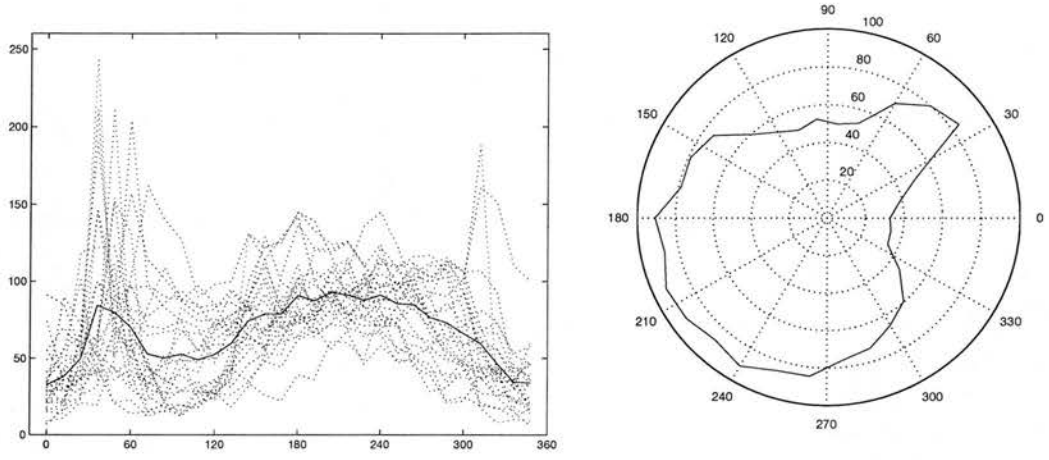


Figure 8.29: Frequency distribution of 30 samples using dynamic sample selection in box pushing behavior (dotted: each sample frequency, solid: average frequency)

which sample positions are difficult to evolve. Sample positions between 140° and 300° are relatively difficult positions for box pushing. Robots face the box behind the light and they need to circle around the box to find their proper location for pushing the box. At sample positions between 30° and 60°, robots face the box and light is in front, but the positions are ambiguous since robots cannot distinguish facing and backing light with the current light sensor configuration; two light sensors just say which of the two is more active. Without internal memory, robots managed to find appropriate controllers for such vague situations. When two infrared sensors in front are active, robots do not push the box straight forward way but prefer to push it at an angle.

8.5 T-Maze Behavior

Memory-based systems have used their internal states effectively to achieve the given robotic tasks. They are based on reactive control mapping with short-term memory and the internal states are treated as connectives of decomposable tasks or a series of actions. The following T-maze decision behavior will require long-term memory to remember the past perceptions. An agent will pass through a narrow corridor and must decide its directional move, left or right, depending on its perception experience when it sees the T-junction in front. This T-maze experiment has been first tested by Jakobi [1997]. In his experiments with one lamp, robots maintained internal states with recurrent neural networks to remember which side the lamp was on such that they could make a correct decision at the junction. In our experiments, we will

focus on quantifying internal states and also various light positions will be tested with one or two light lamps.

For the T-maze task, random controllers will have 50% probability for each direction, left and right. Desirable controllers will, for their choice of actions, depend on perceptual cues or landmarks that have been experienced while they are moving forward. Robots in the T-maze will use light intensity instead of complex landmarks. This will demonstrate an example of how robots can organize their memory from their perceptions. In the experiments, state machine controllers are applied to quantify the amount of memory.

Instead of allowing free movement[1], robots are forced to move forward until they reach the T-junction: this is unlike Jakobi [1997]'s work where robot controllers were evolved to go forward to the junction before the decision of turning. Two ambient light sensors at angle $22°, 158°$ and two infrared sensors at angles $45°, 135°$ will be used in the experiments. Infrared sensor readings and motor actions have a noise of $\pm 10\%$. The intensity of ambient light sensors are influenced with $\pm 5\%$ noise. A threshold of light intensity is assumed to binarize their intensity. While robots move forward from starting positions to the T-junction, their light sensors continue to change and binarized signals will flip from 0 to 1 or from 1 to 0, depending on how close robots are to light.

In evolutionary experiments, the fitness function is defined as follows:

$$f_T = \frac{1}{\kappa \cdot m} \sum_{j=1}^{\kappa} \sum_{i=1}^{m} E(d_{ij}, g_{ij}) \tag{8.1}$$

where d_{ij} is the direction that robots choose at the T-junction, g_{ij} is the direction for goal position, κ is the number of environments to test robots, and m is the number of experimental runs. Expression $E(a,b)$ is the equality function to be set to 1 if the directions of a, b are the same and otherwise 0. The fitness function is influenced by noise and it will prevent accidental success or failure of controllers by evaluating one robot controller several times (m times). As a result, the above fitness function will estimate the success rate that robots predict goal positions through a history of perceptions.

As the average performance over samples is considered, each chromosome in a population is evaluated five times to estimate the fitness in equation 8.1 ($m = 5$). The best chromosome of the population is evaluated again 25 times with $m = 25$ in equation 8.1 (more runs will produce

[1]The free movement in the approach corridor allows a purely reactive solution for one light experiment, using a wall-following strategy.

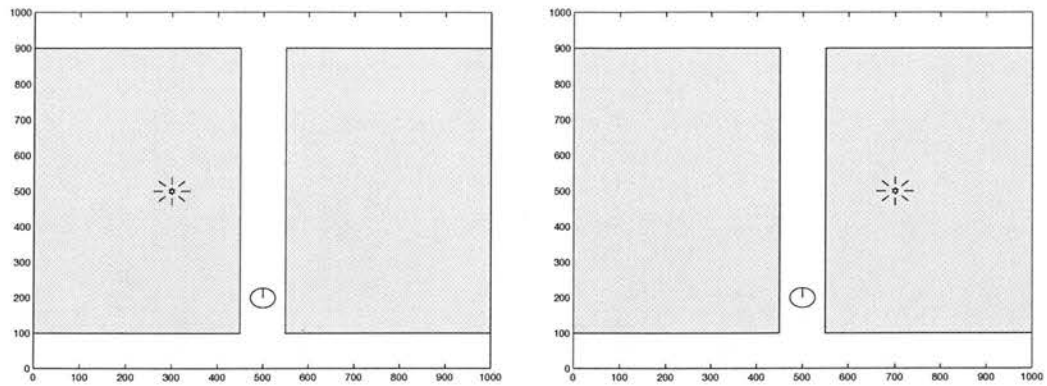


Figure 8.30: T-maze environment 1

a better estimate of the prediction success rate for the chromosome) and the best evaluation chromosome will be maintained for the whole evolutionary run. For the N-K sample method, we have $N = K = 2$ for one light experiments and $N = K = 4$ for two light experiments where N is the number of environmental configurations (this does not follow the method of sample selection because N is small).

8.5.1 One Light Experiments

First, a simple task was tested with one light. In the experiments, light is placed on the left or right block bank. For a landmark that the robot can experience while moving forward in the corridor, one light was used to make a difference in the environment. Light can be placed in the middle of each bank as shown in Figure 8.30; robots do not detect light initially, but when they move forward, they can sense the light as they approach it. When they move away, they cannot sense the light again. When robots are close enough to the T-junction, they decide which direction to move in. If the light is placed on the left side, robots should move left at the T-junction. The goal position is at the end of corridor on the left side. If light is placed on the right side, then robots should move right to reach the goal on the right side. The task will test if robots can do conditional movement depending on one position of the light.

The EMO approach is applied for memory analysis of the one light T-maze with population size 50 and 300 generations. It shows that the task requires two internal states as displayed in Figure 8.31. The experiment was run 25 times and its average performance is estimated with 95% confidence intervals. Purely reactive controllers have 50% success rate and they only choose one direction in any condition of perceptions. When standing at the T-junction, purely

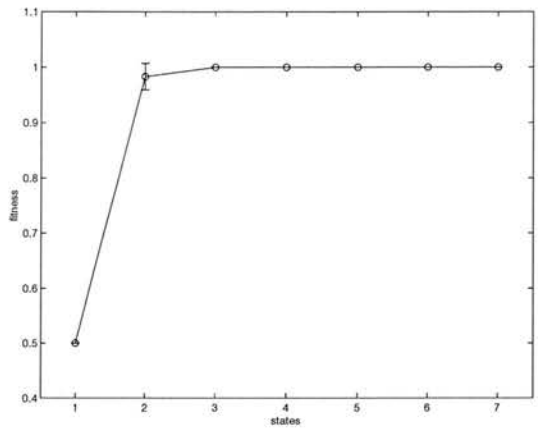


Figure 8.31: Memory analysis with one light experiment over maze problem

reactive robots do not see any light and they cannot consider any previous light condition. They can just choose to turn left or right regardless of light that has been seen in the corridor path. The best controllers with at least two states have internal states to be triggered by left light lamp or right light lamp. For example, a two-state controller has its internal state for turning left by default and ignores light on the left side but if it sees light on the right side, it changes its internal state into the mode for turning right. When the robot reaches the T-junction, the turning move will depend on its internal state.

However, the task can be purely reactive if robots can move freely in the corridor. When they see light on the left side, they can begin to turn left and follow left walls until they reach the goal position. Otherwise, they follow right walls. Robots determine their movement in advance before they cross the T-junction at the instant that they can see light. Thus, it can be easily achieved with purely reactive controllers. When robots are forced to move forward in spite of any perception, they internalise perceptual states using available memory states. The internal states will determine which move they should choose at the T-junction.

8.5.2 Two Light Experiments

More complex situations can be made with two lights. Four possible cases with two lights are created to distinguish environmental situations, as displayed in Figure 8.32; both lights can be placed on the left side or on the right, or one light is placed in the lower area on the one side and another light is in the upper area on the other side. Figure 8.32 also shows examples of noisy light intensity signals. If light is close to an ambient light sensor, its intensity becomes very low. The experiments use a threshold 250 to binarize the intensity signals.

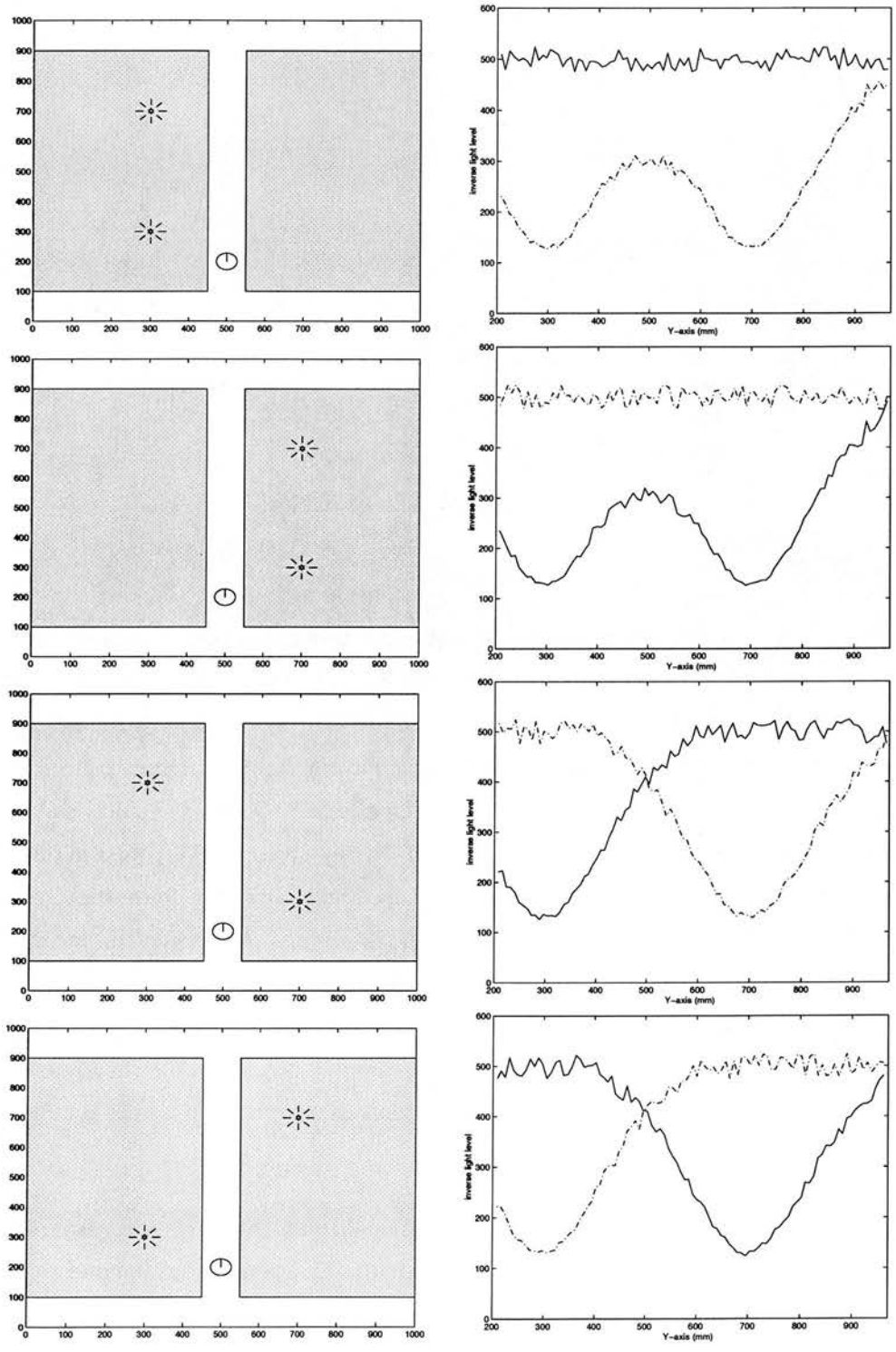


Figure 8.32: T-maze environment 2 (dashdot: left light sensor, solid: right light sensor)

cases	goal position	light position
case 1	left	both lights are on the left
case 2	left	one light in the upper left and the other in the lower right
case 3	right	both lights are on the right
case 4	right	one light in the lower left and the other in the upper right

(a) `tmaze-env1`

cases	goal position	light position
case 1	left	both lights are on the left
case 2	left	one light in the lower left and the other in the upper right
case 3	right	both lights are on the right
case 4	right	one light in the upper left and the other in the lower right

(b) `tmaze-env2`

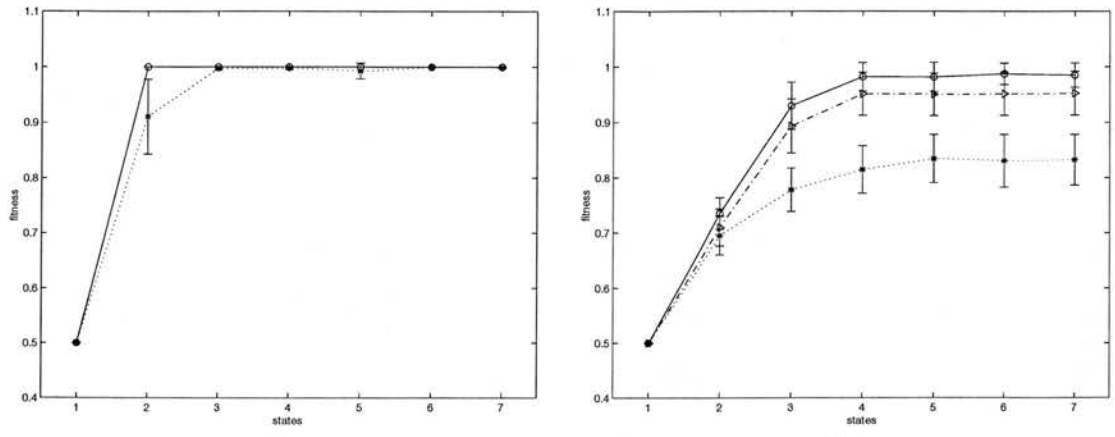
cases	goal position	light position
case 1	left	both lights are on the left
case 2	left	both lights are on the right
case 3	right	one light in the lower left and the other in the upper right
case 4	right	one light in the upper left and the other in the lower right

(c) `tmaze-env3`

Table 8.5: Experiments with two lights

We design three possible tests over four cases with two lights as in Table 8.5; they will be called `tmaze-env1`, `tmaze-env2`, `tmaze-env3`. Each experiment will be tested to see how many memory states are required to make correct decisions for moving to the goal position. The EMO approach is applied to each experiment for memory analysis with a population size of 50. The above experiments will evolve controllers which are attentive to light signals. For the setting `tmaze-env1`, goal positions depend on lights in the upper area and robots should be able to detect lights near the T-junction by ignoring lights near starting positions. Direction of turning left or right will rely on which side light is placed in the upper area. In the environmental setting `tmaze-env2`, goal positions will depend on lights in the lower area. Robots should recognize the direction of lights near the starting position and they should decide their turning actions using their first perception of light. For the setting `tmaze-env3`, the formation of light should be memorized to decide the move. It will be a more complex task since the perceptions of lights should be sequenced and the detection of the second light after the first light is required to make appropriate actions.

The `tmaze-env1` setting has two lights but the environmental situations require only two memory states. One of the best controllers changes its internal state depending on light sensation: while passing light on the left side, robots mark their internal state in advance so that they will move left in the future and when they reach the T-junction, they move left by following their internal state. If light is seen on the right side, robots change their internal state to move right later. While the robot moves forward in the corridor, it may see light on the opposite side. The



(a) (b)

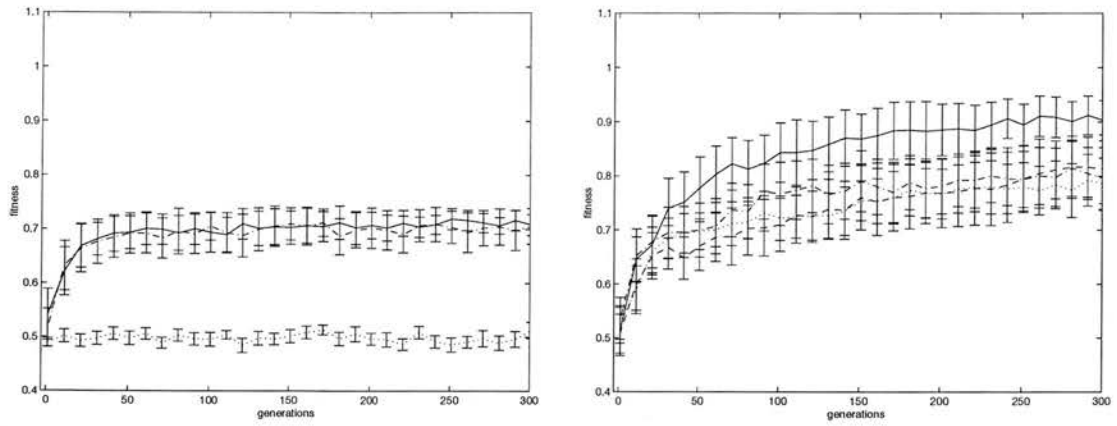
Figure 8.33: Memory analysis over T-maze problem `tmaze-env1` and `tmaze-env2` (a) `tmaze-env1` (b) `tmaze-env2` (dotted: 100 generations, dashdot: 200 generations, solid: 500 generations)

robot then toggles its internal state into the state of light direction. In this way two memory states, one state for each direction, are sufficient to predict goal positions correctly. The EMO approach result in Figure 8.33(a) also shows that two states are required to achieve the task.

Memory analysis for `tmaze-env2` in Figure 8.33(b) says that it requires at least three memory states. One of the best control strategies is to set the robot's internal state into the mode of turning left or right at initial perception of light. It will ignore any light configuration after the robot decides its turning action in advance. In this environment, two state controllers, as well as purely reactive controllers cannot be successful in every case. Desirable controllers need to disambiguate the situations where light is near the T-junction from those where it is near the starting position on the same side. As a result, it will require one more memory state than the `tmaze-env1` experiment.

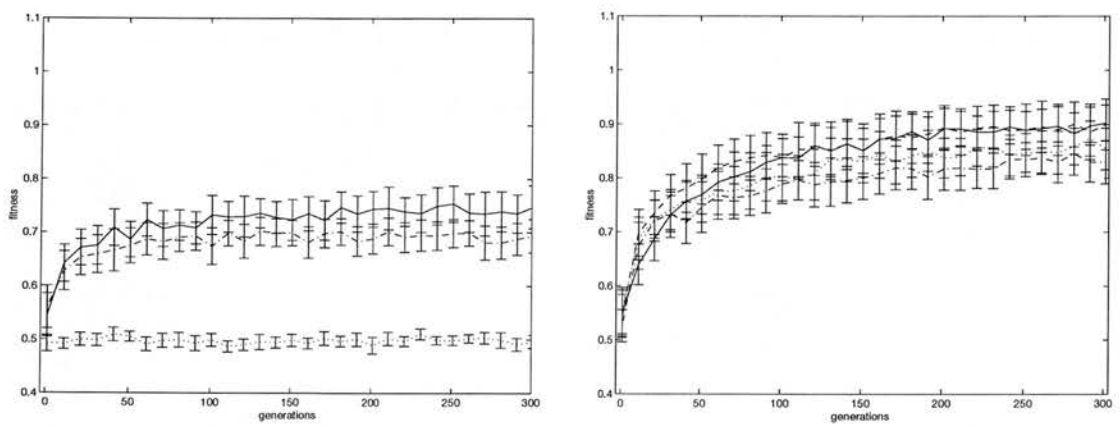
In the experiment `tmaze-env3`, the decision of turning cannot be simply determined after passing the corridor. This task is more complex than the previous two light experiments, since robots should recognize the formation of lights. When noise is involved with sensor readings, the perceptual states may become unstable, which will render the task more difficult. Thus, it is investigated whether or not noise in sensor readings can affect the memory requirement for the given task. Experiments with and without noise in sensor readings are tested with various memory states in the problem `tmaze-env3`. Figure 8.34 shows the fitness distribution with noisy sensor readings for each number of states; infrared sensors and motor actions have $\pm 10\%$ noise, and ambient light sensors have $\pm 5\%$ noise. Figure 8.35 is the result without noise

on ambient light sensors (but noise still remains on infrared sensors and motor actions). Each evolutionary experiment is repeated 25 times with population size 50 and 300 generations.



(a) (b)

Figure 8.34: Average fitness distribution over maze problem `tmaze-env3` with noise on ambient light sensors (a) one to three states (dotted: 1 state, dashdot: 2 states, solid: 3 states) (b) four to seven states (dotted: 4 states, dashdot: 5 states, dash: 6 states, solid: 7 states)



(a) (b)

Figure 8.35: Average fitness distribution over maze problem `tmaze-env3` without noise on ambient light sensors (a) one to three states (dotted: 1 state, dashdot: 2 states, solid: 3 states) (b) four to six states (dotted: 4 states, dashdot: 5 states, dash: 6 states, solid: 7 states)

The experiments evolved controllers with each number of states from one to eight. Tournament selection of group size four was used. There is a hierarchy of behavior performance with memory size in noisy environments. Purely reactive controllers are significantly worse than two or three state controllers. The difference of performance between two and three states is not as clear as in the evolutionary experiments without noise. Sensor noise makes it a harder problem and the hierarchy of behavior performance is a little different. Four or more states

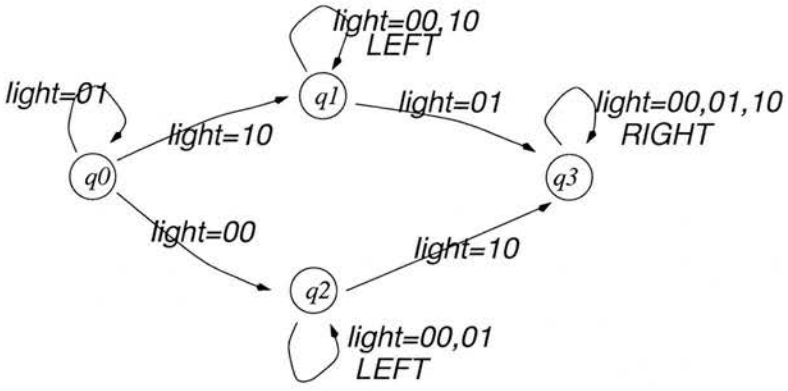


Figure 8.36: State transition diagram with four states for `tmaze-env3`

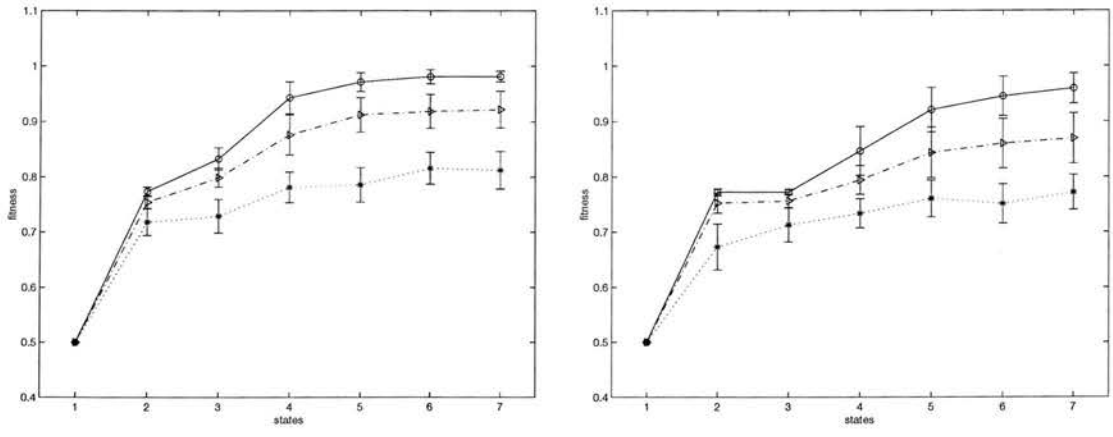
	light sensors without noise				light sensors with noise			
states	90% success		95% success		90% success		95% success	
1	0	$[7.57C, -]$	0	$[7.57C, -]$	0	$[7.57C, -]$	0	$[7.57C, -]$
2	0	$[7.57C, -]$	0	$[7.57C, -]$	0	$[7.57C, -]$	0	$[7.57C, -]$
3	0	$[7.57C, -]$	0	$[7.57C, -]$	0	$[7.57C, -]$	0	$[7.57C, -]$
4	5	$[2.54C, 11.14C]$	2	$[3.97C, 40.88C]$	5	$[2.54C, 11.54]$	5	$[2.54C, 11.54C]$
5	5	$[2.54C, 11.14C]$	3	$[3.31C, 22.96C]$	6	$[2.29C, 8.64C]$	4	$[2.87C, 15.26C]$
6	14	$[1.36C, 2.71C]$	9	$[1.80C, 4.94C]$	15	$[1.30C, 2.46C]$	13	$[1.42C, 3.00C]$
7	10	$[1.68C, 4.28C]$	6	$[1.36C, 2.71C]$	14	$[1.36C, 2.71C]$	9	$[1.80C, 4.94C]$
8	10	$[1.68C, 4.28C]$	4	$[2.87C, 15.26C]$	14	$[1.36C, 2.71C]$	11	$[1.59C, 3.76C]$

Table 8.6: Effort costs for various memory states in T-maze problem `tmaze-env3` (each column represents the number of successes and the corresponding effort cost interval)

are remarkably better than two or three state controllers in both experiments with and without noise. From the figures, the T-maze problem for `tmaze-env3` needs at least four memory states.

When controllers evolved without noise are applied to noisy environments, the success rate of prediction often becomes worse. Transient error signals caused by noise can severely influence state transition in state machines. Regardless of any noise in light sensor readings, the task can be accomplished with four memory states. One of the best evolved controllers predicted the decision with 100% success rate among 500 evaluations. This controller is shown in Figure 8.36 and noise does not severely effect its state transition. It was observed that controllers with more than four states can also handle transient errors in perceptions. Their control strategies change their internal state into a different internal state temporarily when noise creates transient error signals in light intensity, and when the signal becomes stable, the previous state is restored. Figure 8.36 is the best control strategy with an optimized number of states.

For the effort test, a success is defined as the case in which the fitness value after 300 generations is greater than 0.9 or 0.95. Fitness 0.95 means 95% or more success rate in predicting goal positions. Table 8.6 shows the effort costs estimated with the number of successes among



(a) (b)

Figure 8.37: Memory analysis over T-maze problem `tmaze-env3` (a) without noise (b) with noise (dotted: 100 generations, dashdot: 300 generations, solid: 1000 generations)

25 trial runs with population size 50. Noisy sensor readings do not degrade the performance for the success rate 90% or 95%. On the contrary, they have a little more success cases than light sensors without noise. Controllers with less than four states have no success and so the effort test says that at least four states are required to achieve the T-maze task. More than five state machines are significantly better in effort cost than less than four state machines. Four or five states are in transition.

Figure 8.37 is the result of EMO application to the environment `tmaze-env3`. It shows the performance hierarchy of memory states. Without noise, the performance of two, three and four states is significantly different. Especially there is a distinct difference between two state controllers and three state controllers. For four states or more, the fitness is stabilized as expected. In noisy environments, two and three states have similar performance. More than three memory states significantly outperform three state machines. It shows that noise can influence the memory requirement in robotic tasks. Even though there is an optimal solution with four states, more capacious memory has a tendency to improve solutions rapidly. Seven state controllers have a stable performance and they are significantly better than four states. Purely reactive controllers are only able to choose one direction. Its decision success rate was 0.5. It is almost the same as random decision in performance.

In the T-maze problem, robots need to memorize the sensory patterns experienced in the past. As a result, it requires long-term memory. The above experiment is a very simplified model that robots can experience in the real world. For example, when robots explore the environment and look around the landscape, they can memorize the landscape and its visual features to return

to their home. Some researchers have used recurrent neural networks to record such long-term memory [Yamauchi and Beer, 1994a].

This behavior is a good example of requiring sequential learning or memory. The control structure of feedforward neural networks cannot solve this problem, even though all infrared sensors and ambient light sensors on the Khepera robot are used. Recurrent neural networks with feedback neurons will be one of the alternatives to handle this problem. Even with the one light experiment, feedfoward neural networks cannot succeed, unless free movement is allowed in the approach corridor.

8.6 Summary

In this chapter, a variety of robotic tasks were tested with a realistic robot model and simulation program. The simulation experiments show how a real Khepera robot behaves, since robotic dynamics and sensor readings simulate real robot tasks very closely. The realistic robot model is comparable with simple robot model described in chapter 6. For wall following, corridor following and box pushing, all of them produced the same results in terms of memory requirement.

One of the main issues is to see the effect of internal memory in various robotic tasks, such as wall following, corridor following, box pushing and T-maze decision. Thus, the evolutionary multiobjective optimization (EMO) approach is applied for memory analysis. The fitness test and effort test are also applied appropriately. It is shown that the EMO approach is more effective than the fitness test and effort test to produce significance statistics on the performance difference of variable state machines in noisy environments. According to the memory analysis, wall following behavior and corridor following behavior require internal states to improve performance. Box pushing behavior does not need internal state, but when the box size is reduced, the effect of memory states can be seen. Especially for T-maze decision, purely reactive controllers cannot succeed in predicting correct decisions from perceptual experience. Multi-states are required to memorize perceptual states in the T-maze problem.

Experiments tested various control structures including feedforward neural networks and rule-based state machines from two sensors to six sensors in addition to finite state machines. The controllers are evolved to see memory effects depending on the number of sensors or sensor range. Evolutionary computation attempts to find the best controllers, but the evolved controllers may not be optimal. In the experiments, many runs for the same control architecture

are repeated to find the best solution controllers adapted to a given environment. When various control architectures are compared, it is assumed that the best solution controllers evolved with each control structure will represent the control structure. For each robotic task, the two best controllers for each control structure are collected and they are compared with the two best controllers for other control structures..

Evolved controllers are tested in new environments to see how robustly they can survive. Two criteria, the collision factor and fitness factor, have been considered to choose the best controllers among the various control architectures. Thus, selection of desirable controllers is based on multi-criteria, the minimum collision and the best fitness. It can be argued that the two criteria are not sufficient to choose desirable controllers and more criteria for good controllers can be added. We leave the choice of multi-criteria for future work.

For wall following behavior, a couple of states were required to distinguish perceptual states in free space and close to a wall. These two situations needed two different motor actions to escape aliases and the problem could be solved with internal states. Even though the number of sensors was increased with the state machine control structure, there was no improvement in performance. Purely reactive controllers with restricted motor actions showed circular movement when they were in free space. Neural network controllers allowed a variety of motor actions and so they could avoid the alias problem by generating motor actions depending on continuous sensor values. For instance, they could make circular movements with large radius or move forward in free space and then follow walls after they reached them. When the best controllers of various control architectures were tested in several test environments, increasing the number of sensors gave more opportunities for producing robust controllers. For wall following behavior, binarized sensor information with internal states was better in terms of collision and fitness performance than purely reactive neural networks in test environments. State machines with four sensors and two internal states showed the best performance among the various control architectures.

For corridor following behavior, the number of sensors significantly influences the performance. When two infrared sensors were used with state machines, two internal states were required to improve performance within a limited exploration time. Internal states were used to escape difficult situations effectively and make a speedy movement by sequencing motor actions. When the number of sensors was increased into four, robots could recognize the environment well and reduced collisions in test environments. Neural networks as a purely reactive controller showed the best performance among the various control architectures.

It has been shown that box pushing behavior can be achieved with purely reactive controllers. Moreover, two infrared sensors were sufficient to obtain satisfactory performance. However, memory states could make robots behave better or more intelligently in some situations. When the box size was reduced, two infrared sensors had difficulty in detecting the box. Thus, internal states were useful to design efficient controllers. If more sensors were applied to detect the small box, purely reactive controllers showed good performance. Dynamic selection in N-K sampling shows difficult sample positions by counting the number of selections for each generation. It was tested for the box pushing task and the result showed there was a difficulty in evolving controllers when robots had ambiguous situations, for example, when they were positioned in front of the box with the light in a direction either directly in front or behind them.

In the experiments, robots are placed near the box. If they are distant from the box, exploration will be needed to search for the box. There is a scalability problem, if one tries to evolve the whole complex behavior from exploration to box pushing. The exploration process can be easily obtained but it will need much time because a box can be placed at any random position in the arena.

The T-maze problem needs long-term memory to recognize the environmental situations before decisions of turning left or right. The amount of memory required for the task will depend on environmental states. If there are complicated landmarks in the environment, this increases the necessary memory amount. Several experiments were tested with lights acting as environmental landmarks. The intensity of light signals fluctuates depending on the distance between the sensors and the lights. Robots need to internalize perceptual states into memory for future decision. When one light was used for the direction of future movements, two internal states were required. When two lights were placed at four possible positions and the turning decision relied on the formation of lights, a different size of memory states from two to four were required. If more lights are used at various positions, more memory states will be required to record various environmental situations. The T-maze problem is a problem that purely reactive controllers cannot handle. Normally memoryless controllers even with feedforward neural networks have the same performance as random controllers. The problem cannot be solved with purely reactive controllers regardless of the number of sensors or continuous sensor range. Memory is the only solution to handle the problem. Continuous sensor range or more sensors can sometimes decrease the memory requirement, but in some tasks such as T-maze decision, there is a limitation of performance regardless of the expansion of sensor range.

In this chapter, we discussed how to quantify internal states required for desirable robot behav-

iors. Boolean logic networks with varying dynamic elements were sufficient to achieve various robot behaviors. They belong to the class of regular languages, which have a finite memory. A complex behavior, called homing navigation, will be demonstrated as a task difficult to evolve with state machines in the next chapter.

Chapter 9

Conclusion

This thesis studied the effects of memory on autonomous agent behaviors. Memory is a tool for temporal information processing in complex robotic tasks and also it can play a role in solving perceptual aliases that occur in many agent problems.

Memory is very broadly studied in many fields including psychology and neurobiology. The concept of memory may be vague depending on application fields. In the thesis, memory was mainly defined as a set of internal states. It can be seen as a short-term memory to record perceptions, actions or their situation states. To keep a simple memory structure, the internal states are represented as register-bits. Success of reactive controllers in behavior-based robotics implies that robot controllers may not necessarily need complex memory. Thus, purely reactive controllers and reactive controllers with small-sized memory are compared in the thesis.

The problem set was largely divided into two parts, grid world problems and robotic tasks. In the grid world problems, agents were supposed to move on the grid with a restricted number of motor actions. Three problems, the artificial ant problem, the Tartarus problem and the woods problem, were tackled with memory analysis. Memory plays a role in improving performance and designing successful controllers. In robotic tasks, various behaviors, including wall following, exploration, obstacle avoidance, corridor following and T-maze decisions, were tested with two kinds of sensor modeling and robot kinematics simulation. The sensors were modeled with whisker-like binary sensors and with realistic simulation. The above problems are related with the following issues:

- How to decide evolutionary parameters
- How to model sensors

- How to design memory-based controllers
- How to analyze memory requirement with evolutionary computation
- How to handle many sample positions in evolutionary computation
- How to design robust controllers
- How to evaluate controllers in test environments

9.1 Summary of Methods

To analyze memory effects on robotic behaviors, several issues relating to evolutionary algorithms are covered. A new evolutionary computation method and various control architectures are investigated.

9.1.1 Performance Issues

Evolutionary computation is stochastic and one single experimental run is not sufficient to represent the performance of robot controllers. By sampling theory, each experiment can be regarded as a sample run. Using a fitness test, the average fitness distributions of many experimental runs are observed to decide desirable evolutionary parameters and controller structures. The confidence intervals are estimated with a t-test. Also an effort test of how much effort is spent to obtain desirable performance is developed to measure the confidence intervals of the effort cost. It is based on the Beta distribution over the number of success cases among multiple runs. Two tests, the fitness test and the effort test, have been useful methods to determine desirable evolutionary parameters. In the artificial ant problem, these two tests were applied to determine population size, the number of generations, exploration time, mutation rate and selection mechanism. Furthermore, they have been applied to decide how many memory elements are required to achieve a given task.

To analyze memory requirement for tasks, state machines were used as a control structure. Memory analysis with two tests needs many experimental runs for each control configuration. Thus, a novel method of evolving various control structures together in a population is developed. Each population maintains variable state machines to compete each other. It involves new genetic operators over variable-length chromosomes and various solution architectures are handled with Pareto-optimization. Ultimately it can produce the best performance for each

number of states and it can see how many memory states are required to achieve desirable performance.

9.1.2 Robotic Models

In the thesis, robot models are divided into simple models and realistic models. Simple robot models use whisker-like infrared sensors and robot kinematics equations, while realistic simulation models assume cone-bearings for each infrared sensor and take many sensor points to simulate real Khepera robot sensors. The former is very different from reality, but the robotic performance is quite similar to the latter model. In spite of its simplicity, noise-sensitive controllers are easily evolved with the simple robot model. Realistic simulation models employ random noise on sensor readings and motor actions. This helps to design controllers which are transferable to physical robots. The model of noisy sensors and motors is especially important to produce robust controllers. On the other hand, noisy signals make it more difficult for an evolutionary approach to find desirable controllers.

9.1.3 Sample Selection

In robotic tasks, robots can be positioned in various environmental situations. Evolving controllers at many positions is useful to generate robust controllers. However, more positions requires more computation time. An efficient technique, called the N-K sample method [Gathercole, 1998; Lee, 1998], is used to overcome this difficulty and produce desirable controllers efficiently. It selects K samples among N possible positions and tries to evolve on the K sample positions every generation. As a result, it can reduce the computation time to almost the rate $\frac{K}{N}$. Two sample selection methods, random selection and dynamic selection, are applied to handle various problems. Random selection chooses samples randomly and it gives a uniform selection probability to each sample position. Dynamic selection chooses samples depending on the difficulty of sample positions; more difficult positions have more selection probability and the difficulty is measured by fitness.

In noisy environments, one evaluation of a chromosome for fitness was not sufficient to obtain robust controllers, since it tends to create much fluctuation of fitness curves with noise-sensitive controllers. Thus, several evaluations over N positions is applied to a genome controller and then the average or the worst score is considered as its fitness. It was observed that in noisy environments the elitism strategy significantly influences evolutionary performance. The best

chromosome with multiple evaluations over N positions is chosen as the elite and every generation it is inserted into a new population. For the evolutionary multiobjective optimization (EMO) approach, a similar elitism process is applied to the best chromosomes for each number of states. The EMO approach keeps a pool of the best genomes for each state and one of the pool members is randomly chosen for elitism.

9.1.4 State Machines

To measure the effects of memory, control structures are based on finite state machines. They can represent internal states very effectively and they are equivalent to Boolean logic networks with memory. It is easy to quantify memory amount by enumerating the number of states in the machines. Variations of state machines are developed to handle several sensors and complex sensor readings. Finite state machines have limitations of representation for many sensor states. To cover many sensors and motors, a rule-based structure is suggested. Controllers have a set of rules and each rule consists of sensor conditions and actions and is defined on its own internal state. The chromosome size for a rule-based structure is proportional to the number of sensors, while the genome size for finite state machines is exponentially proportional to the number of sensors. Thus, the rule-based structure is useful to robots with many sensors and several memory states. The finite state machines and rule-based state machines are evolved with genetic algorithms.

Another control format with a tree structure is called a tree state machine, which is useful to handle continuous sensor ranges. It has the form of a decision tree. Leaf nodes include motor actions and the next state transitions, and internal nodes in a tree defines decision space with a sensor and its threshold. As a result, traversal from root node to a leaf node will build a sensor state, and motor actions and state transitions are specified on the sensor state. Tree state machines have multiple trees and a decision tree is defined for each state. Thus, only one of multiple trees is evaluated when its internal state is triggered. It also allows a different set of sensor states for each internal state.

9.1.5 Test Environments

When controllers are evolved in a given environment, they are tested in a few other environments to see their robustness. In many cases, evolved controllers only show good performance in the environment in which they were evolved, and when they are tested in other environments,

the performance degrades.

In robotic experiments, various control architectures were evolved and evaluated in test environments. This shows what control architectures can survive various environments. In test environments, two criteria, a collision factor and a fitness factor, were considered to choose the best controllers. Other criteria can be added when fitness cannot cover the characteristics of desirable controllers properly. These are therefore multi-criterion decision problems. In the experiments, the best control architecture were chosen to have the minimal collisions and the best fitness.

9.2 Summary of Results

Two kinds of tasks, grid world tasks and robotic tasks, are tested to see the effect of memory in autonomous agent behaviors. It is analyzed together with sensor and action space.

9.2.1 Grid World Problems

Grid world problems have been studied by many researchers. Various memory-based control structures have been used to improve performance, but the complexity issues of the problems have been neglected. In this thesis, three kinds of grid world problems, the artificial ant problem, the Tartarus problem and the woods problems, were handled to show how many memory elements are required to achieve desirable performance.

The artificial ant problem involves following food trails on the grid. Ant agents are supposed to pick up as many food pellets as possible. They have only one sensor and they had to decide their motor actions depending on their internal states. The EMO result says the task requires at least five memory states and increasing memory states monotonically improves the performance of how early they can collect all the food.

The Tartarus problem is a box pushing task to push six blocks to walls. It has many board configurations and its randomness makes the problem more difficult, which required N-K sampling in evolutionary computation. Finite state machines with three sensors and tree state machines with eight sensors were evolved and they showed similar performance. The problem required at least six states for satisfactory performance.

The woods problem is a goal search problem to find a route from any empty space to the goal position, only depending on perceptions with eight sensors. Woods101, woods102, maze10,

gridworld and woods14 were tested with a different number of sensors. The problem shows how to handle perceptual aliases with memory. When the number of sensors is decreased, more perceptual aliases are generated. However, it is shown that more perceptive aliases do not necessarily increase the memory amount required to solve the problems. Woods101, woods102 and maze10 need two internal states to solve problems. Sutton's gridworld requires no internal state with eight sensors, but two internal states with four sensors. Woods14 is a Markovian environment with eight sensors, but four-sensor agents require four states and agents with two sensors, left and right, requires six states. If a front sensor is included for two-sensor agents, four states are required. A single front sensor enables agents to solve the problem with four states.

9.2.2 Robotic Tasks

The robotic tasks involve more dynamic factors than grid world problems. Dynamic movements of robots can escape perceptual aliases effectively in many cases, using a variety of motor actions. Sensor readings also provide various perceptions, unlike grid world problems. The sensorimotor mapping can have dynamic attractors to avoid being stuck at difficult situations. In spite of such reactive properties of robots, difficult tasks still exist. The following robotic tasks from primitive behaviors to moderately complex behaviors were tested to see memory effects in reactive control systems.

Obstacle avoidance and exploration behavior were tested with simple robot models. Those primitive behaviors are intrinsically purely reactive. When only two sensors were used for obstacle avoidance, robots did not move forward confidently and they had a tendency of moving around objects to escape possible collisions. When two more sensors were added in front, they could have more confident forward movement. For exploration with cylinder-shaped obstacles, two-sensor robots showed a safe strategy similar to wall following, since it can cover much of the arena's area; fitness was assigned as visiting all areas uniformly with an entropy calculation. Increasing the number of sensors encourages robots to move forward.

Wall following behavior was evolved with 30 random positions in realistic simulation. Purely reactive controllers with two or four sensors failed to follow walls at 10 positions and showed circular movement in free space. Two internal states were quite effective to avoid staggering in open space. Feedforward neural networks could solve the problem by processing continuous sensor values. However, four sensors with internal states showed the best performance among the various control architectures. Increasing the number of sensors could reduce the number of

collisions in test environments.

Corridor following involves environments with narrow corridors. It is easy for robots to collide with walls. Purely reactive controllers had difficulty in finding the exit at the end of narrow corridors, while memory-based controllers utilize their internal states efficiently to sequentialize motor actions and speed up movements. However, binarized four sensors or two-sensor neural networks could easily solve this problem by recognizing the situation better. Four-sensor neural networks showed the best performance in test environments for corridor following.

Box pushing looks like a complex task, but it was achieved with purely reactive controllers. Binarized sensor information was sufficient for this behavior. When a small box with almost half the radius of the robot was used, purely reactive controllers with two sensors were worse than memory-based controllers. When two more sensors were added in front, memoryless controllers could push the box efficiently. Neural networks showed good performance even with a small box.

T-maze decision is an example on which memoryless controllers show their limitations. Purely reactive controllers can achieve the same performance as random controllers do. Their turning decision takes only one direction regardless of perception history. In the experiment with one light, two internal states were required to remember which side it is on. Two light experiments had three possible formations where each light can be placed on either side near the T-junction or far from it. The first formation was to detect lights close to T-junction and it requires two internal states. In the second formation, robots were supposed to detect lights far away from the T-junction. In this case robots used three internal states to remember their decision made at initial perception but which could be confused with a change of light intensity. The last formation requires sequencing perceptions. Robots should memorize the formation of a light in series. As a result, it required four states.

The desert ants *Cataglyphis* show particular and elegant homing navigation behavior, returning to their nest with a direct path after exploring their environments. Homing navigation behavior with path integration requires innumerable memory states to record a history of motor actions and perceptions. The behavior need roughly 100 recurrent neurons in a set of cellular arrays with a light compass. Its complexity is beyond the power of purely reactive systems or even state machines. The complexity of control systems reflects the behavior complexity. Homing navigation behavior is one of the most difficult robotic tasks, and it is extremely hard to evolve with state machine controllers.

Table 9.1 summarizes the results of memory analysis for grid world tasks and robotic tasks.

environments	task	sensor	actions	memory size
grid world	Santa Fe trail	1 sensor	4 actions	5
	John Muir trail	1 sensor	4 actions	5
	the Tartarus problem	3 sensors	3 actions	6
	the Tartarus problem	8 sensors	3 actions	6
	woods101	4 sensors	4 actions	2
	woods101	8 sensors	4 actions	2
	woods102	4 sensors	4 actions	2
	maze10	4 sensors	4 actions	2
	gridworld	8 sensors	4 actions	1
	gridworld	4 sensors	4 actions	2
	woods14	8 sensors	8 actions	1
	woods14	4 sensors	8 actions	4
	woods14	2 sensors (S_f, S_r)	8 actions	4
	woods14	2 sensors (S_f, S_l)	8 actions	4
	woods14	2 sensors (S_l, S_r)	8 actions	6
	woods14	1 sensors (S_f)	8 actions	4
robotic world	obstacle avoidance	2 sensors	8 actions for each wheel	1
	obstacle avoidance	4 sensors	8 actions for each wheel	1
	wall following	2 sensors	8 actions for each wheel	2
	wall following	4 sensors	8 actions for each wheel	2
	wall following	2 sensors, neural	continuous integer values for each wheel	1
	wall following	4 sensors, neural	continuous integer values for each wheel	1
	wall following	6 sensors, neural	continuous integer values for each wheel	1
	exploration	2 sensors	8 actions for each wheel	1
	exploration	4 sensors	8 actions for each wheel	1
	box pushing	2 sensors	8 actions for each wheel	1
	small box pushing	2 sensors	8 actions for each wheel	2
	small box pushing	4 sensors	8 actions for each wheel	1
	corridor following	2 sensors	8 actions for each wheel	2
	corridor following	4 sensors	8 actions for each wheel	1
	corridor following	2 sensors, neural	continuous integer values for each wheel	1
	corridor following	4 sensors, neural	continuous integer values for each wheel	1
	T-maze, 1 light	4 sensors	8 actions for each wheel	1
	T-maze, 2 lights, `env1`	4 sensors	8 actions for each wheel	2
	T-maze, 2 lights, `env2`	4 sensors	8 actions for each wheel	3
	T-maze, 2 lights, `env3`	4 sensors	8 actions for each wheel	4

Table 9.1: Memory analysis results for each task

Autonomous agents have a tendency of requiring more memory elements when sensors or motor actions are restricted. Thus, the effect of memory should be considered together with sensor and motor action space.

9.2.3 Sensors, Actions and Memory

The performance function G can be defined on a set of sensors, a set of motor actions[1] and the size of memory.

$$G(S,A,M) = G(\{s_0, s_1, ..., s_p\}, \{a_0, a_1, ..., a_q\}, M)$$

[1] A set of motors as well as a set of actions for each motor can be considered, but it is assumed that a set of motors is pre-defined.

where S is a set of sensors, A is a set of motor actions, and the memory size M is the number of memory states. The performance G is partially ordered on the set of sensors and the set of motor actions and memory size.

$$G(S_1,A_1,M_1) \leq G(S_2,A_2,M_2)$$

where $S_1 \subseteq S_2$, $A_1 \subseteq A_2$, $M_1 \leq M_2$. When a set of sensors S_1 is a subset of S_2 and controllers for S_2 use only elements which belong to the set S_1, the optimal performance of the controllers for S_2 is the same as that for S_1. Thus, more sensor space has a potential of better performance. Similarly more motor action space or larger memory size can improve performance.

The above partial order relations will lead to a lattice[2] structure on sensors, motors and memory. In other words, a hierarchy of performance function can be built on the power set of a set of sensors and also on the power set of a set of motor actions. Since the power set has a lattice structure, it has the least upper bound for performance. This means there exists an optimal control structure for desirable behavior performance in robotic problems. For a given set of motor actions with fixed size of memory, a different set of sensors will have a hierarchy of performance with a lattice structure. Similarly, for given sensors and memory, a set of motor actions will build a hierarchy of performance as a lattice. One of the main concerning issues is that memory can involve a hierarchy of behavior performance. It holds a partial ordering relationship over the number of internal states for agent problems.

For a given set of sensors, a different size of memory states will have a hierarchy of performance with linear ordering. From this fact, the EMO approach was developed to show behavior performance for each size of memory states. The experimental results were demonstrated in grid world problems and robotic tasks. We argued that there is a significant gap between memoryless controller and memory-based controller. The lattice structure illustrates the difference. From the above partial order relation, limiting sensors increases the possibility of requiring more memory to achieve a given task, since restricting sensors increases the probability of decreased performance.

In this thesis several robotic behaviors were tested to see the memory effects. In conclusion, memory is an important element to improve performance when sensors or motor actions are restricted, or to record and process temporal sensor information through a history of perceptions.

[2] A lattice is a partially ordered set in which all nonempty finite subsets have a least upper bound and a greatest lower bound.

9.3 Discussion

The current sensor modeling with many sensor points is inefficient when many sensor readings are required. A combination model between realistic sensor simulation and the tabular method is proposed for future work. It will reduce the effort to catch sensor values for every object and save much time in calculating many sensor points during the evolutionary process.

The effort test counts the number of success cases among multiple runs. To define a success, desirable performance should be determined in advance as a boundary limit of fitness. If fitness is over the limit, then it is regarded as a success. In the experiments, desirable performance was specified by experience, without any definite rule. When we are interested in comparing two controllers, the middle point of the best average fitnesses for two controllers can be a candidate boundary limit. However, based on the Beta distribution of the effort test, a more theoretical foundation should be studied to compare two fitness distributions.

EMO takes a relatively long time to reach a clear difference on variable state machines, since it evolves a population of variable state machines together. Special genetic operators were suggested to obtain solutions concurrently, but rigorous evaluation should be explored to see the effect of each genetic operator and possibly propose new genetic operators which can accelerate evolutionary search. Tree state machines were suggested to cover continuous range sensors as well as memory representation. In the thesis, tree state machines were not linked with the EMO approach. For future study, the EMO approach on tree state machines should be developed. The method will be useful in many applications to quantify memory amount with real sensor values.

Single agent and single robot tasks have been handled in the thesis. The memory analysis can be extended to multi-robot experiments. Possibly multi-robot behaviors may require less memory through interactions among robots. Social behaviors of many insects show each agent has a simple and reactive strategy. For instance, swarm of social ants builds networks of regular traffic using a process of pheromone, laying and following. Their strategies reactively depend on their perceptions from the environment. Future study will illuminate the memory effects in multi-robot behaviors such as grouping, cooperation and other social behaviors.

Blocks world problems have been solved with purely reactive systems [Baum, 1999]. Baum used genetic programming with Automatically Defined Functions (ADF) to solve the problem. As a result, the control system learned to solve large Blocks World problems with no hand-crafted features. Planning is normally modeled with a series of actions or a series of subtasks. Thus, the planning process can be possibly evolved with memory. A history of world states

or internal memory can be incorporated to improve solutions. We are interested in evolving state machines to see whether they can implement an efficient planning process, and whether memory-based structure can improve performance.

So far the memory capacity for evolved robotic tasks is relatively small compared with brain systems. For example, recurrent neural networks can include a number of states with many kinds of attractors. Biological animals use various styles of memory for their behaviors. The homing navigation of desert ants is believed to involve the combination of two schemes, landmark-based navigation and path integration. Homing navigation with path integration can be simulated with recurrent neural networks with the format of a circular array of neurons [Kim and Hallam, 2000]. Honey bees communicate with other bees using a waggling dance. The waggling dance implies that bees keeps the information of localizing resource location and they use their internal memory to record resource location [von Frisch, 1967]. *Aplysia* shows habituation and sensitization of neurons in response to external stimuli. The phenomena involve memory states, since a history of perceptions is memorized in a set of neurons. Pavlov's conditional reaction is also related to memory-based systems. Some people modeled the process with several recurrent neurons.

Memory is an important tool for biological animals to survive in the real world. The current state machine structure is useful to quantify memory amount, but it has limitations with representing the complex processes that recurrent neurons can simulate. In some respects, it is believed that recurrent neurons have more power to represent various memory structures. According to Chomsky's language hierarchy [Hopcroft and Ullman, 1979], controllers can have a language hierarchy of regular language, context-free language, context-sensitive language, and Turing computable language. From this aspect of language hierarchy, robot behaviors can be analyzed based on the complexity of the controller language. The study will elucidate a greater diversity of robot behaviors. Also we can explore what control structures will be useful to achieve biological behaviors.

9.4 Future Work

In many cases, recurrent neural networks are desirable control structure, especially for complex behaviors. Yet we still do not know the process of their state transitions. We need to investigate more about the possibility of the minimal state machines equivalent to the recurrent neural networks.

In many evolutionary approaches, the best genome controllers are evolved to be adapted to given environments. As shown in the artificial ant problem, desirable controllers tend to memorize the environment. However, controllers with a large number of memory elements may be fragile when noise is involved or even when only a small part of environments are changed. Thus, only a small amount of memory can be more effective and robust in adapting agents to various environments for the same style of task. The relationship between problem space and the amount of memory can be investigated for robust controllers. Presumably more agent environments may require less internal memory to achieve the given task.

For some robotic tasks, controllers with a few states and binarized sensors do almost as well as feedforward neural network controllers with continuous sensor ranges. Single-threshold state machines has been applied so far and multi-threshold state machines are possible solutions to extend sensor states. The ability of multiple thresholds on state machines can be studied over a variety of robotic tasks for future work. Tree state machines, which allow variable multi-thresholds and a variety of sensor states, may have a potential of achieving robotic tasks with less amount of memory than finite state machines with binarized sensors. We will study the relationship between internal memory and sensor states, especially when the sensor states are expanded with multi-thresholds on each sensor.

Instead of using two sensor states for one sensor, three or more sensor states can be created by allowing fixed multi-thresholds for each sensor. I believe that it is worth while to compare fixed multi-thresholds with variable multi-thresholds. The performance measure, such as fitness test and effort test, will be able to be useful tools to analyze those techniques. If we allow fine-scaled multi-thresholds, some sensor states can be severely influenced by noise. I think that there is a correlation between multi-thresholds and noise level; more fine-scaled thresholds may be sensitive to higher noise level. I leave the study for future work.

Classifier systems with internal memory have been studied on perceptual aliasing. Pure evolutionary computation and classifier systems with reinforcement learning can be investigated together to see the role of reinforcement learning on the same domain problem. This study can clarify the role of reinforcement learning in classifier systems.

Appendix A

Beta distribution

A.1 Bayes Estimation

Assume that $n = \alpha + \beta$ independent experiments have α successes and β failures. The probability distribution of success rate p can be obtained with Bayesian estimation. It is different from the maximum likelihood estimation of $p = \frac{\alpha}{\alpha+\beta}$.

Suppose that $X_1, X_2, ..., X_n$ are independent Bernoulli random variables to represent success or failure. The estimation of the probability p over the experimental data $X_1 = x_1$, $X_2 = x_2$, ..., $X_n = x_n$ will follow the posterior distribution $f(p|[x_1, x_2, ..., x_n])$.

According to the Bernoulli property, each random variable $X_i = x_i$, $i = 1, ..., n$ has a probability density function given by

$$f(x_i|p) = p^{x_i}(1-p)^{1-x_i}, \quad x_i = 0, 1$$

where p is unknown. Thus, we assume that p follows a uniform distribution as a prior condition and is equally likely to be any value (0,1). Then the prior density function $g(p)$ can be defined by

$$g(p) = c, \qquad 0 < p < 1$$

where c is a constant.

If we apply the Bayesian theorem to the probability distribution, the probability density func-

tion of p can be estimated as follows[Ross, 2000]:

$$\begin{aligned} f(p|[x_1,x_2,...,x_n]) &= \frac{f([x_1,x_2,...,x_n],p)}{f(x_1,x_2,...,x_n)} \\ &= \frac{f([x_1,x_2,...,x_n]|p)\cdot g(p)}{\int_0^1 f([x_1,x_2,...,x_n]|p)\cdot g(p)dp} \\ &= \frac{p^{\sum_{i=1}^n x_i}(1-p)^{n-\sum_{i=1}^n x_i}}{\int_0^1 p^{\sum_{i=1}^n x_i}(1-p)^{n-\sum_{i=1}^n x_i}dp} \\ &= \frac{p^{\alpha}(1-p)^{\beta}}{\int_0^1 p^{\alpha}(1-p)^{\beta}dp} \\ &= \frac{1}{B(\alpha+1,\beta+1)}p^{\alpha}(1-p)^{\beta} \end{aligned}$$

where $\alpha = \sum_{i=1}^n x_i$ is the number of successes, $\beta = n-\alpha$ is the number of failures and Beta function B is defined by

$$\begin{aligned} B(\alpha+1,\beta+1) &= \int_0^1 p^{\alpha}(1-p)^{\beta}dp \\ &= \frac{\Gamma(\alpha+1)\cdot\Gamma(\beta+1)}{\Gamma(\alpha+\beta+2)} = \frac{\alpha!\beta!}{(\alpha+\beta+1)!} \end{aligned}$$

with $\Gamma(n+1) = n\Gamma(n)$.

Hence, the Bayes estimation of the success probability p has the following density function:

$$f(p,\alpha,\beta) = \frac{1}{B(\alpha+1,\beta+1)}p^{\alpha}(1-p)^{\beta}$$

Then the mean success rate is

$$\begin{aligned} E(p|[x_1,x_2,...,x_n]) &= \int p\cdot f(p|[x_1,x_2,...,x_n])dp \\ &= \frac{\alpha+1}{\alpha+\beta+2} \end{aligned}$$

Appendix B

Santa Fe trail experiments

B.1 Tables

- Table B.1 - Santa Fe trail result with various exploration times
- Table B.2 - Santa Fe trail result with various population sizes
- Table B.3 - Santa Fe trail result with various selection mechanisms
- Table B.4 - John Muir trail result with various exploration times
- Table B.5 - Artificial ant trail results with 200 time steps

		Generations						
exploration time	time limit	≤ 500	≤ 1000	≤ 2000	≤ 3000	≤ 4000	≤ 5000	≤ 10000
300	300	0	3	7	13	18	18	23
	278	0	0	1	4	6	15	22
	250	0	0	0	0	1	2	15
	240	0	0	0	0	0	0	10
	230	0	0	0	0	0	0	4
	220	0	0	0	0	0	0	0
400	400	13	21	22	23	23	24	24
	300	0	4	14	18	19	21	24
	278	0	0	4	8	14	18	23
	250	0	0	0	2	3	3	12
	240	0	0	0	0	1	2	7
	230	0	0	0	0	0	1	3
	220	0	0	0	0	0	0	0
500	500	17	23	24	24	24	25	25
	400	14	21	24	24	24	25	25
	300	0	2	10	18	20	22	25
	278	0	0	4	10	15	17	22
	250	0	0	0	2	4	7	11
	240	0	0	0	1	2	2	10
	230	0	0	0	0	0	0	5
	220	0	0	0	0	0	0	2
600	600	23	25	25	25	25	25	25
	500	22	24	25	25	25	25	25
	400	16	24	25	25	25	25	25
	300	0	3	12	15	20	22	24
	278	0	0	2	9	13	16	22
	250	0	0	0	0	0	3	10
	240	0	0	0	0	0	0	7
	230	0	0	0	0	0	0	3
	220	0	0	0	0	0	0	1
700	700	22	25	25	25	25	25	25
	600	22	24	25	25	25	25	25
	500	22	24	25	25	25	25	25
	400	17	24	25	25	25	25	25
	300	0	2	10	19	24	24	25
	278	0	0	5	12	17	17	22
	250	0	0	0	1	3	7	12
	240	0	0	0	1	2	4	10
	230	0	0	0	1	1	2	4
	220	0	0	0	0	0	0	0
800	800	22	25	25	25	25	25	25
	700	22	25	25	25	25	25	25
	600	22	24	25	25	25	25	25
	500	20	24	25	25	25	25	25
	400	17	23	25	25	25	25	25
	300	1	4	13	22	23	23	25
	278	0	1	6	15	20	23	24
	250	0	0	1	1	4	10	20
	240	0	0	0	1	2	3	18
	230	0	0	0	0	0	1	9
	220	0	0	0	0	0	1	3

Table B.1: Santa Fe trail result with various exploration times (each number represents the number of success cases among 25 experiments)

		Generations						
population	time limit	≤ 500	≤ 1000	≤ 2000	≤ 3000	≤ 4000	≤ 5000	≤ 10000
50	400	10	17	20	21	22	23	24
	300	0	0	2	6	11	13	23
	278	0	0	1	2	3	8	17
	250	0	0	1	1	1	1	3
	240	0	0	1	1	1	1	1
	230	0	0	0	0	0	0	0
	220	0	0	0	0	0	0	0
100	400	13	21	22	23	23	24	24
	300	0	4	14	18	19	21	24
	278	0	0	4	8	14	18	23
	250	0	0	0	2	3	3	12
	240	0	0	0	0	1	2	7
	230	0	0	0	0	0	1	3
	220	0	0	0	0	0	0	0
150	400	24	25	25	25	25	25	25
	300	1	11	25	25	25	25	25
	278	0	3	20	20	22	22	24
	250	0	0	3	9	11	13	19
	240	0	0	1	4	7	9	15
	230	0	0	1	4	7	9	11
	220	0	0	0	0	2	3	5
200	400	23	24	25	25	25	25	25
	300	0	10	21	24	24	24	25
	278	0	2	13	22	22	24	25
	250	0	0	3	5	8	10	23
	240	0	0	1	2	4	6	15
	230	0	0	1	1	2	2	12
	220	0	0	0	0	0	0	3

Table B.2: Santa Fe trail problem result with various population sizes

			Generations						
selection	mutation	time limit	≤ 500	≤ 1000	≤ 2000	≤ 3000	≤ 4000	≤ 5000	≤ 10000
tournament	0.03	278	7	12	17	17	20	22	25
		210	0	0	3	7	8	10	14
		200	0	0	0	3	3	5	8
tournament	0.1	278	0	0	6	12	17	20	25
		210	0	0	0	0	0	0	0
		200	0	0	0	0	0	0	0
fitness	0.01	278	1	3	10	15	18	18	21
		210	0	1	1	2	3	4	8
		200	0	0	1	1	1	2	3
fitness	0.03	278	1	5	8	15	16	17	20
		210	0	1	1	1	1	1	2
		200	0	0	1	1	1	1	1
fitness	0.1	278	0	3	4	10	14	14	19
		210	0	0	0	0	0	0	1
rank-based	0.1	278	0	0	0	0	0	0	0
rank-based	0.03	278	0	0	0	0	0	0	0

Table B.3: Santa Fe trail problem result with various selection mechanisms (population size : 100, exploration time : 400)

		Generations						
exploration time	time limit	≤ 500	≤ 1000	≤ 2000	≤ 3000	≤ 4000	≤ 5000	≤ 10000
300	300	22	23	25	25	25	25	25
	250	16	21	25	25	25	25	25
	230	0	9	20	20	24	24	25
	220	0	1	8	11	24	24	25
	210	0	0	1	2	4	7	13
	200	0	0	0	0	0	1	3
400	300	22	25	25	25	25	25	25
	250	13	24	25	25	25	25	25
	230	1	9	17	21	24	25	25
	220	0	1	7	15	18	21	25
	210	0	0	0	0	3	5	10
	200	0	0	0	0	0	1	2
500	300	24	25	25	25	25	25	25
	250	11	24	25	25	25	25	25
	230	0	6	17	24	25	25	25
	220	0	0	5	13	17	18	25
	210	0	0	1	3	4	5	13
	200	0	0	0	1	1	1	5
600	300	22	24	25	25	25	25	25
	250	14	23	25	25	25	25	25
	230	0	9	15	21	22	23	25
	220	0	3	5	14	16	20	24
	210	0	0	0	3	3	8	14
	200	0	0	0	1	1	1	5
700	300	21	23	25	25	25	25	25
	250	12	20	22	23	24	24	25
	230	1	7	16	20	20	21	25
	220	0	1	7	14	18	19	24
	210	0	0	0	1	4	4	14
	200	0	0	0	0	1	1	4

Table B.4: John Muir trail result with various exploration times; mutation rate 0.1 and population size 100 (success : an ant agent collects all food within time limit)

		Generations						
Trail	Food	≤ 500	≤ 1000	≤ 2000	≤ 3000	≤ 4000	≤ 5000	≤ 10000
Santa Fe	70	1	6	12	16	17	18	22
	75	0	0	3	9	10	12	17
	80	0	0	0	1	3	3	9
	85	0	0	0	0	0	0	1
John Muir	70	25	25	25	25	25	25	25
	75	25	25	25	25	25	25	25
	80	23	25	25	25	25	25	25
	81	22	23	25	25	25	25	25
	85	0	5	11	18	21	22	24
	87	0	1	1	1	1	2	7
	89	0	0	0	0	0	0	1

Table B.5: Artificial ant trail results with 200 time steps

B.2 Figures

- Figure B.1 - Santa Fe trail experiments with various numbers of memory states
- Figure B.2 - Distribution of the best controllers with N-state elitism

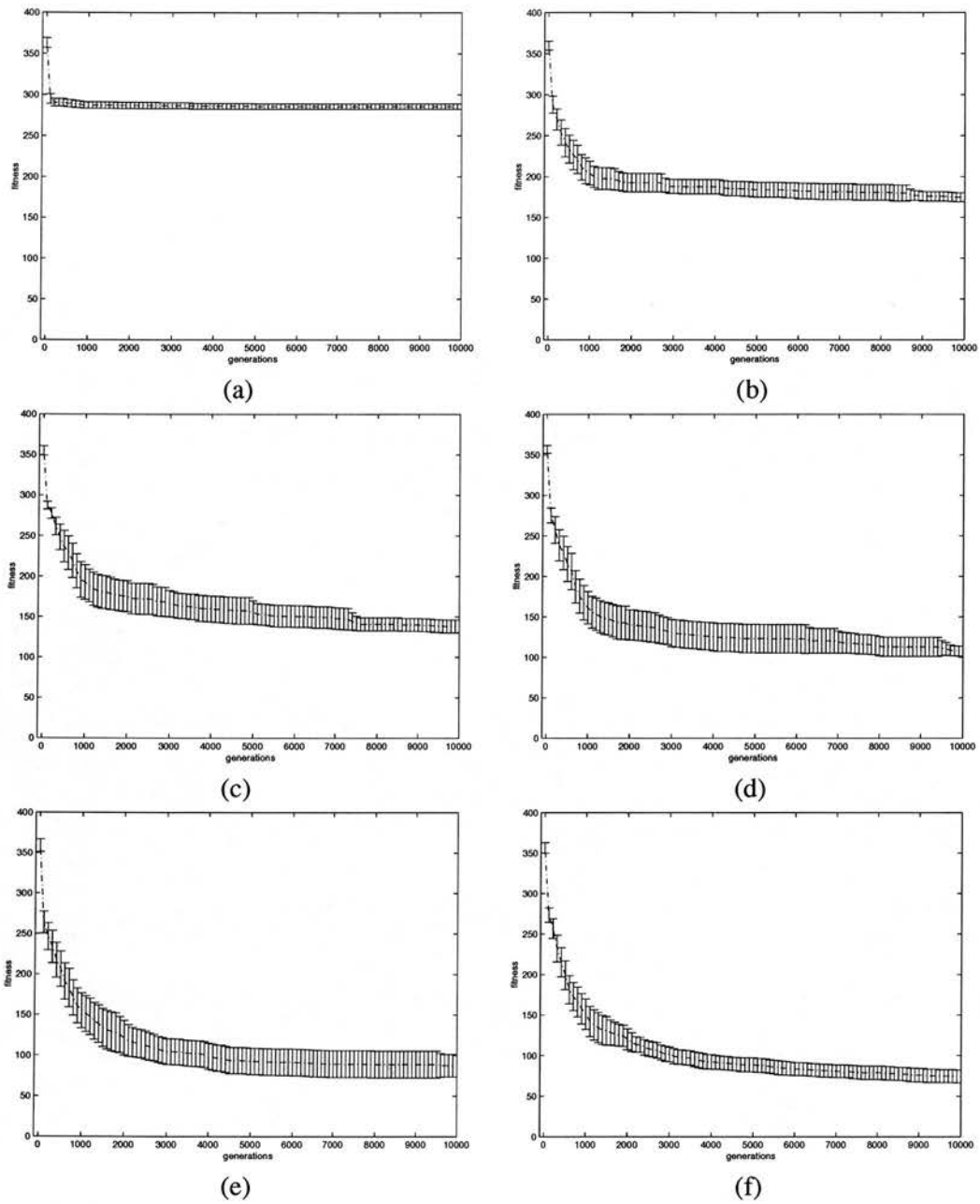


Figure B.1: Santa Fe trail experiments with various numbers of memory states (a) 4 states (b) 5 states (c) 6 states (d) 7 states (e) 8 states (f) 9 states (g) 10 states (h) 11 states (i) 12 states (j) 13 states (k) 14 states (l) 15 states (m) 18 states (n) 20 states

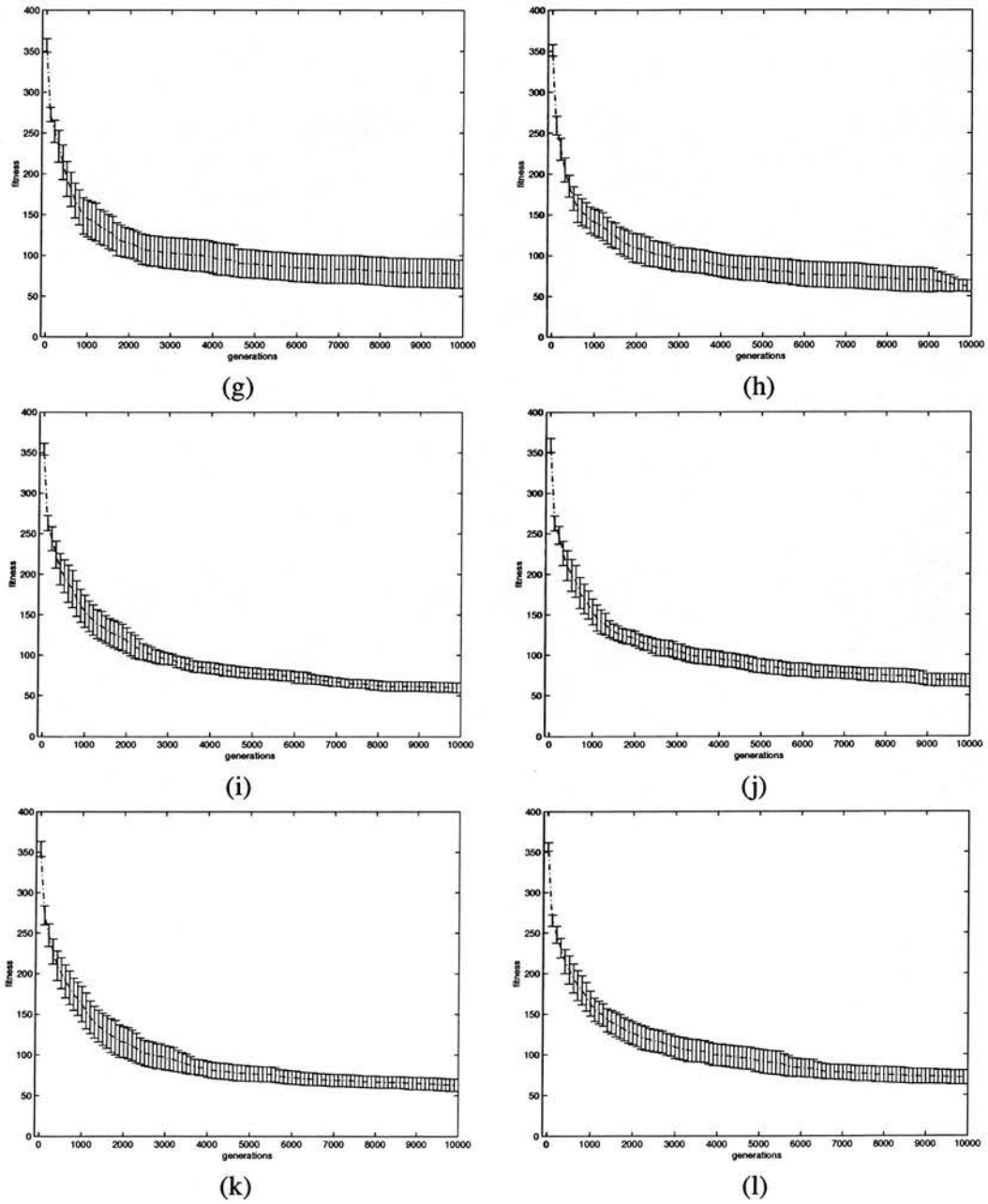
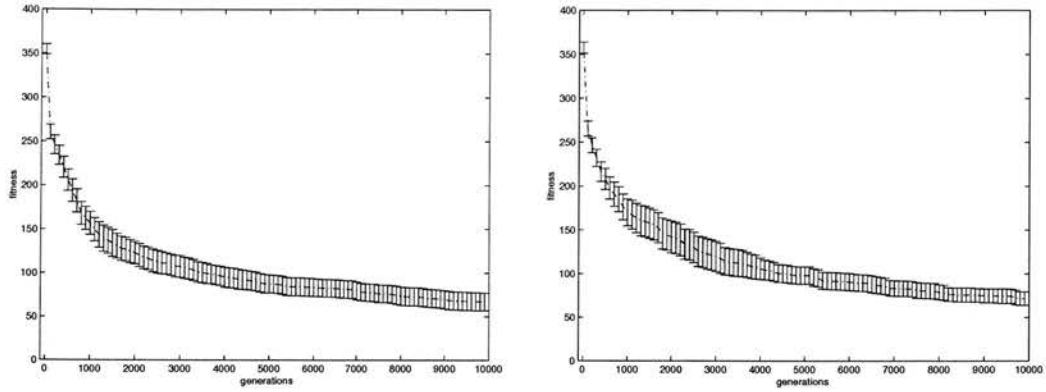
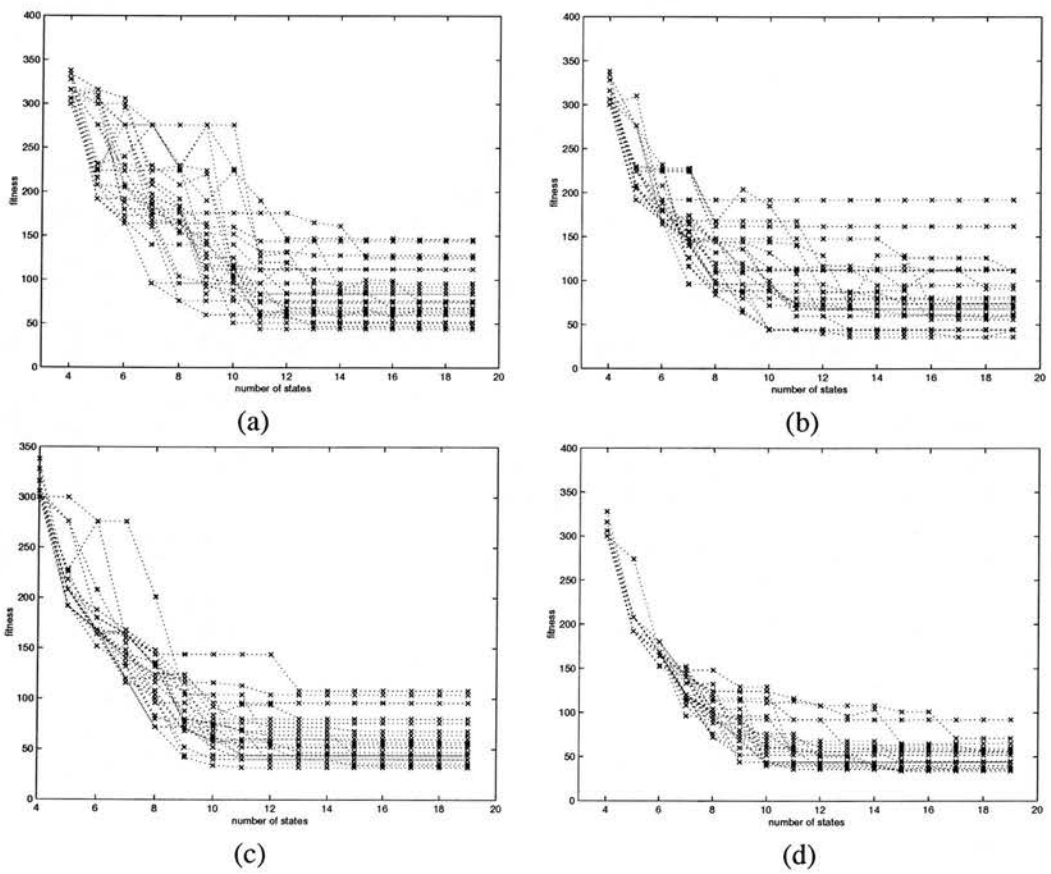


Figure B.1 continued ...



(m) (n)

Figure B.1 continued ...



(a) (b)

(c) (d)

Figure B.2: Distribution of the best controllers with N-state elitism Pareto optimization in Santa Fe trail problem (after 20,000 generations) (a) population size 25 (b) population size 50 (c) population size 100 (d) population size 200

Appendix C

Tartarus experiments

C.1 Figures

- Figure C.1 - Fitness distribution with 10 random sample boards and 8 states
- Figure C.2 - Fitness distribution with 20 random sample boards and 8 states
- Figure C.3 - Fitness distribution with 50 random sample boards and 8 states
- Figure C.4 - GP automata with 8 states
- Figure C.5 - FSM results with 8 states
- Figure C.6 - Fitness distribution with board sample size 50 and population size 100

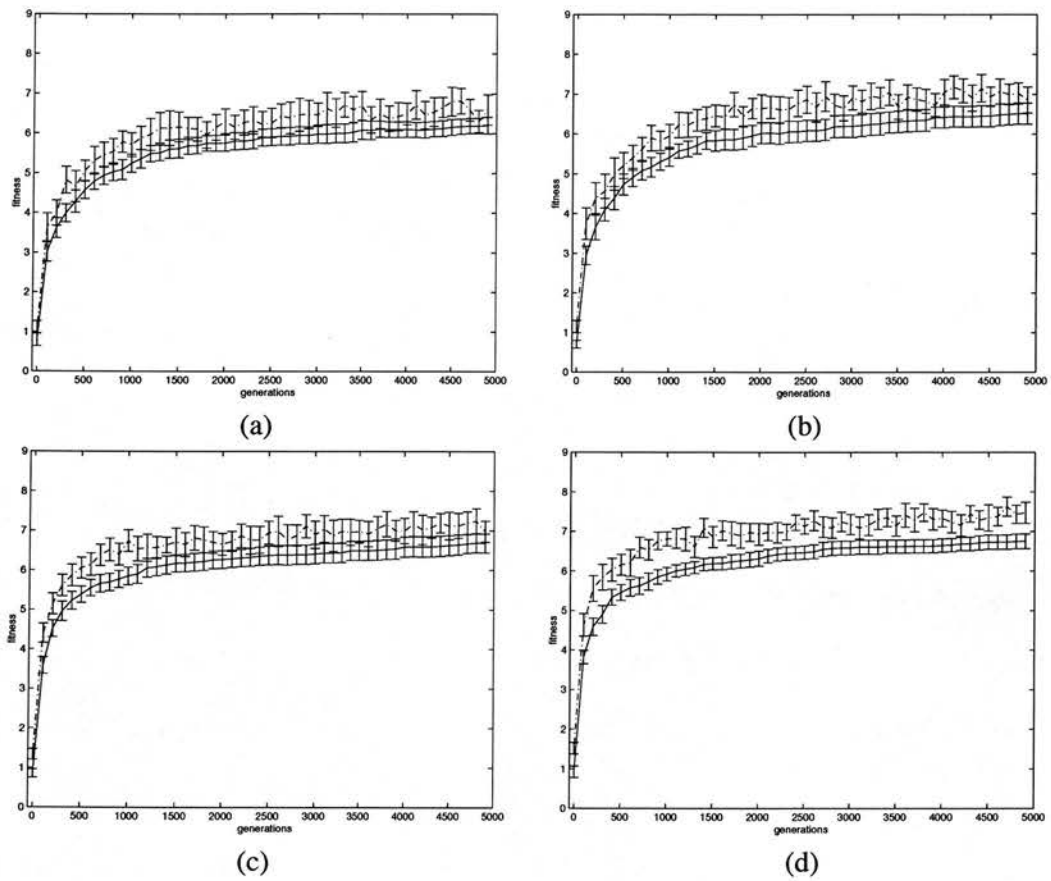


Figure C.1: Fitness distribution with 10 random sample boards and 8 states (a) population size 25 (b) population size 50 (c) population size 100 (d) population size 200 (dotdash : the average over K sample boards, solid : the average over $N(= 10,000)$ boards)

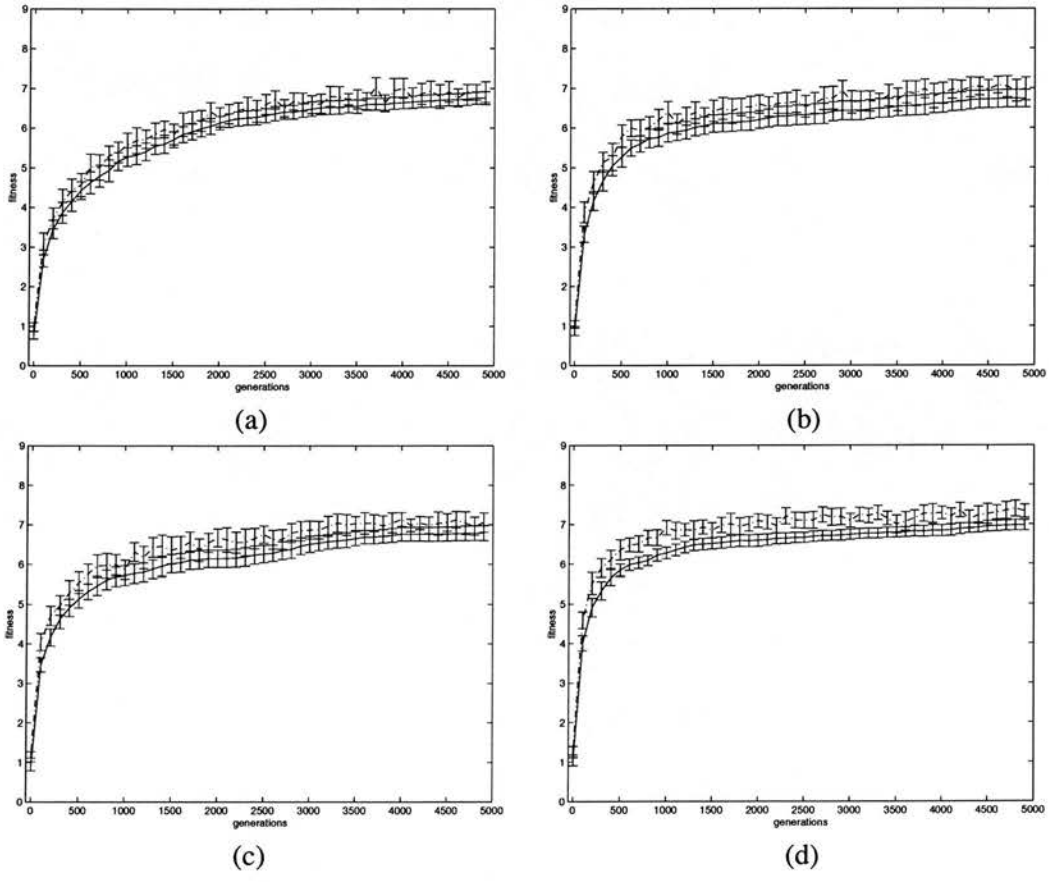


Figure C.2: Fitness distribution with 20 random sample boards and 8 states (a) population size 25 (b) population size 50 (c) population size 100 (d) population size 200 (dotdash : the average over K sample boards, solid : the average over $N(= 10,000)$ boards)

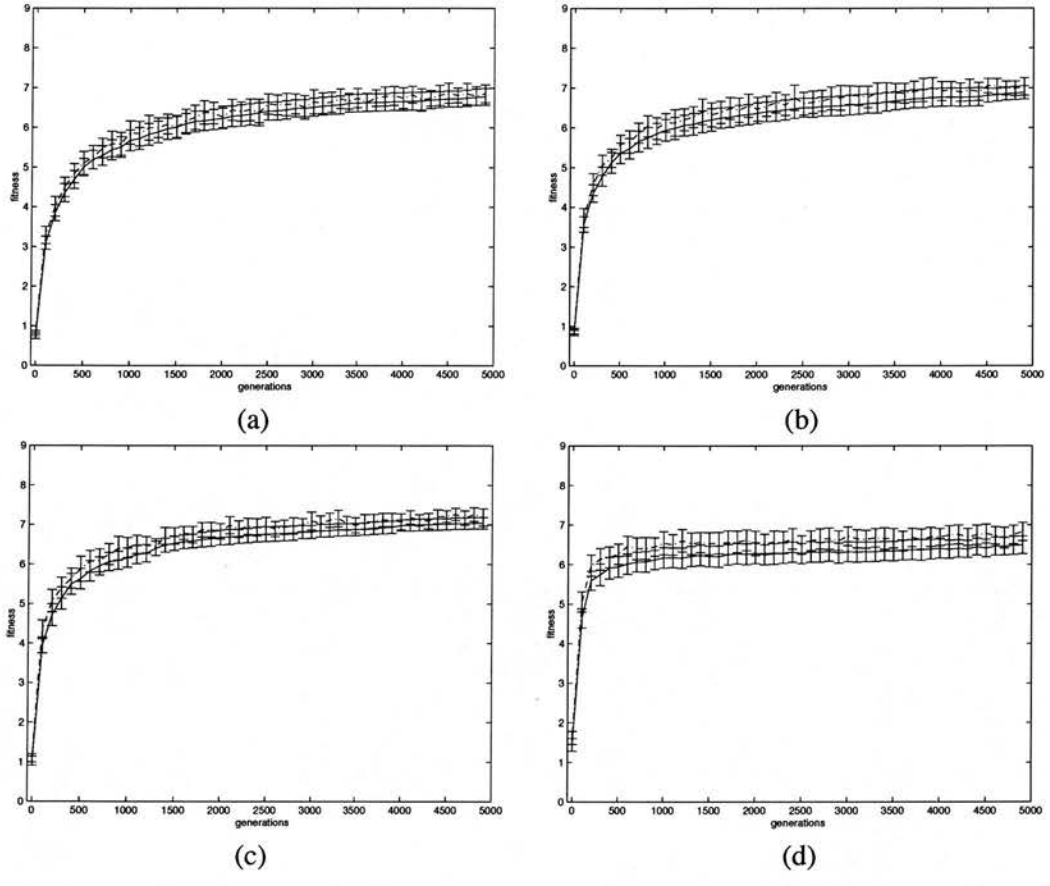


Figure C.3: Fitness distribution with 50 random sample boards and 8 states (a) population size 25 (b) population size 50 (c) population size 100 (d) population size 200 (dotdash : the average over K sample boards, solid : the average over $N(= 10,000)$ boards)

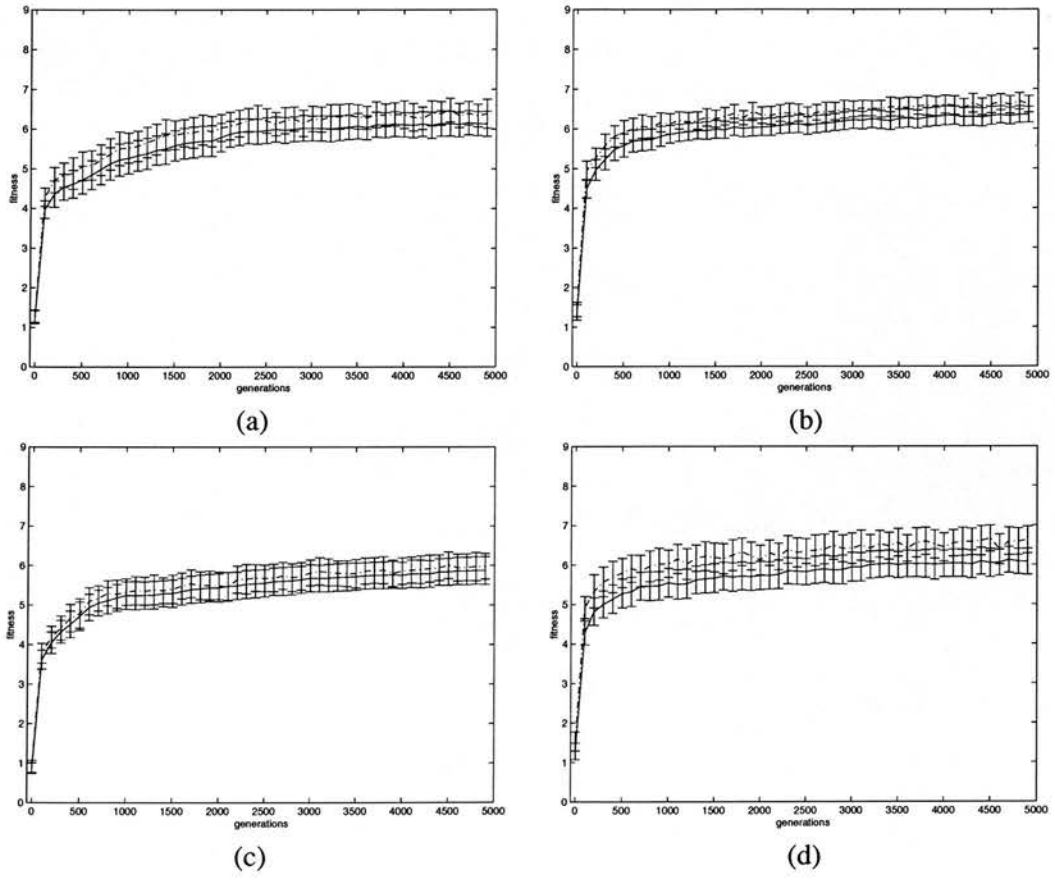


Figure C.4: GP automata results with 8 states (a) population size 50, 50 random boards (b) population size 100, 50 random boards (c) population size 20, 100 random boards (d) population size 100, 20 random boards (dotdash : the average over K sample boards, solid : the average over $N(= 10,000)$ boards)

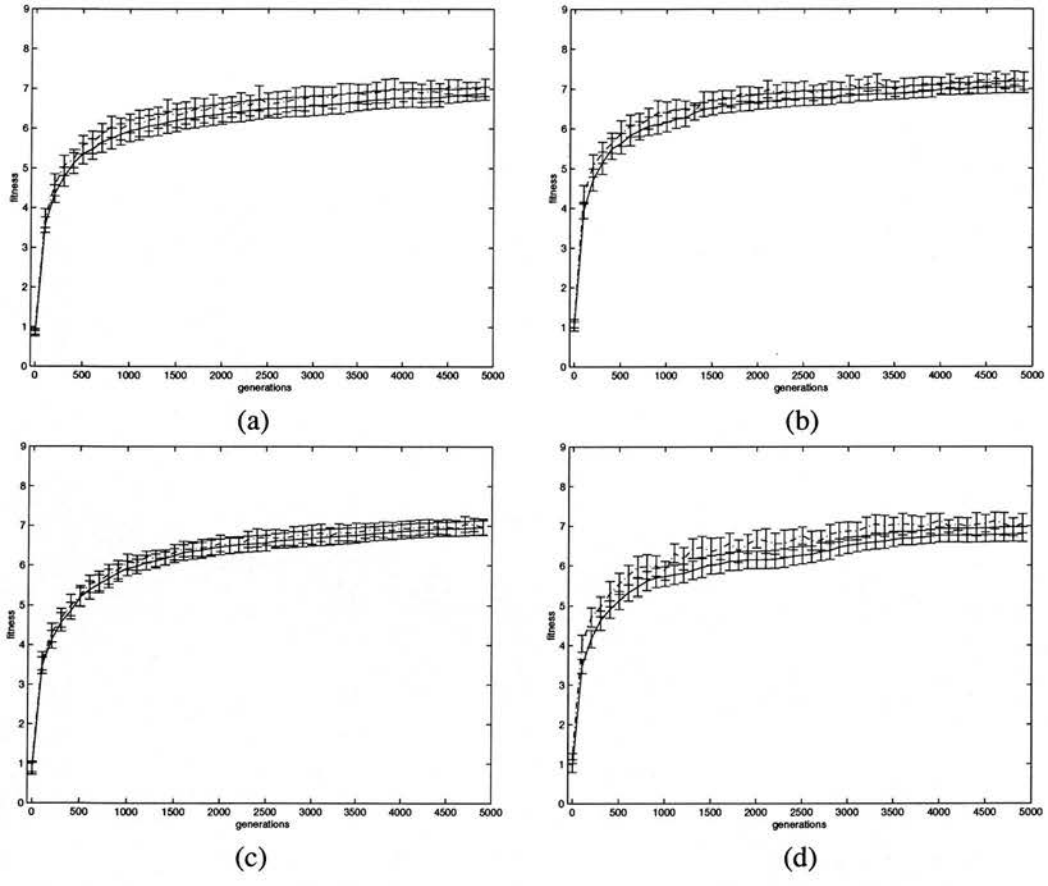


Figure C.5: FSM results with 8 states (a) population size 50, 50 random boards (b) population size 100, 50 random boards (c) population size 20, 100 random boards (d) population size 100, 20 random boards (dotdash : the average over K sample boards, solid : the average over $N(= 10,000)$ boards)

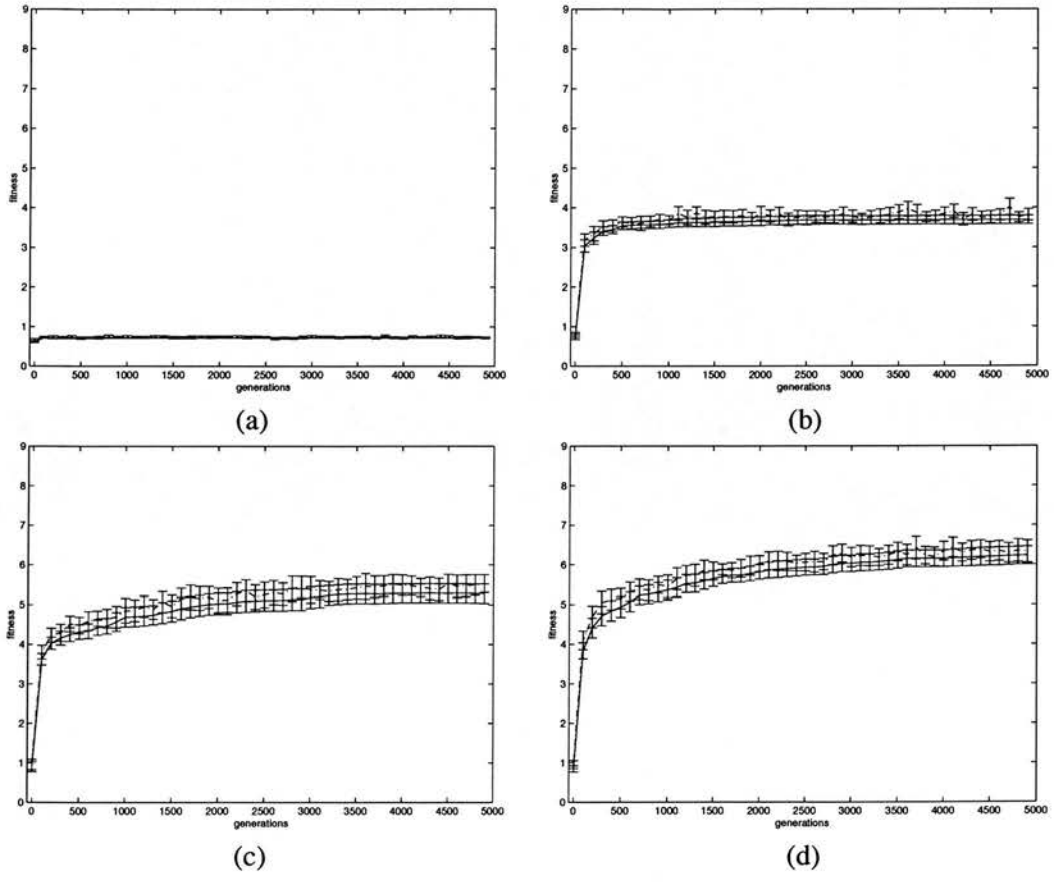
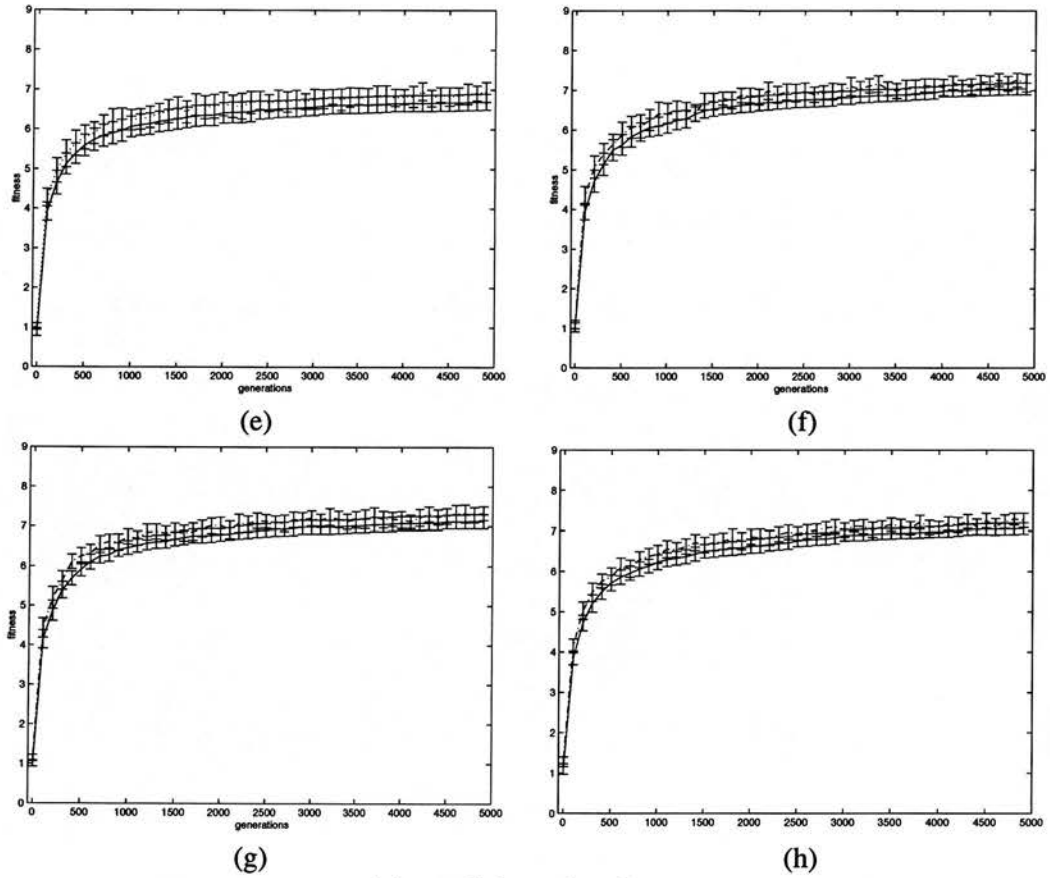


Figure C.6: Fitness distribution with board sample size 50 and population size 100 (a) 1 state (b) 2 states (c) 3 states (d) 4 states (e) 6 states (f) 8 states (g) 10 states (h) 12 states (dotdash : the average over K sample boards, solid : the average over $N(= 10,000)$ boards)



(e) (f)

(g) (h)

Figure C.6 continued ...

Appendix D

Woods Environment Experiments

D.1 Figure

- Figure D.1 - Memory analysis over woods102 environment
- Figure D.2 - Memory analysis over maze10 environment
- Figure D.3 - Memory analysis over woods14 environment with 4 sensors

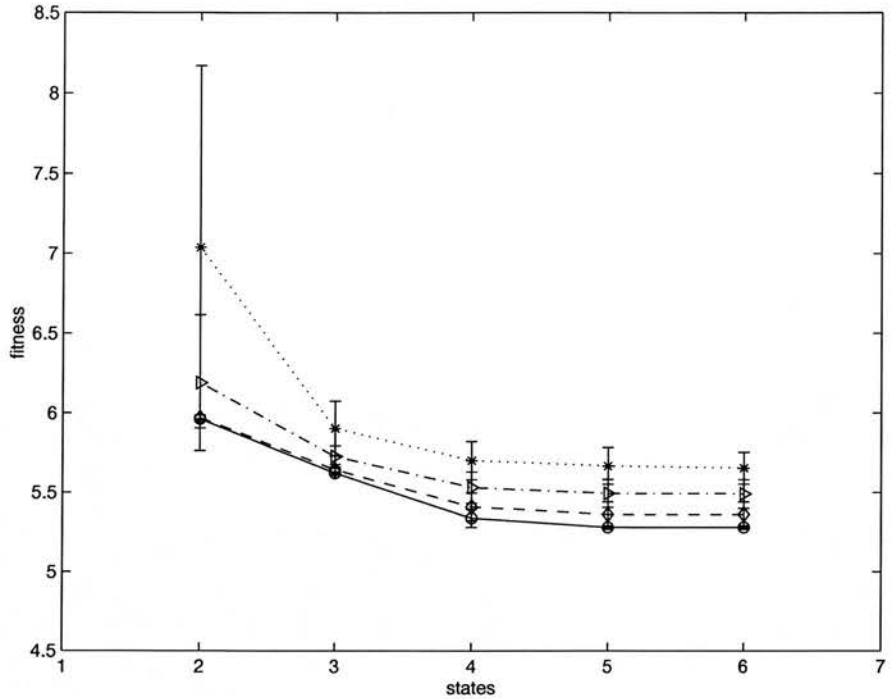


Figure D.1: Memory analysis over woods102 environment (* : 100 generations, ▷ : 200 generations, ◇ : 500 generations, ○ : 1000 generations)

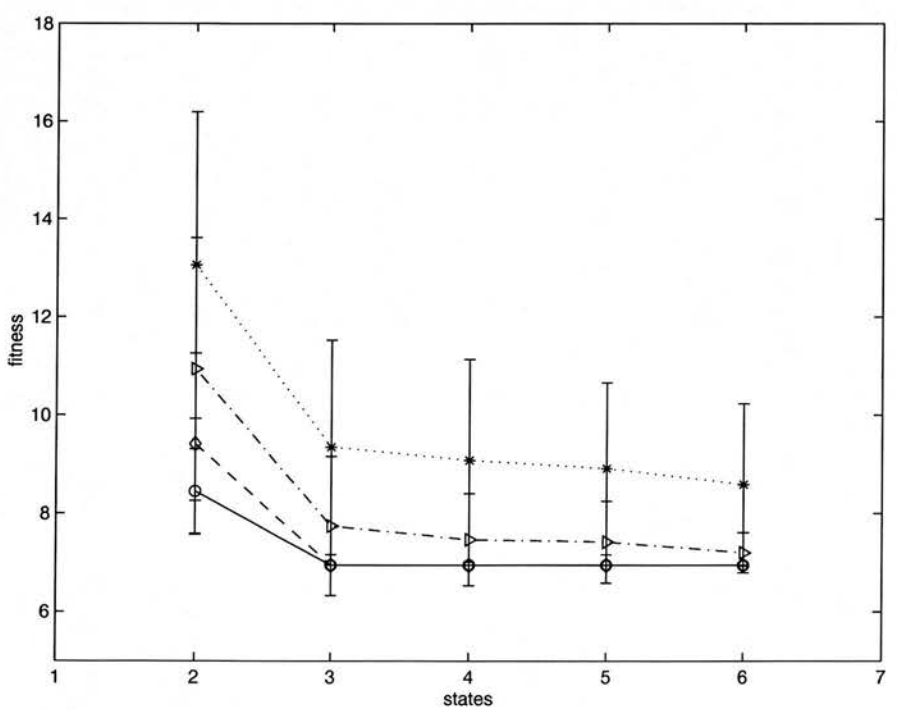


Figure D.2: Memory analysis over maze10 environment (* : 50 generations, ▷ : 200 generations, ◇ : 500 generations, ○ : 1000 generations)

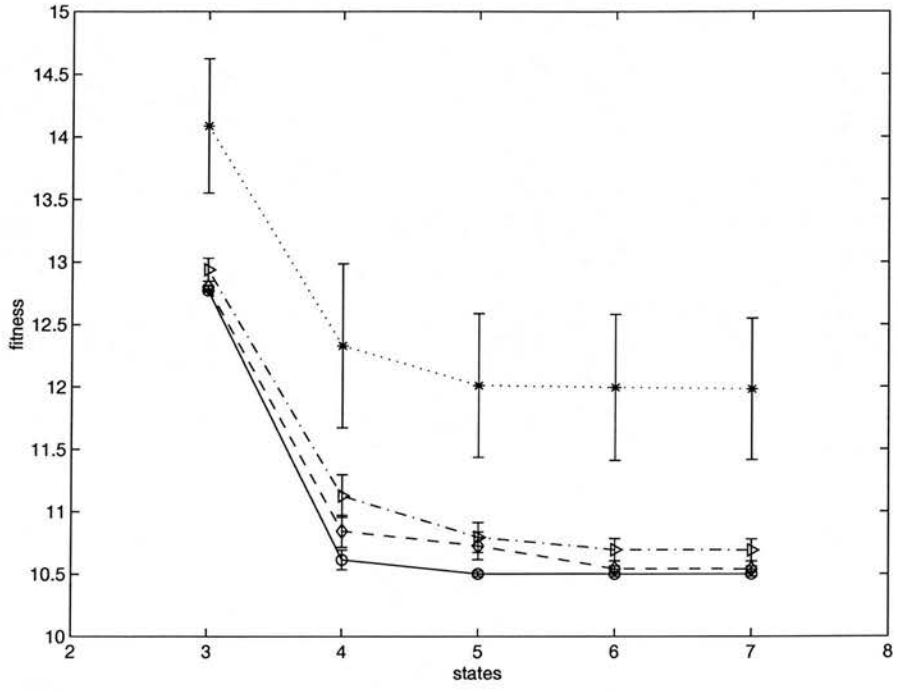


Figure D.3: Memory analysis over woods14 environment with 4 sensors (* : 100 generations, ▷ : 500 generations, ◇ : 1000 generations, ○ : 3000 generations)

Appendix E

Robotic tasks

E.1 Tables

- Table E.1 - Performance of various memory states with 2 sensors for wall following
- Table E.2 - Performance of various memory states with 4 sensors for wall following
- Table E.3 - Performance with neural networks for wall following
- Table E.4 - Performance of various memory states with 2 sensors for corridor following
- Table E.5 - Performance of various memory states with 4 sensors for corridor following
- Table E.6 - Performance with neural networks for corridor following
- Table E.7 - Box pushing performance with various memory states over 2 infrared sensors and 4 infrared sensors
- Table E.8 - Box pushing performance with a small-sized box over 2 infrared sensors and 4 infrared sensors
- Table E.9 - Box pushing performance with rule-based structure and neural networks
- Table E.10 - Box pushing performance with rule-based structure and neural networks in small box experiments

control structure		env0	env1	env2	env3
2 sensors, 1 state	c_0	0.0 ± 0.0	1.0 ± 0.0	3.4 ± 1.0	1.2 ± 0.4
	c_1*	0.2 ± 0.4	1.2 ± 0.9	5.0 ± 1.3	15.2 ± 2.0
	c_2	0.0 ± 0.0	2.6 ± 1.3	92.2 ± 2.3	17.6 ± 3.0
	c_3	0.1 ± 0.3	0.4 ± 0.7	2.7 ± 0.9	2.0 ± 0.0
	c_4	0.2 ± 0.6	1.7 ± 1.7	3.5 ± 0.9	2.1 ± 0.3
	c_5*	0.1 ± 0.3	0.7 ± 0.7	3.3 ± 0.6	2.6 ± 1.6
	c_6*	0.0 ± 0.0	0.6 ± 0.7	2.9 ± 0.9	0.1 ± 0.3
	c_7	0.4 ± 1.0	1.7 ± 1.4	2.9 ± 0.6	2.2 ± 0.4
	c_8	0.0 ± 0.0	1.0 ± 0.0	4.0 ± 0.4	3.0 ± 0.0
	c_9	0.2 ± 0.6	1.6 ± 1.2	2.7 ± 0.8	3.3 ± 1.2
2 sensors, 2 states	c_0*	0.0 ± 0.0	2.5 ± 0.5	6.2 ± 0.4	34.6 ± 3.9
	c_1	0.0 ± 0.0	3.5 ± 0.5	36.0 ± 0.7	64.5 ± 3.3
	c_2	0.0 ± 0.0	4.4 ± 0.8	3.8 ± 0.7	35.8 ± 3.6
	c_3*	0.0 ± 0.0	1.2 ± 0.8	4.6 ± 0.7	0.5 ± 0.7
	c_4	0.0 ± 0.0	2.0 ± 0.5	93.2 ± 2.3	7.5 ± 1.7
	c_5	0.0 ± 0.0	1.6 ± 0.7	28.7 ± 4.0	21.9 ± 3.8
	c_6	0.0 ± 0.0	1.2 ± 0.4	4.8 ± 0.7	7.2 ± 2.6
	c_7	0.0 ± 0.0	2.9 ± 0.6	12.1 ± 1.0	45.9 ± 3.7
	c_8	0.0 ± 0.0	1.8 ± 0.8	6.6 ± 0.9	30.5 ± 3.4
	c_9*	0.0 ± 0.0	2.0 ± 0.0	3.8 ± 0.7	0.0 ± 0.0
2 sensors, 3 states	c_0*	0.0 ± 0.0	0.0 ± 0.0	4.4 ± 0.5	0.0 ± 0.0
	c_1	0.1 ± 0.3	1.9 ± 0.3	7.7 ± 1.1	62.8 ± 1.9
	c_2	0.0 ± 0.0	3.3 ± 0.5	6.5 ± 0.8	25.5 ± 5.6
	c_3	0.0 ± 0.0	2.0 ± 0.0	89.3 ± 3.8	1.9 ± 0.9
	c_4	0.0 ± 0.0	0.1 ± 0.3	4.9 ± 0.9	6.1 ± 2.6
	c_5*	0.0 ± 0.0	0.0 ± 0.0	4.1 ± 0.9	0.0 ± 0.0
	c_6*	0.0 ± 0.0	0.9 ± 0.3	4.6 ± 0.5	4.5 ± 2.3
	c_7	0.0 ± 0.0	4.2 ± 0.9	44.1 ± 4.2	75.3 ± 1.2
	c_8	0.0 ± 0.0	1.7 ± 0.5	52.3 ± 5.2	2.1 ± 2.0
	c_9	0.0 ± 0.0	1.7 ± 0.7	6.2 ± 0.4	16.7 ± 3.9
2 sensors, 4 states	c_0	0.0 ± 0.0	1.0 ± 0.0	6.0 ± 0.0	1.8 ± 0.4
	c_1*	0.0 ± 0.0	0.0 ± 0.0	4.7 ± 0.9	0.1 ± 0.3
	c_2	0.0 ± 0.0	1.1 ± 0.3	4.8 ± 0.4	2.6 ± 0.9
	c_3*	0.0 ± 0.0	0.8 ± 0.6	4.7 ± 0.5	0.9 ± 0.7
	c_4	0.0 ± 0.0	1.4 ± 0.7	61.3 ± 5.4	5.8 ± 1.5
	c_5*	0.0 ± 0.0	1.1 ± 0.3	3.5 ± 0.5	2.7 ± 1.3
	c_6	0.0 ± 0.0	0.7 ± 0.9	2.7 ± 0.8	4.2 ± 1.4
	c_7*	0.0 ± 0.0	0.0 ± 0.0	4.3 ± 0.8	2.1 ± 2.0
	c_8*	0.0 ± 0.0	0.5 ± 0.5	4.0 ± 0.0	0.6 ± 0.7
	c_9	0.0 ± 0.0	1.3 ± 1.1	5.2 ± 0.6	22.5 ± 3.3

(a) collision test

Table E.1: Performance of various memory states with 2 sensors for wall following

control structure		env0	env1	env2	env3
2 sensors, 1 state	c_0	70.33 ± 0.91	72.09 ± 0.30	87.29 ± 0.24	92.39 ± 0.27
	c_1*	67.97 ± 0.85	69.70 ± 0.43	38.38 ± 0.14	82.38 ± 0.57
	c_2	69.52 ± 0.51	71.66 ± 0.29	83.07 ± 2.67	89.70 ± 0.38
	c_3	69.71 ± 0.45	71.11 ± 0.21	84.56 ± 0.37	90.36 ± 0.25
	c_4	68.39 ± 0.51	70.27 ± 0.14	84.29 ± 0.26	90.67 ± 0.27
	c_5*	68.18 ± 0.72	69.92 ± 0.33	40.00 ± 0.19	82.98 ± 0.53
	c_6*	68.21 ± 0.62	69.83 ± 0.29	37.07 ± 0.13	82.03 ± 0.30
	c_7	68.40 ± 0.49	69.89 ± 0.27	84.22 ± 0.22	90.56 ± 0.29
	c_8	71.46 ± 0.62	73.05 ± 0.25	87.22 ± 0.29	92.47 ± 0.19
	c_9	68.37 ± 0.60	69.53 ± 0.43	84.28 ± 0.24	90.37 ± 0.32
2 sensors, 2 states	c_0	54.00 ± 0.01	54.00 ± 0.00	45.62 ± 0.08	83.41 ± 0.80
	c_1	54.00 ± 0.00	54.18 ± 0.14	39.86 ± 0.64	89.26 ± 2.27
	c_2	54.00 ± 0.00	54.00 ± 0.00	35.73 ± 0.22	79.67 ± 0.68
	c_3*	54.00 ± 0.00	54.00 ± 0.00	46.76 ± 0.19	77.98 ± 0.11
	c_4	54.00 ± 0.01	54.02 ± 0.02	48.31 ± 1.84	78.69 ± 0.04
	c_5	54.00 ± 0.00	54.05 ± 0.12	42.29 ± 0.30	80.43 ± 0.52
	c_6	54.00 ± 0.00	54.00 ± 0.00	40.82 ± 0.35	77.58 ± 0.54
	c_7	54.03 ± 0.08	54.34 ± 0.44	37.71 ± 0.53	80.35 ± 0.84
	c_8	54.01 ± 0.02	54.00 ± 0.00	45.28 ± 0.51	80.72 ± 0.90
	c_9*	54.00 ± 0.00	54.00 ± 0.00	38.96 ± 0.06	76.84 ± 0.24
2 sensors, 3 states	c_0*	54.00 ± 0.00	54.00 ± 0.00	44.58 ± 0.10	78.91 ± 0.04
	c_1	54.02 ± 0.02	54.03 ± 0.01	48.60 ± 0.05	93.50 ± 1.11
	c_2	54.02 ± 0.03	54.07 ± 0.02	46.05 ± 0.23	81.03 ± 0.89
	c_3	54.02 ± 0.02	54.05 ± 0.01	44.52 ± 2.54	78.05 ± 0.06
	c_4	54.00 ± 0.00	54.00 ± 0.00	38.57 ± 0.19	75.45 ± 0.30
	c_5*	54.00 ± 0.00	54.00 ± 0.00	39.27 ± 0.12	76.89 ± 0.33
	c_6*	54.00 ± 0.00	54.00 ± 0.00	38.82 ± 0.18	75.92 ± 0.18
	c_7	54.00 ± 0.01	54.02 ± 0.04	46.97 ± 0.58	97.93 ± 0.91
	c_8	54.00 ± 0.00	54.00 ± 0.01	42.61 ± 1.30	78.23 ± 0.23
	c_9	54.00 ± 0.00	54.00 ± 0.00	38.34 ± 0.23	78.72 ± 0.74
2 sensors, 4 states	c_0	54.00 ± 0.00	54.00 ± 0.00	37.51 ± 0.08	75.99 ± 0.34
	c_1*	54.00 ± 0.00	54.00 ± 0.00	39.82 ± 0.22	76.80 ± 0.19
	c_2	54.00 ± 0.00	54.00 ± 0.00	38.17 ± 0.09	74.89 ± 0.22
	c_3*	54.00 ± 0.00	54.00 ± 0.00	48.50 ± 0.18	78.53 ± 0.37
	c_4	54.00 ± 0.00	54.00 ± 0.00	41.10 ± 1.33	80.68 ± 0.73
	c_5*	54.00 ± 0.00	54.00 ± 0.00	36.86 ± 0.11	76.91 ± 0.43
	c_6	54.00 ± 0.00	54.00 ± 0.00	83.20 ± 0.44	93.97 ± 0.28
	c_7*	54.00 ± 0.00	54.00 ± 0.00	36.15 ± 0.19	77.02 ± 0.39
	c_8*	54.00 ± 0.00	54.00 ± 0.00	41.83 ± 0.08	78.04 ± 0.07
	c_9	54.00 ± 0.00	54.00 ± 0.00	45.23 ± 0.11	80.94 ± 0.52

(b) performance test

Table E.1 continued ...

control structure		env0	env1	env2	env3
4 sensors, 1 state	c_0*	0.0±0.0	0.0±0.0	0.0±0.0	0.3±0.5
	c_1*	0.0±0.0	0.2±0.4	0.3±0.5	1.0±0.6
	c_2	0.0±0.0	1.0±0.0	14.3±2.6	8.6±1.9
	c_3*	0.0±0.0	0.5±0.5	0.0±0.0	1.7±0.9
	c_4	0.2±0.6	4.4±0.8	91.9±2.9	93.6±2.2
	c_5	0.0±0.0	2.8±0.4	94.1±2.3	95.1±1.4
4 sensors, 2 states	c_0*	0.0±0.0	0.6±0.5	1.4±0.5	0.6±0.8
	c_1	0.0±0.0	2.0±0.0	1.5±0.7	1.3±0.6
	c_2	0.0±0.0	1.6±0.7	94.5±2.0	3.0±0.9
	c_3	0.0±0.0	1.6±0.5	100.0±0.0	0.3±0.5
	c_4	0.0±0.0	2.5±0.7	5.1±1.0	11.3±2.6
	c_5*	0.0±0.0	1.0±0.0	0.0±0.0	1.0±0.0
4 sensors, 3 states	c_0	0.0±0.0	0.0±0.0	46.6±5.5	32.7±4.1
	c_1*	0.3±0.5	2.0±0.8	4.9±2.0	27.9±3.1
	c_2*	0.0±0.0	0.6±0.8	0.8±0.9	0.3±0.5
	c_3	0.0±0.0	2.0±0.0	14.5±3.7	20.9±5.8
	c_4	0.0±0.0	2.0±0.0	61.9±3.6	63.4±3.4
	c_5	0.0±0.0	1.0±0.0	16.9±3.3	3.8±1.2
	c_6	0.0±0.0	1.6±0.5	99.5±0.9	0.1±0.3
	c_7*	0.0±0.0	4.0±0.0	2.1±0.9	1.6±0.8
4 sensors, 4 states	c_0	0.2±0.4	1.2±1.0	59.3±3.6	64.7±1.9
	c_1	0.0±0.0	4.3±1.0	89.5±1.43	74.4±1.1
	c_2*	0.0±0.0	0.5±0.5	0.7±0.9	0.7±0.6
	c_3	0.0±0.0	1.6±0.7	68.0±4.6	22.3±4.3
	c_4	0.0±0.0	0.4±0.5	87.7±3.3	79.1±2.7
	c_5*	0.0±0.0	0.5±0.7	3.0±1.2	0.5±0.7
	c_6	0.0±0.0	1.3±0.5	100.0±0.0	0.1±0.3
	c_7*	0.0±0.0	2.2±0.4	4.6±0.8	7.5±2.9

(a) collision test

Table E.2: Performance of various memory states with 4 sensors for wall following

control structure		env0	env1	env2	env3
4 sensors, 1 state	c_0*	67.71 ± 0.94	69.67 ± 0.37	38.43 ± 0.14	81.61 ± 0.41
	c_1*	72.42 ± 0.67	74.76 ± 0.30	50.58 ± 0.43	86.96 ± 0.29
	c_2	67.88 ± 0.61	70.05 ± 0.31	42.22 ± 0.24	84.73 ± 0.50
	c_3*	68.27 ± 0.91	69.90 ± 0.32	45.74 ± 0.40	83.67 ± 0.36
	c_4	68.41 ± 0.54	69.29 ± 0.19	79.19 ± 2.23	78.38 ± 2.08
	c_5	70.41 ± 0.51	70.92 ± 0.41	78.23 ± 3.64	78.64 ± 1.30
4 sensors, 2 states	c_0*	54.36 ± 0.03	54.37 ± 0.02	49.27 ± 0.04	78.92 ± 0.29
	c_1*	54.07 ± 0.11	54.03 ± 0.05	85.92 ± 0.58	89.84 ± 0.52
	c_2	54.03 ± 0.05	54.02 ± 0.02	57.02 ± 7.32	85.27 ± 0.40
	c_3	54.01 ± 0.01	54.05 ± 0.14	100.0 ± 0.0	78.35 ± 0.05
	c_4	54.00 ± 0.00	54.00 ± 0.00	38.58 ± 0.22	78.37 ± 0.66
	c_5*	54.00 ± 0.00	54.00 ± 0.00	38.29 ± 0.30	76.10 ± 0.31
4 sensors, 3 states	c_0	54.00 ± 0.00	54.00 ± 0.00	58.06 ± 0.63	85.40 ± 0.50
	c_1*	54.00 ± 0.00	54.00 ± 0.00	33.97 ± 0.30	79.34 ± 0.81
	c_2*	54.00 ± 0.00	59.19 ± 0.59	49.46 ± 0.10	81.67 ± 0.28
	c_3	54.00 ± 0.00	54.00 ± 0.00	44.66 ± 0.57	80.11 ± 0.66
	c_4	54.06 ± 0.13	54.00 ± 0.00	80.28 ± 1.50	91.63 ± 1.17
	c_5	54.00 ± 0.00	54.69 ± 0.21	47.17 ± 0.72	78.97 ± 0.09
	c_6	54.00 ± 0.01	54.11 ± 0.18	100.0 ± 0.0	78.33 ± 0.04
	c_7*	54.00 ± 0.09	54.00 ± 0.00	37.45 ± 0.24	77.88 ± 0.42
4 sensors, 4 states	c_0	54.02 ± 0.05	54.01 ± 0.07	45.95 ± 1.07	95.38 ± 1.25
	c_1	54.00 ± 0.00	54.00 ± 0.00	52.23 ± 3.12	98.08 ± 0.56
	c_2*	54.00 ± 0.01	59.15 ± 0.27	49.62 ± 0.12	81.81 ± 0.21
	c_3	54.00 ± 0.00	54.24 ± 0.29	45.93 ± 2.15	87.32 ± 1.10
	c_4	54.00 ± 0.00	54.00 ± 0.00	86.96 ± 1.94	95.19 ± 1.66
	c_5*	54.01 ± 0.02	54.16 ± 0.20	48.01 ± 0.38	79.11 ± 0.11
	c_6	54.00 ± 0.00	54.00 ± 0.00	100.0 ± 0.0	78.32 ± 0.06
	c_7*	54.00 ± 0.00	54.00 ± 0.00	38.57 ± 0.27	77.89 ± 0.58

(b) performance test

Table E.2 continued ...

control structure		env0	env1	env2	env3
2 sensors, neural	c_0*	0.0±0.0	1.9±0.3	34.2±2.8	7.0±0.0
	c_1	0.0±0.0	0.0±0.0	85.6±1.2	1.1±0.3
	c_2	0.0±0.0	0.3±0.5	92.7±1.9	1.8±0.4
	c_3	0.0±0.0	1.0±0.0	57.9±1.4	1.0±0.0
	c_4	0.0±0.0	0.0±0.0	79.1±2.7	0.6±0.5
	c_5	0.0±0.0	1.0±0.0	81.1±2.5	3.0±0.0
	c_6	0.0±0.0	1.8±0.4	60.9±5.8	3.1±0.3
	c_7	0.2±0.4	0.0±0.0	47.9±5.3	2.0±0.0
	c_8	0.0±0.0	0.0±0.0	71.6±2.5	5.6±0.5
	c_9*	0.0±0.0	1.3±0.6	54.8±3.8	0.1±0.3
	c_{10}	0.0±0.0	1.0±0.0	78.1±2.0	0.0±0.0
4 sensors, neural	c_0	0.0±0.0	0.0±0.0	14.9±1.7	1.5±0.8
	c_1	0.0±0.0	0.0±0.0	13.8±4.4	0.1±0.3
	c_2*	0.0±0.0	0.0±0.0	2.4±1.5	0.8±0.4
	c_3	0.0±0.0	0.0±0.0	34.9±3.8	2.8±1.3
	c_4	0.0±0.0	0.0±0.0	20.2±4.5	0.2±0.4
	c_5*	0.0±0.0	1.0±0.0	6.6±1.0	4.9±0.3
	c_6	0.0±0.0	0.0±0.0	38.2±5.8	0.0±0.0
	c_7	0.0±0.0	0.0±0.0	3.2±1.2	0.0±0.0
	c_8	0.0±0.0	1.0±0.0	5.7±2.6	0.2±0.4
	c_9	0.0±0.0	0.0±0.0	4.0±1.8	4.8±0.4
	c_{10}	0.0±0.0	0.5±0.5	1.5±0.9	2.6±0.5
6 sensors, neural	c_0	0.0±0.0	0.0±0.0	0.6±5.7	20.2±2.7
	c_1*	0.0±0.0	0.0±0.0	2.4±0.7	0.1±0.3
	c_2	0.0±0.0	3.2±0.4	100.0±0.0	2.8±1.3
	c_3	0.0±0.0	0.0±0.0	0.2±0.4	0.0±0.0
	c_4	0.0±0.0	2.0±0.0	45.5±3.7	0.0±0.0
	c_5*	0.0±0.0	0.0±0.0	1.0±0.0	0.1±0.3
	c_6	0.0±0.0	3.2±0.4	7.9±1.2	8.0±0.0
	c_7	0.0±0.0	0.0±0.0	9.9±3.1	65.4±1.6
	c_8	0.0±0.0	1.7±0.8	100.0±0.0	66.3±1.6
	c_9	0.0±0.0	0.3±0.8	1.2±0.4	38.6±3.3

(a) collision test

Table E.3: Performance with neural networks for wall following

control structure		env0	env1	env2	env3
2 sensors, neural	c_0*	56.46 ± 0.59	60.25 ± 0.64	50.84 ± 0.32	77.92 ± 0.44
	c_1	55.10 ± 0.42	61.14 ± 0.17	59.37 ± 1.34	82.42 ± 0.17
	c_2	56.70 ± 0.35	62.45 ± 0.26	63.39 ± 4.13	83.55 ± 0.45
	c_3	56.59 ± 0.07	62.40 ± 0.22	81.39 ± 1.10	81.68 ± 0.11
	c_4	56.52 ± 0.15	62.69 ± 0.21	73.99 ± 1.99	83.95 ± 0.34
	c_5	54.94 ± 0.03	60.97 ± 0.24	70.47 ± 3.03	82.07 ± 0.12
	c_6	56.02 ± 1.16	60.53 ± 1.05	51.99 ± 0.70	80.04 ± 0.25
	c_7	56.94 ± 1.10	59.37 ± 0.53	57.27 ± 1.21	82.34 ± 0.48
	c_8	55.66 ± 0.70	59.62 ± 0.77	58.65 ± 1.34	79.00 ± 0.43
	c_9*	55.66 ± 1.15	59.70 ± 0.67	49.62 ± 0.65	77.78 ± 0.21
	c_{10}	54.93 ± 0.01	60.86 ± 0.17	60.39 ± 2.53	81.95 ± 0.44
4 sensors, neural	c_0	55.19 ± 0.83	59.26 ± 0.51	46.23 ± 0.48	77.72 ± 0.40
	c_1	56.56 ± 0.57	60.69 ± 1.03	52.01 ± 1.12	78.60 ± 0.50
	c_2*	55.57 ± 0.94	59.54 ± 0.67	49.57 ± 0.43	77.05 ± 0.55
	c_3	55.45 ± 0.78	59.75 ± 0.52	54.31 ± 0.76	80.24 ± 0.34
	c_4	55.82 ± 0.69	59.49 ± 0.83	44.06 ± 0.50	77.45 ± 0.34
	c_5*	55.71 ± 1.13	59.20 ± 0.67	56.81 ± 0.94	77.84 ± 0.31
	c_6	55.92 ± 0.83	59.04 ± 0.92	45.28 ± 0.76	78.09 ± 0.33
	c_7	55.62 ± 0.83	60.34 ± 0.79	47.83 ± 0.25	77.16 ± 0.33
	c_8	55.58 ± 1.00	58.80 ± 0.71	60.31 ± 0.38	78.36 ± 0.19
	c_9	57.75 ± 0.08	60.51 ± 0.20	75.12 ± 0.07	84.55 ± 0.15
	c_{10}	55.89 ± 0.65	59.39 ± 0.54	64.87 ± 0.69	79.68 ± 0.22
6 sensors, neural	c_0	54.04 ± 0.12	54.03 ± 0.05	79.21 ± 0.50	95.83 ± 0.33
	c_1*	54.00 ± 0.00	57.66 ± 0.24	41.18 ± 0.09	78.91 ± 0.09
	c_2	54.00 ± 0.00	54.00 ± 0.00	100.00 ± 0.00	70.42 ± 0.15
	c_3	56.66 ± 0.57	60.55 ± 0.90	73.05 ± 0.54	80.87 ± 0.43
	c_4	54.03 ± 0.07	54.01 ± 0.03	50.79 ± 0.92	77.58 ± 0.36
	c_5*	54.61 ± 0.60	58.86 ± 0.63	46.89 ± 0.12	77.19 ± 0.60
	c_6	56.80 ± 0.90	60.98 ± 0.96	76.67 ± 0.43	83.92 ± 0.53
	c_7	54.14 ± 0.44	57.91 ± 0.75	45.28 ± 0.23	97.42 ± 1.00
	c_8	54.05 ± 0.17	54.00 ± 0.00	100.00 ± 0.00	98.67 ± 0.07
	c_9	54.01 ± 0.02	56.05 ± 0.30	45.77 ± 0.11	84.22 ± 0.78

(b) performance test

Table E.3 continued ...

control structure		env0	env1	env2	env3
2 sensors, 1 state	c_0	0.4±0.7	3.4±1.1	4.7±0.6	9.5±0.8
	c_1	0.0±0.0	1.0±0.0	5.8±0.4	7.2±1.1
	c_2	0.4±0.5	2.4±0.5	1.7±0.5	8.2±0.8
	c_3	0.2±0.4	1.4±0.5	2.6±0.8	5.9±0.9
	c_4	0.0±0.0	1.1±0.5	1.7±0.5	2.4±0.8
	c_5*	0.0±0.0	1.2±0.4	2.0±0.0	2.2±0.7
	c_6	0.1±0.3	1.0±0.0	2.2±0.6	6.0±1.1
	c_7*	0.0±0.0	1.2±0.4	2.0±0.4	5.1±1.3
	c_8	0.1±0.3	1.2±0.4	1.9±0.5	5.2±0.7
	c_9*	0.0±0.0	1.2±0.4	2.0±0.0	1.8±1.0
	c_{10}	0.3±0.5	1.7±0.6	1.5±0.5	8.4±0.5
	c_{11}	0.0±0.0	0.9±0.3	2.0±0.0	1.9±0.7
2 sensors, 2 states	c_0	0.6±1.0	5.1±0.8	7.5±0.5	6.2±0.9
	c_1	0.0±0.0	3.6±0.8	5.2±0.9	4.1±0.7
	c_2	0.1±0.3	2.4±0.8	6.5±0.5	4.5±0.7
	c_3	0.2±0.4	3.3±1.0	4.6±0.8	5.6±0.9
	c_4	0.1±0.3	3.7±1.0	4.6±0.8	4.3±1.2
	c_5	0.0±0.0	2.3±0.5	7.5±1.6	5.3±1.1
	c_6*	0.0±0.0	1.8±0.4	4.5±0.5	3.0±0.9
	c_7*	0.0±0.0	1.9±0.5	4.6±0.7	2.0±1.0
	c_8	0.0±0.0	3.0±0.6	8.0±0.8	7.1±1.0
	c_9	0.0±0.0	1.7±0.6	5.5±0.7	5.0±0.8
2 sensors, 3 states	c_0	0.2±0.4	4.4±1.1	6.4±0.8	6.7±0.8
	c_1	0.0±0.0	3.0±0.8	3.5±0.5	4.8±0.6
	c_2	0.1±0.3	3.0±0.7	9.2±0.9	7.7±0.6
	c_3	0.0±0.0	3.3±0.6	3.8±1.0	5.4±1.1
	c_4*	0.0±0.0	2.2±0.4	2.1±0.3	3.7±1.0
	c_5	0.0±0.0	1.8±0.4	3.9±1.3	4.6±0.7
	c_6	0.1±0.3	3.5±1.1	4.8±0.6	5.9±1.3
	c_7*	0.0±0.0	4.1±0.9	2.4±0.5	4.7±1.1
	c_8	0.0±0.0	2.0±0.8	4.3±0.5	3.8±1.0
	c_9	0.0±0.0	2.6±0.8	5.2±0.4	6.3±1.1

(a) collision test

Table E.4: Performance of various memory states with 2 sensors for corridor following

control structure		env0	env1	env2	env3
2 sensors, 1 state	c_0	58.57 ± 1.15	60.09 ± 0.61	60.88 ± 0.13	61.29 ± 0.10
	c_1	58.27 ± 0.79	58.44 ± 0.38	59.38 ± 0.13	59.38 ± 0.52
	c_2	58.16 ± 1.16	60.10 ± 0.75	61.39 ± 0.07	61.28 ± 0.09
	c_3	58.00 ± 0.63	58.47 ± 0.55	57.65 ± 0.08	59.28 ± 0.41
	c_4	59.32 ± 1.12	59.43 ± 0.73	62.91 ± 0.06	62.25 ± 0.22
	c_5*	58.60 ± 0.83	59.03 ± 0.60	61.42 ± 0.29	61.02 ± 0.26
	c_6	58.53 ± 0.73	58.50 ± 0.45	58.05 ± 0.15	59.26 ± 0.56
	c_7*	59.66 ± 1.22	59.46 ± 0.34	54.68 ± 0.08	55.84 ± 0.54
	c_8	58.06 ± 1.24	58.55 ± 0.18	58.11 ± 0.06	59.39 ± 0.46
	c_9*	59.05 ± 1.02	59.15 ± 0.37	61.15 ± 0.21	61.37 ± 0.35
	c_{10}	58.70 ± 0.82	59.73 ± 0.65	57.75 ± 0.09	57.88 ± 0.15
	c_{11}	59.69 ± 1.02	59.22 ± 0.48	62.96 ± 0.38	61.89 ± 0.42
2 sensors, 2 states	c_0	49.99 ± 0.11	50.46 ± 0.09	49.48 ± 0.08	47.75 ± 0.19
	c_1	50.17 ± 0.11	50.58 ± 0.13	50.71 ± 0.23	48.92 ± 0.41
	c_2	51.25 ± 0.72	51.99 ± 0.33	50.15 ± 0.11	48.55 ± 0.21
	c_3	50.97 ± 0.32	51.73 ± 0.04	52.01 ± 0.14	50.92 ± 0.19
	c_4	49.92 ± 0.11	51.00 ± 0.14	49.60 ± 0.08	47.93 ± 0.24
	c_5	51.05 ± 0.08	51.59 ± 0.08	49.70 ± 0.11	47.76 ± 0.21
	c_6*	50.03 ± 0.09	50.49 ± 0.07	49.03 ± 0.10	47.24 ± 0.09
	c_7*	50.11 ± 0.12	50.65 ± 0.22	49.02 ± 0.06	47.33 ± 0.21
	c_8	50.29 ± 0.06	50.77 ± 0.07	49.39 ± 0.12	47.88 ± 0.16
	c_9	50.18 ± 0.17	50.70 ± 0.15	49.05 ± 0.14	47.15 ± 0.11
2 sensors, 3 states	c_0	50.19 ± 0.23	50.81 ± 0.16	49.71 ± 0.08	47.79 ± 0.16
	c_1	50.83 ± 0.09	51.38 ± 0.09	49.97 ± 0.11	48.11 ± 0.17
	c_2	50.64 ± 0.50	51.64 ± 0.23	50.33 ± 0.16	48.75 ± 0.14
	c_3	50.77 ± 0.08	51.53 ± 0.11	49.75 ± 0.07	47.89 ± 0.15
	c_4*	50.92 ± 0.27	51.57 ± 0.07	49.78 ± 0.07	47.54 ± 0.12
	c_5	50.97 ± 0.17	51.80 ± 0.11	51.41 ± 0.09	50.07 ± 0.12
	c_6	49.98 ± 0.12	50.46 ± 0.08	49.69 ± 0.12	47.95 ± 0.26
	c_7*	50.16 ± 0.15	51.81 ± 0.07	49.98 ± 0.17	47.49 ± 0.15
	c_8	50.91 ± 0.22	50.30 ± 0.11	49.91 ± 0.12	48.31 ± 0.21
	c_9	49.82 ± 0.07	50.25 ± 0.18	49.15 ± 0.07	47.55 ± 0.11

(b) performance test

Table E.4 continued ...

control structure		env0	env1	env2	env3
4 sensors, 1 state	c_0	0.0±0.0	0.1±0.3	1.0±0.0	1.4±0.5
	c_1	0.0±0.0	1.1±0.3	1.1±0.7	3.5±0.8
	c_2	0.0±0.0	0.4±0.5	1.7±0.0	0.5±0.7
	c_3	0.0±0.0	0.1±0.3	1.0±0.0	4.9±0.3
	c_4*	0.0±0.0	0.0±0.0	1.0±0.0	0.6±0.5
	c_5	0.0±0.0	1.0±0.0	1.0±0.0	1.7±0.5
	c_6	0.0±0.0	0.0±0.0	1.0±0.0	1.5±0.5
	c_7	0.0±0.0	0.0±0.0	1.0±0.0	0.0±0.0
	c_8*	0.0±0.0	0.0±0.0	1.0±0.0	0.2±0.4
	c_9	0.0±0.0	0.2±0.4	1.0±0.0	0.9±0.3
	c_{10}	0.0±0.0	0.0±0.0	1.0±0.0	1.0±0.0
4 sensors, 2 states	c_0	0.0±0.0	0.0±0.0	2.1±0.3	0.1±0.3
	c_1	0.0±0.0	0.4±0.7	1.1±0.3	0.6±0.5
	c_2	0.0±0.0	0.0±0.0	3.8±0.7	5.1±0.9
	c_3	0.0±0.0	1.2±0.4	3.5±0.5	1.0±0.3
	c_4*	0.0±0.0	0.0±0.0	0.3±0.5	0.0±0.0
	c_5	0.0±0.0	1.3±0.5	2.4±0.8	1.0±1.0
	c_6*	0.0±0.0	0.1±0.3	1.1±0.8	1.0±0.0
	c_7	0.3±0.5	0.3±0.5	1.7±0.9	2.0±0.0
	c_8	0.0±0.0	0.1±0.3	0.2±0.4	1.0±0.0
	c_9	1.0±0.0	1.0±0.0	1.0±0.0	1.0±0.4
4 sensors, 3 states	c_0	0.0±0.0	0.0±0.0	2.1±0.3	0.3±0.5
	c_1	0.0±0.0	0.0±0.0	2.3±0.6	0.0±0.0
	c_2	0.0±0.0	0.0±0.0	0.3±0.5	2.4±0.5
	c_3	0.0±0.0	1.3±0.5	1.4±0.7	4.0±0.0
	c_4	0.0±0.0	1.0±0.9	1.9±0.7	1.9±0.7
	c_5	0.1±0.3	0.0±0.0	3.4±0.5	5.2±0.6
	c_6*	0.0±0.0	0.2±0.4	0.5±0.5	0.2±0.4
	c_7	0.0±0.0	0.1±0.3	2.2±0.4	0.6±0.5
	c_8	0.0±0.0	0.1±0.3	4.8±0.4	4.6±0.5
	c_9*	0.0±0.0	0.6±0.8	1.0±0.0	0.3±0.5

(a) collision test

Table E.5: Performance of various memory states with 4 sensors for corridor following

control structure		env0	env1	env2	env3
4 sensors, 1 state	c_0	50.67 ± 0.15	50.68 ± 0.07	49.79 ± 0.04	46.74 ± 0.11
	c_1	50.70 ± 0.13	50.91 ± 0.06	49.14 ± 0.15	46.70 ± 0.16
	c_2	50.19 ± 0.11	50.76 ± 0.08	49.63 ± 0.18	47.64 ± 0.27
	c_3	50.44 ± 0.09	50.63 ± 0.13	48.96 ± 0.05	46.50 ± 0.14
	c_4*	50.19 ± 0.15	50.73 ± 0.09	48.96 ± 0.05	46.97 ± 0.16
	c_5	50.14 ± 0.09	50.68 ± 0.12	48.92 ± 0.12	47.04 ± 0.18
	c_6	50.24 ± 0.04	50.70 ± 0.09	49.37 ± 0.12	47.01 ± 0.10
	c_7	50.33 ± 0.05	50.76 ± 0.05	49.15 ± 0.09	47.26 ± 0.14
	c_8*	50.19 ± 0.13	50.79 ± 0.09	49.33 ± 0.15	46.66 ± 0.22
	c_9	50.14 ± 0.20	50.73 ± 0.09	51.04 ± 0.42	48.02 ± 0.25
	c_{10}	50.10 ± 0.13	50.66 ± 0.12	49.19 ± 0.07	47.29 ± 0.16
4 sensors, 2 states	c_0	49.03 ± 0.14	49.11 ± 0.04	48.72 ± 0.33	46.41 ± 0.18
	c_1	49.08 ± 0.06	49.14 ± 0.06	49.75 ± 0.40	46.68 ± 0.22
	c_2	49.05 ± 0.10	49.11 ± 0.12	45.71 ± 0.18	45.96 ± 0.19
	c_3	49.03 ± 0.03	49.19 ± 0.12	47.71 ± 0.12	46.55 ± 0.56
	c_4*	49.14 ± 0.16	49.19 ± 0.09	48.02 ± 0.16	46.03 ± 0.14
	c_5	49.00 ± 0.00	49.04 ± 0.02	47.60 ± 0.13	45.04 ± 0.18
	c_6*	49.09 ± 0.08	49.09 ± 0.10	47.79 ± 0.15	44.98 ± 0.06
	c_7	49.13 ± 0.05	49.13 ± 0.04	47.80 ± 0.07	45.08 ± 0.14
	c_8	49.62 ± 0.05	49.75 ± 0.07	49.49 ± 0.19	46.17 ± 0.17
	c_9	49.41 ± 0.03	49.46 ± 0.08	48.05 ± 0.04	45.81 ± 0.14
4 sensors, 3 states	c_0	49.03 ± 0.07	49.13 ± 0.06	48.97 ± 0.35	46.58 ± 0.25
	c_1	49.02 ± 0.07	49.04 ± 0.04	47.82 ± 0.12	45.59 ± 0.21
	c_2	49.48 ± 0.09	49.58 ± 0.05	48.28 ± 0.05	45.75 ± 0.18
	c_3	49.07 ± 0.02	49.15 ± 0.14	47.75 ± 0.11	44.98 ± 0.12
	c_4	49.01 ± 0.07	49.07 ± 0.02	47.62 ± 0.08	45.57 ± 0.09
	c_5	49.93 ± 0.15	50.09 ± 0.08	48.56 ± 0.08	46.00 ± 0.11
	c_6*	49.10 ± 0.09	49.51 ± 0.06	48.80 ± 0.19	46.67 ± 0.29
	c_7	49.08 ± 0.03	49.35 ± 0.05	49.04 ± 0.08	45.97 ± 0.13
	c_8	48.06 ± 0.45	49.06 ± 0.02	49.97 ± 1.03	46.50 ± 0.91
	c_9*	49.02 ± 0.07	49.11 ± 0.05	47.62 ± 0.11	45.30 ± 0.22

(b) performance test

Table E.5 continued ...

control structure		env0	env1	env2	env3
2 sensors, neural	c_0	0.0 ± 0.0	0.9 ± 0.8	1.1 ± 0.3	0.0 ± 0.0
	c_1	0.0 ± 0.0	0.7 ± 0.5	4.0 ± 0.0	6.9 ± 0.9
	c_2	0.0 ± 0.0	1.0 ± 0.0	3.7 ± 1.1	4.8 ± 1.0
	c_3	0.0 ± 0.0	1.8 ± 0.6	3.9 ± 0.3	8.7 ± 1.3
	c_4*	0.0 ± 0.0	1.2 ± 0.4	0.7 ± 0.6	0.5 ± 0.5
	c_5	0.2 ± 0.4	1.2 ± 0.4	7.0 ± 0.0	8.2 ± 1.1
	c_6	0.0 ± 0.0	1.5 ± 0.7	7.5 ± 1.0	4.6 ± 0.5
	c_7	0.0 ± 0.0	0.5 ± 0.5	1.0 ± 0.0	4.4 ± 0.5
	c_8	0.0 ± 0.0	1.2 ± 0.6	1.7 ± 0.6	0.9 ± 0.3
	c_9*	0.0 ± 0.0	0.0 ± 0.0	1.5 ± 0.5	0.0 ± 0.0
4 sensors, neural	c_0	0.0 ± 0.0	0.0 ± 0.0	1.0 ± 0.0	0.0 ± 0.0
	c_1	0.0 ± 0.0	1.0 ± 0.0	3.9 ± 0.7	2.5 ± 0.9
	c_2*	0.0 ± 0.0	0.0 ± 0.0	0.7 ± 0.5	1.0 ± 0.0
	c_3	0.0 ± 0.0	1.0 ± 0.0	1.1 ± 0.3	7.4 ± 0.5
	c_4	0.0 ± 0.0	0.0 ± 0.0	1.2 ± 0.4	0.8 ± 0.4
	c_5*	0.0 ± 0.0	0.0 ± 0.0	1.0 ± 0.0	0.0 ± 0.0
	c_6	0.0 ± 0.0	0.0 ± 0.0	1.0 ± 0.0	0.0 ± 0.0
	c_7	0.0 ± 0.0	0.1 ± 0.3	2.0 ± 0.0	0.5 ± 0.5
	c_8	0.0 ± 0.0	1.9 ± 0.3	2.7 ± 0.5	4.8 ± 0.4
	c_9*	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	1.0 ± 0.0

(a) collision test

control structure		env0	env1	env2	env3
2 sensors, neural	c_0	49.00 ± 0.00	49.55 ± 0.23	51.43 ± 0.40	48.23 ± 0.24
	c_1	49.00 ± 0.01	49.00 ± 0.01	48.31 ± 0.06	46.02 ± 0.20
	c_2	49.01 ± 0.01	49.06 ± 0.02	47.85 ± 0.05	46.20 ± 0.18
	c_3	49.10 ± 0.29	49.93 ± 0.45	48.70 ± 0.04	48.00 ± 0.53
	c_4*	49.54 ± 0.05	49.55 ± 0.06	47.96 ± 0.07	45.37 ± 0.18
	c_5	49.71 ± 0.11	49.03 ± 0.04	51.22 ± 0.09	49.88 ± 0.13
	c_6	49.01 ± 0.04	49.05 ± 0.04	50.77 ± 0.59	48.41 ± 0.87
	c_7	49.00 ± 0.00	49.79 ± 0.52	47.25 ± 0.01	44.75 ± 0.09
	c_8	49.04 ± 0.08	50.14 ± 0.23	65.17 ± 1.16	52.37 ± 0.92
	c_9*	49.76 ± 0.08	49.82 ± 0.06	48.36 ± 0.16	45.85 ± 0.15
4 sensors, neural	c_0	49.00 ± 0.00	50.80 ± 0.33	68.78 ± 0.87	51.92 ± 0.69
	c_1	49.00 ± 0.00	49.09 ± 0.09	47.79 ± 0.09	46.61 ± 0.09
	c_2*	49.00 ± 0.00	49.47 ± 0.00	46.61 ± 0.07	43.70 ± 0.04
	c_3	49.00 ± 0.00	49.16 ± 0.07	48.13 ± 0.24	45.06 ± 0.16
	c_4	49.00 ± 0.00	49.14 ± 0.23	49.86 ± 0.11	49.08 ± 0.25
	c_5	49.00 ± 0.00	49.02 ± 0.03	46.34 ± 0.12	42.83 ± 0.18
	c_6	49.00 ± 0.00	49.13 ± 0.06	47.50 ± 0.02	45.62 ± 0.09
	c_7	49.00 ± 0.00	51.16 ± 0.19	46.00 ± 0.04	50.93 ± 0.96
	c_8	49.00 ± 0.00	49.01 ± 0.03	46.07 ± 0.01	42.77 ± 0.05
	c_9*	49.00 ± 0.00	49.00 ± 0.01	47.50 ± 0.04	44.44 ± 0.39

(b) performance test

Table E.6: Performance with neural networks for corridor following

control structure		env0	env1
2 infrared sensors, 1 state	c_0*	0.2±0.4	1.8±1.2
	c_1	0.1±0.3	3.9±1.7
	c_2	0.0±0.0	2.1±0.2
	c_3	0.2±0.4	2.6±1.3
	c_4	0.1±0.3	3.0±1.7
	c_5*	0.1±0.3	1.8±1.5
	c_6	0.1±0.3	1.9±1.0
	c_7	0.0±0.0	2.8±1.7
	c_8	0.0±0.0	2.6±1.5
	c_9*	0.1±0.3	2.1±1.5
2 infrared sensors, 2 states	c_0	0.0±0.0	2.8±1.8
	c_1*	0.0±0.0	1.2±1.1
	c_2	0.1±0.3	3.4±1.6
	c_3	0.3±0.5	2.8±1.2
	c_4	0.0±0.0	1.8±0.6
	c_5	0.0±0.0	1.2±0.7
	c_6*	0.0±0.0	1.8±0.9
	c_7	0.0±0.0	4.1±1.7
	c_8*	0.0±0.0	2.6±1.3
	c_9	0.0±0.0	3.7±1.1
2 infrared sensors, 3 states	c_0	0.0±0.0	2.1±1.0
	c_1*	0.0±0.0	1.3±0.9
	c_2	0.6±0.7	4.1±2.1
	c_3	0.1±0.3	3.4±0.9
	c_4	0.1±0.3	4.0±1.2
	c_5	0.4±0.5	4.6±1.6
	c_6*	0.0±0.0	2.2±1.2
	c_7	0.1±0.3	2.3±1.3
	c_8*	0.0±0.0	1.8±1.0
	c_9	0.0±0.0	2.0±1.1
4 infrared sensors, 1 state	c_0	0.1±0.3	2.6±1.0
	c_1*	0.0±0.0	0.7±0.8
	c_2*	0.0±0.0	1.2±0.6
	c_3	0.0±0.0	2.6±1.4
	c_4	0.0±0.0	0.5±0.7
	c_5	0.1±0.3	1.5±1.3
	c_6	0.4±0.5	3.9±1.4
	c_7	0.0±0.0	1.3±0.9
	c_8*	0.0±0.0	1.1±0.8
	c_9	0.0±0.0	2.3±1.0
4 infrared sensors, 2 states	c_0*	0.2±0.4	2.9±1.4
	c_1	0.1±0.3	1.6±1.1
	c_2*	0.0±0.0	0.0±0.0
	c_3	0.0±0.0	2.1±1.4
	c_4*	0.0±0.0	0.4±0.5
	c_5	0.1±0.3	0.6±0.7
	c_6	0.2±0.4	2.5±1.7
	c_7	0.1±0.3	2.8±0.9
	c_8	0.1±0.3	0.6±0.5
	c_9	0.0±0.0	0.2±0.4

(a) failure test

control structure		env0	env1
2 infrared sensors, 1 state	c_0*	56.57±7.08	127.31±5.39
	c_1	64.96±10.72	130.60±9.62
	c_2	66.29±9.78	135.44±6.70
	c_3	68.00±11.99	128.60±8.48
	c_4	55.96±6.26	138.26±7.39
	c_5*	62.29±2.91	134.71±7.48
	c_6	67.54±9.60	149.03±5.77
	c_7	61.87±7.19	144.43±3.40
	c_8	60.16±7.08	135.35±6.21
	c_9*	59.77±5.62	134.56±9.35
2 infared sensors, 2 states	c_0	66.08±7.98	143.51±7.29
	c_1*	67.66±6.86	145.93±6.55
	c_2	65.29±8.39	141.89±6.96
	c_3	59.00±5.00	133.50±6.63
	c_4	70.04±4.51	156.81±5.06
	c_5	70.61±5.36	162.57±7.40
	c_6*	58.07±1.78	131.03±8.24
	c_7	59.36±4.00	137.90±10.43
	c_8*	55.52±3.09	118.69±6.35
	c_9	54.63±1.11	132.96±6.39
2 infared sensors, 3 states	c_0	68.63±6.51	130.98±7.87
	c_1*	61.29±6.30	147.92±7.00
	c_2	71.97±10.51	145.86±7.13
	c_3	65.82±3.95	159.90±10.67
	c_4	59.17±3.56	127.79±4.27
	c_5	64.95±8.68	174.71±5.11
	c_6*	57.26±1.90	118.08±7.35
	c_7	63.14±7.44	140.33±6.01
	c_8*	60.63±2.77	132.92±6.55
	c_9	65.24±9.58	147.07±10.16
4 infared sensors, 1 state	c_0	75.42±5.02	141.32±9.25
	c_1*	75.08±3.09	162.60±4.86
	c_2*	62.36±3.03	147.59±4.87
	c_3	70.22±4.43	147.88±7.87
	c_4	73.96±1.25	172.38±3.25
	c_5	68.30±6.27	142.41±7.83
	c_6	54.94±5.38	121.46±8.53
	c_7	79.01±2.02	156.86±6.18
	c_8*	59.45±2.93	137.82±8.26
	c_9	64.52±3.82	151.74±4.95
4 infared sensors, 2 states	c_0*	64.69±5.40	123.49±6.11
	c_1	81.40±2.96	172.30±4.79
	c_2*	61.62±2.32	154.33±1.91
	c_3	85.32±7.24	172.17±8.50
	c_4*	59.06±6.19	138.13±7.81
	c_5	63.20±4.47	151.58±8.03
	c_6	69.35±7.19	148.77±9.90
	c_7	78.96±8.89	142.16±6.46
	c_8	72.68±5.19	172.45±6.79
	c_9	67.38±3.08	157.83±3.47

(b) performance test

Table E.7: Box pushing performance with various memory states over 2 infrared sensors and 4 infrared sensors (each column value represents the average score and standard deviation over 10 evaluations)

control structure		env0	env1
2 infrared sensors, 1 state	c_0	0.0±0.0	1.8±0.9
	c_1	0.1±0.3	2.2±1.8
	c_2	0.0±0.0	2.9±1.4
	c_3	0.0±0.0	2.3±1.4
	c_4	0.2±0.4	3.9±1.4
	c_5	0.1±0.3	2.4±1.2
	c_6*	0.0±0.0	2.1±0.9
	c_7*	0.0±0.0	1.1±1.0
	c_8	0.0±0.0	2.5±1.4
	c_9	0.0±0.0	1.5±0.9
2 infrared sensors, 2 states	c_0	0.0±0.0	1.4±1.6
	c_1*	0.0±0.0	0.1±0.3
	c_2	0.0±0.0	0.3±0.5
	c_3	0.0±0.0	0.9±0.9
	c_4	0.2±0.4	4.1±1.8
	c_5*	0.0±0.0	0.4±0.5
	c_6*	0.0±0.0	0.1±0.3
	c_7*	0.1±0.3	0.4±0.7
	c_8	0.1±0.3	1.6±1.6
	c_9	0.0±0.0	0.7±0.5
2 infrared sensors, 3 states	c_0	0.0±0.0	0.2±0.4
	c_1	0.0±0.0	0.5±0.6
	c_2	0.1±0.3	1.3±0.8
	c_3	0.0±0.0	0.8±0.7
	c_4*	0.0±0.0	0.3±0.5
	c_5	0.0±0.0	0.9±1.2
	c_6	0.0±0.0	0.7±0.5
	c_7	0.1±0.3	2.4±1.3
	c_8*	0.0±0.0	0.0±0.0
	c_9*	0.0±0.0	0.4±0.7
4 infrared sensors, 1 state	c_0	0.0±0.0	1.4±1.2
	c_1	0.1±0.3	1.7±1.5
	c_2	0.0±0.0	1.8±1.0
	c_3*	0.1±0.3	0.5±0.9
	c_4	0.0±0.0	1.6±1.1
	c_5	0.0±0.0	0.7±0.8
	c_6	0.0±0.0	0.1±0.3
	c_7*	0.0±0.0	0.4±0.5
	c_8	0.1±0.3	1.5±1.2
	c_9	0.2±0.4	2.4±1.4
4 infrared sensors, 2 states	c_0*	0.0±0.0	0.0±0.0
	c_1	0.0±0.0	0.3±0.5
	c_2	0.0±0.0	1.0±0.9
	c_3	0.0±0.0	0.0±0.0
	c_4	0.1±0.3	1.0±1.0
	c_5	0.4±0.5	1.5±0.7
	c_6*	0.1±0.3	0.2±0.4
	c_7	0.0±0.0	0.6±0.8
	c_8	0.0±0.0	1.0±0.9
	c_9	0.1±0.3	1.5±0.8

(a) failure test

control structure		env0	env1
2 infrared sensors, 1 state	c_0	73.63±5.31	179.85±3.93
	c_1	60.83±10.09	141.90±11.79
	c_2	69.43±9.82	152.72±11.82
	c_3	69.44±14.46	151.44±7.34
	c_4	83.94±17.27	160.77±13.09
	c_5	74.80±5.18	179.85±6.24
	c_6*	60.58±10.66	148.09±6.25
	c_7*	62.76±8.59	145.94±9.15
	c_8	65.30±8.16	147.28±11.02
	c_9	62.29±11.39	157.23±10.66
2 infared sensors, 2 states	c_0	58.78±5.56	139.02±5.84
	c_1*	50.85±2.43	123.87±3.07
	c_2	68.69±4.53	139.97±5.53
	c_3	71.85±10.66	155.81±6.62
	c_4	64.93±3.40	135.38±3.91
	c_5*	52.20±1.92	122.80±4.64
	c_6*	58.22±6.83	126.94±5.78
	c_7*	51.30±3.02	119.05±2.72
	c_8	55.29±5.85	141.34±5.06
	c_9	57.04±7.32	133.30±5.48
2 infared sensors, 3 states	c_0	62.93±6.91	147.76±6.27
	c_1	66.09±2.26	152.10±2.42
	c_2	61.34±5.70	153.70±6.55
	c_3	61.15±11.89	136.58±7.62
	c_4*	53.31±1.94	122.00±5.31
	c_5	55.33±4.15	144.10±5.35
	c_6	55.41±4.37	135.89±9.40
	c_7	60.25±9.58	127.90±5.71
	c_8*	54.57±1.92	138.83±2.74
	c_9*	56.17±6.86	126.40±2.65
4 infared sensors, 1 state	c_0	55.02±3.98	138.72±9.59
	c_1	60.83±7.25	130.68±6.63
	c_2	67.15±6.20	161.54±5.27
	c_3*	60.45±6.09	143.37±4.66
	c_4	72.75±5.07	156.03±5.34
	c_5	63.35±4.91	149.05±5.72
	c_6	61.51±2.15	159.45±2.76
	c_7*	51.90±1.74	123.87±4.98
	c_8	59.58±7.89	145.09±5.30
	c_9	65.12±9.33	153.64±7.30
4 infared sensors, 2 states	c_0*	53.04±2.40	121.18±4.39
	c_1	64.77±5.99	144.64±7.57
	c_2	61.50±7.45	142.71±4.01
	c_3	66.78±5.93	171.54±4.92
	c_4	70.04±7.36	154.91±3.70
	c_5	68.67±3.88	149.17±7.43
	c_6*	57.74±2.11	128.54±3.34
	c_7	76.28±11.58	162.08±6.42
	c_8	62.23±3.37	152.03±4.08
	c_9	58.12±1.96	137.03±5.47

(b) performance test

Table E.8: Box pushing performance with a small-sized box over 2 infrared sensors and 4 infrared sensors

control structure		env0	env1
4 infrared sensors, 3 states rule-based	c_0	0.1 ± 0.3	2.9 ± 1.1
	c_1	0.0 ± 0.0	2.5 ± 0.8
	c_2	0.1 ± 0.3	5.1 ± 1.9
	c_3	0.3 ± 0.5	3.4 ± 1.6
	c_4	0.1 ± 0.3	3.7 ± 1.3
	c_5*	0.1 ± 0.3	2.7 ± 1.7
	c_6*	0.0 ± 0.0	0.0 ± 0.0
4 infrared sensors, neural network	c_0*	0.0 ± 0.0	3.2 ± 1.2
	c_1	0.2 ± 0.4	4.7 ± 2.0
	c_2*	0.0 ± 0.0	0.2 ± 0.4
	c_3	0.6 ± 0.7	3.6 ± 1.3
	c_4	0.0 ± 0.0	2.1 ± 0.8
	c_5	0.0 ± 0.0	0.3 ± 0.5
	c_6	0.0 ± 0.0	8.8 ± 1.7

(a) failure test

control structure		env0	env1
4 infrared sensors, 3 states rule-based	c_0	83.15 ± 4.88	211.16 ± 2.57
	c_1	97.35 ± 4.10	173.61 ± 4.28
	c_2	80.63 ± 5.08	177.01 ± 8.06
	c_3	73.77 ± 8.46	197.95 ± 4.63
	c_4	97.26 ± 4.19	181.37 ± 5.31
	c_5*	86.31 ± 4.80	139.56 ± 3.76
	c_6*	75.05 ± 3.46	170.64 ± 1.60
4 infrared sensors, neural network	c_0*	76.73 ± 11.34	154.66 ± 7.21
	c_1	83.62 ± 5.22	167.76 ± 5.34
	c_2*	77.94 ± 5.52	162.25 ± 1.53
	c_3	78.67 ± 10.49	185.60 ± 7.71
	c_4	74.44 ± 8.12	167.53 ± 5.23
	c_5	82.60 ± 3.08	195.17 ± 2.08
	c_6	66.06 ± 3.75	157.00 ± 16.68

(b) performance test

Table E.9: Box pushing performance with rule-based structure and neural networks

control structure		env0	env1
4 infrared sensors, 3 states rule-based	c_0*	0.0 ± 0.0	0.9 ± 0.9
	c_1	0.3 ± 0.5	2.5 ± 1.8
	c_2	0.1 ± 0.3	1.8 ± 1.1
	c_3	0.1 ± 0.3	4.3 ± 1.6
	c_4	0.1 ± 0.3	0.8 ± 0.9
	c_5	0.0 ± 0.0	4.0 ± 1.2
	c_6*	0.0 ± 0.0	0.1 ± 0.3
4 infrared sensors, neural network	c_0*	0.0 ± 0.0	2.0 ± 1.5
	c_1*	0.0 ± 0.0	0.5 ± 0.5
	c_2	0.1 ± 0.3	2.8 ± 1.5
	c_3	0.0 ± 0.0	1.3 ± 0.8
	c_4	0.5 ± 0.5	3.8 ± 1.1
	c_5	0.0 ± 0.0	0.5 ± 0.5
	c_6	0.0 ± 0.0	1.8 ± 1.2

(a) failure test

control structure		env0	env1
4 infrared sensors, 3 states rule-based	c_0*	68.13 ± 2.81	156.53 ± 1.63
	c_1	67.57 ± 5.22	174.29 ± 6.52
	c_2	66.53 ± 4.34	161.13 ± 3.18
	c_3	60.24 ± 4.94	145.63 ± 3.20
	c_4	69.02 ± 1.71	183.66 ± 2.42
	c_5	71.00 ± 6.09	152.19 ± 2.85
	c_6*	54.35 ± 6.41	120.15 ± 2.30
4 infrared sensors, neural network	c_0*	71.03 ± 4.89	161.91 ± 4.85
	c_1*	67.58 ± 3.24	123.16 ± 4.14
	c_2	65.78 ± 6.00	162.32 ± 6.62
	c_3	69.33 ± 6.34	174.08 ± 7.56
	c_4	66.09 ± 3.60	146.19 ± 5.25
	c_5	83.59 ± 4.09	203.73 ± 2.22
	c_6	62.30 ± 4.75	168.44 ± 7.93

(b) performance test

Table E.10: Box pushing performance with rule-based structure and neural networks in small box experiments

E.2 Figures

- Figure E.1 - Box pushing behavior of purely reactive controllers with two sensors and a small box
- Figure E.2 - Box pushing behavior of two state controllers with two sensors and a small box
- Figure E.3 - Box pushing behavior of rule-based state controllers with a small box

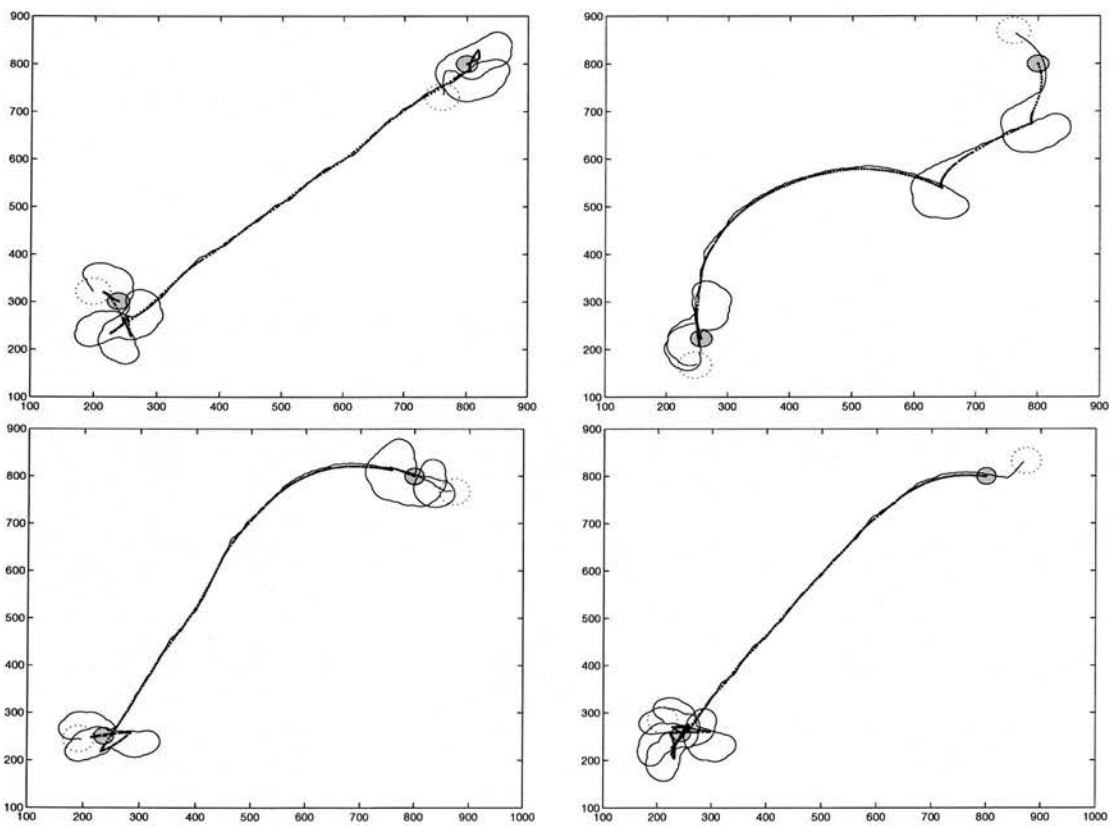


Figure E.1: Box pushing behavior of purely reactive controllers with two sensors and a small box (solid circle: box, dotted circle: robot)

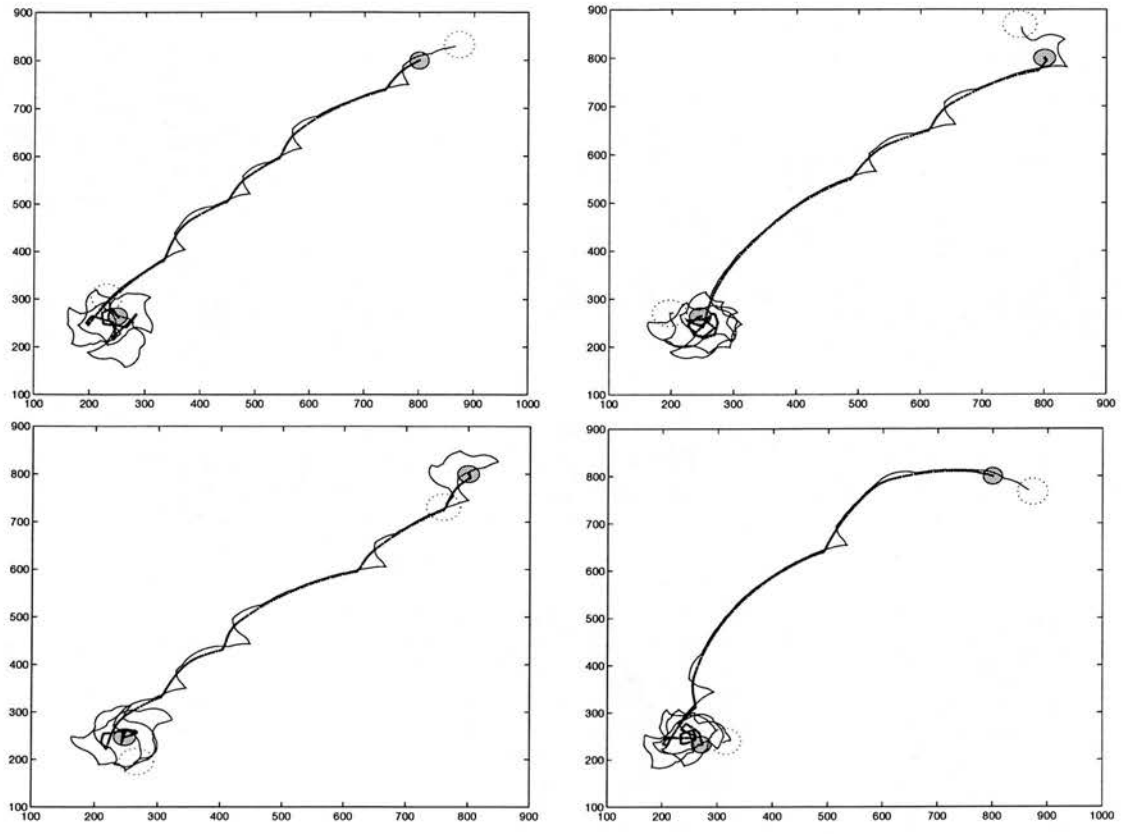


Figure E.2: Box pushing behavior of two state controllers with two sensors and a small box (solid circle: box, dotted circle: robot)

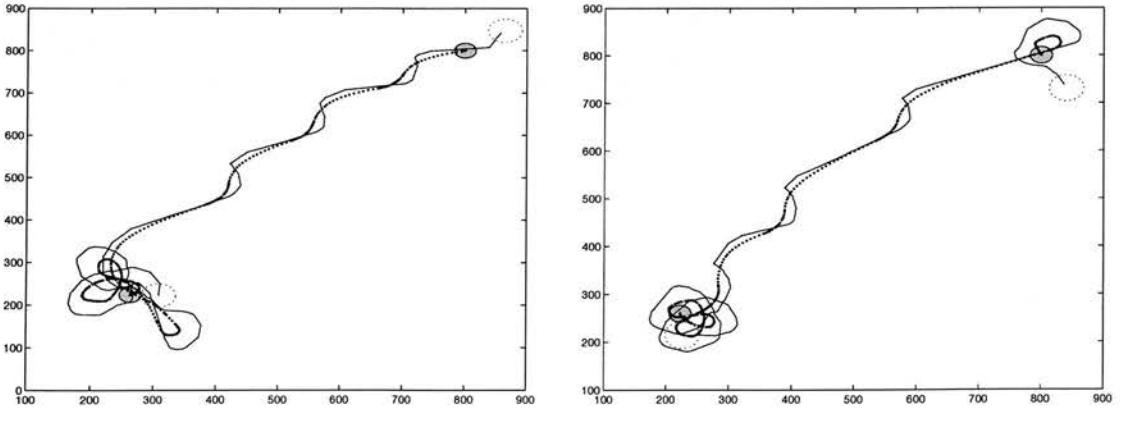


Figure E.3: Box pushing behavior of rule-based controllers with a small box (solid circle: box, dotted circle: robot)

Bibliography

P.J. Angeline. Multiple interacting programs: A representaiom for evolving complex behaviors. *Cybernetics and Systems*, 29(8):779–806, 1998.

R.C. Arkin. *Behavior-based Robotics*. MIT Press, Cambridge, MA, 1998.

R.C. Arkin and D. MacKenzie. Temporal coordination of perceptual algorithms for mobile robot navigation. *IEEE Transactions on Robotics and Automation*, Vol. 10, No.3:276–286, June 1994.

D. Ashlock. Gp-automata for dividing the dollar. In *Genetic Programming 97*, pages 18–26. MIT Press, 1997.

D. Ashlock. Isac lists, a different representation for program induction. In *Genetic Programming 98*, pages 3–10. Morgan Kauffman, 1998.

D. Ashlock, M.D. Smucker, and A.A. Stanley. Iterated prisoner's dilemma game with choice and refusal of partners. In *Advances in Artificial Life : European Conference on Artificial Life*, 1995.

T. Back and H.P. Schwefel. An overview of evolutionary algorithms for parameter optimization. *Evolutionary Computation*, 1, 1:1–23, 1993.

B. Bakker and M. de Jong. The epsilon state count. In *From Animals to Animats 6: Proceedings of the Sixth International Conference on Simulation of Adaptive Behaviour*, pages 51–60. MIT Press, 2000.

Karthik Balakrishnan and Vasant Honavar. On sensor evolution in robotics. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 455–460, Stanford University, CA, USA, 1996. MIT Press.

S. Baluja. Evolution of an artificial neural network based autonomous land vehicle controller. *IEEE Transactions on Systems, Man, and Cybernetics*, 26, 3:450–563, 1996.

E. B. Baum. Toward a model of intelligence as an economy of agents. *Machine Learning*, 35(2):155–185, 1999.

R.D. Beer and J.C. Gallagher. Evolving dynamic neural networks for adaptive behavior. *Adaptive Behavior*, 1(1):91–122, 1992.

R. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Autonomation*, Vol. RA-2, No.1:14–23, 1986.

R. Brooks. Intelligence without representation. In *Workshop on th Foundations of Artificial Intelligence*, Endicott House, Dedham Mass., 1987.

R.A. Brooks. Intelligence without reason. AI memo 1293, Artificial Intelligence Laboratory, MIT, 1991.

D. Chapman and L. P. Kaelbling. Learning from delayed reinforcement in a complex domain. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, 1991.

D. Cliff, P. Husbands, and I. Harvey. Evolving visually guided robots. In *From Animals to Animats 2: Proceedings of the Second International Conference on Simulation of Adaptive Behaviour*, pages 374–383. MIT Press, 1992.

D. Cliff and S. Ross. Adding temporary memory to ZCS. *Adaptive Behavior*, 3(2):101–150, 1995.

P. R. Cohen. *Empirical methods for artificial intelligence*. MIT Press, Cambridge, Mass., 1995.

M. Colombetti and M. Dorigo. Learning to control an autonomous robot by distributed genetic algorithm. In *From Animals to Animats 2: Proceedings of the Second International Conference on Simulation of Adaptive Behaviour*, pages 305–312. MIT Press, 1993.

M. Colombetti and M. Dorigo. Training agents to perform sequential behavior. *Adaptive Behavior*, 2(3):305–312, 1994.

M. Dorigo and M. Colombetti. *Robot Shaping: An Experiment in Behavior Engineering*. MIT Press/Bradford Books, 1998 edition, 1998.

G. Dudek and M. Jenkin. *Computational Principles of Mobile Robotics*. Cambridge University Press, Cambridge, UK, 2000.

P. Eggenberger. Cell interactions as a control tool of developmental processes for evolutionary robotics. In *From Animals to Animats 4: Proceedings of the Fourth International Conference on Simulation of Adaptive Behaviour*. MIT Press, 1996.

D. Floreano and F. Mondada. Automatic creation of an autonomous agent: Genetic evolution of a neural network driven robot. In *From Animals to Animats 3: Proceedings of the Third International Conference on Simulation of Adaptive Behaviour*. MIT Press, 1994.

D. Floreano and F. Mondada. Evolution of homing navigation in a real robot. *IEEE Transactions on Systems, Man and Cybernetics*, 26(3):396–407, 1996.

D.B. Fogel. *Evolutionary computation : toward a new philosophy of machine intelligence*. IEEE Press, New York, 1995.

L.J. Fogel, A.J. Owens, and M.J. Walsh. *Artificial intelligence through simulated evolution*. Wiley, New York, 1966.

C. M. Foncea and P. J. Fleming. Genetic algorithms for multiobjective optimization: Formulation, discussion and generalization. In S. Forrest, editor, *Proceedings of the Fifth Inter-*

national Conference on Genetic Algorithms, pages 416–423, San Mateo, California, 1993. Morgan Kaufmann.

J.C. Gallagher, R.D. Beer, K.S. Espenschield, and R.D. Quinn. Application of evolved locomotion controllers to a hexapod robot. *Robotics and Autonomous Systems*, 19:95–103, 1996.

S. Galt, B.L. Luk, and A.A. Collie. Evolution of smooth and efficient walking motions for an 8-legged robot. In *Proceedings of the 6th European Workshop on Learning Robots*, Brighton, UK, 1997.

E. Gat and G. Dorais. Robot navigation by conditional sequencing. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 1293–1299, 1994.

C. Gathercole. *An Investigation of Supervised Learning in Genetic Programming*. Ph. D. dissertation, University of Edinburgh, 1998.

C. Gathercole and P. Ross. Dynamic training subset selection for supervised learning in genetic programming. In Yuval Davidor, Hans-Paul Schwefel, and Reinhard Männer, editors, *Parallel Problem Solving from Nature III*, pages 312–321, Jerusalem, 1994. Springer-Verlag.

D. Goldberg. *Computer-Aided Gas Pipeline Operation using Genetic Algorithms and Rule Learning*. Ph. D. dissertation, University of Michigan, 1983.

D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison Wesly, Reading, MA, 1989.

T. Gomi and A. Griffith. Evolutionary robotics an overview. In *Proceedings of the IEEE 3rd International Conference on Evolutionary Computation*. IEEE Society Press, 1996.

J. Grefenstette and A. Schultz. An evolutionary approach to learning in robots. In *Machine Learning Workshop on Robot Learning*, New Brunswick, NJ, July, 1994.

F. Gruau. Automatic definition of modular neural networks. *Adaptive Behavior*, pages 151–183, 1995.

F. Gruau and K. Quartramaran. Cellular encoding for interactive evolutionary robotics. In *Proceedings of the Fourth European Conference on Artificial Life*, 1997.

I. Harvey. The SAGA cross: The mechanics of crossover for variable-length genetic algorithms. In R. Männer and B. Manderick, editors, *Parellel Problem Solving from Nature 2*, pages 269–278. North-Holland, 1992a.

I. Harvey. Species adaptation genetic algorithms: The basis for a continuous SAGA. In F. J. Varela and P. Bourgine, editors, *Proceedings of the First European Conference on Artificial Life*, pages 346–354, Cambridge, Mass., 1992b. MIT Press/Bradford Books.

I. Harvey, P. Husbands, and D. Cliff. Issues in evolutionary robotics. In *From Animals to Animats 2: Proceedings of the Second International Conference on Simulation of Adaptive Behaviour*, pages 364–373. MIT Press, 1992.

I. Harvey, P. Husbands, and D. Cliff. Seeing the light: Artificial evolution, real vision. In *From*

Animals to Animats 3: Proceedings of the Third International Conference on Simulation of Adaptive Behaviour, pages 392–401. MIT Press, 1994.

I. Harvey, P. Husbands, D. Cliff, A. Thompson, and N. Jakobi. Evolutionary robotics at sussex. In *Proceedings of ISRAM96, International Symposium on Robotics and Manufacturing*, pages 392–401, Montpellier, France, 1996.

J. Haugeland. *Artificial Intelligence: The very idea.* MIT Press, Cambridge, Mass., 1985 edition, 1985.

S. Haykin. *Neural Networks : a comprehensive foundation.* MacMillan Publishing Company, New York, 1994.

John H. Holland. *Adaptation in Natural and Artificial Systems.* University of Michigan Press, Ann Arbor, MI, 1975.

J.E. Hopcroft and J.D. Ullman. *Introduction to automata theory, languages, and computation.* Addison Wesley, Reading, MA, 1979.

J.J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences*, 79:2554–2558, 1982.

K. Hornik, M. Stichcombe, and H. White. Multi-layer feed-forward networks are universal approximators. *Neural Networks*, 3:359–366, 1989.

A. Ijspeert. *Design of artificial neural oscillatory circuits for the control of lamprey- and salamander-like locomotion using evolutionary algorithms.* Ph. D. dissertation, University of Edinburgh, 1998.

A.J. Ijspeert, J. Hallam, and D. Willshaw. Comparing naturally and artificially evolved swimming controllers. In *Proceedings of the Fourth European Conference on Artificial Life*, pages 256–265. MIT Press/Bradford Books, 1997.

N. Jakobi. Evolutionary robotics and the radical envelope of noise hyperthesis. *Adaptive Behavior*, 6, 1:131–174, 1997.

N. Jakobi. *Minimal Simulations for Evolutionary Robotics.* Ph. D. dissertation, School of Cognitive and Computing Sciences, University of Sussex, 1998.

N. Jakobi, P. Husbands, and I. Harvey. Noise and the reality gap: The use of simultation in evolutionary robotics. In *Advances in Artificial Life: Proceedings of the Third International Conference on Artificial Life, Moran, F. Moreno, A., Merelo, J. Chacon, P. (eds.).* Springer-Verlag, 1995.

J.-S. R. Jang, C.-T. Sun, and E. Mizutani. *Neuro-Fuzzy and Soft Computing.* Prentice-Hall, Inc., Upper Saddle River, NJ, 1997.

D. Jefferson and R. Collins. Evolution as a theme in artificial life. In C. Langton, editor, *Artificial Life II*, 1991.

L. P. Kaelbling. An architecture for intelligent reactive systems. In *Reasoning about Actions and Plans: Proceedings of the 1986 Workshop*, Endicott House, Dedham Mass., 1986.

Stuart A. Kauffman. *The Orgins of Order : self-organization and selection in evolution.* Oxford University Press, New York, 1993.

D. Kim and J. Hallam. Neural network approach to path integration for homing navigation. In *From Animals to Animats 6, Proceedings of the International Conference on the Simulation of Adaptive Behavior*, pages 228–235. MIT Press, 2000.

D. Kim and J. Hallam. Mobile robot control based on boolean logic with internal memory. In *Advances in Artificial Life, Lecture Notes in Computer Science vol. 2159 : European Conference on Artificial Life*, pages 529–538, 2001.

Zvi Kohavi. *Switching and Finite Automata Theory.* McGraw-Hill, New York, London, 1970.

John R. Koza. *Genetic Programming.* MIT Press, Cambridge, MA, 1992.

C.R. Kube and H. Zhang. Collective robotics: From social insects to robots. *Adaptive Behavior*, 2(2):189–219, 1993.

B. C. Kuo. *Automatic Control Systems.* Prentice-Hall, Englewood Cliffs, New Jersey, 6th edition, 1991.

D. Lambrinos. Navigating with an adaptive light compass. In *Proceedings of the Third European Conference on Artificial Life*, 1995.

P. L. Lanzi. An analysis of the memory mechanism of XCSM. In *Genetic Programming 98*, pages 643–651. Morgan Kauffman, 1998.

P.L. Lanzi. Adaptive agents with reinforcement learning and internal memory. In *From Animals to Animats 6: Proceedings of the Sixth International Conference on Simulation of Adaptive Behaviour*, pages 333–342. MIT Press, 2000.

W-P. Lee, J. Hallam, and H.H. Lund. A hybrid gp/ga approach for co-evolving controllers and robot bodies to achieve fitness-specified tasks. In *Proceedings of IEEE International Conference on Evolutionary Computation*, Nagoya, Japan, 1996.

W-P. Lee, J. Hallam, and H.H. Lund. Applying genetic programming to evolve behavior primitives and arbitrators for mobile robots. In *Proceedings of IEEE International Conference on Evolutionary Computation*, Indianapolis, USA, 1997.

Wei-Po Lee. *Applying Genetic Programming to Evolve Behavior Primitives and Arbitrators for Mobile Robots.* Ph. D. dissertation, University of Edinburgh, 1998.

M. Lewis, A. Fagg, and G. Bekey. Genetic algorithms for gait synthesis in a hexapod robot. In *Recent Trends in Mobile Robots*, pages 317–331. World Scientific, Singapore, 1994.

L. Lin and T. M. Mitchell. Reinforcement learning with hidden states. In *From Animals to Animats 2: Proceedings of the Second International Conference on Simulation of Adaptive Behaviour*, pages 271–280. MIT Press, 1992.

M. Littman. Memoryless policies: theoretical limitations and practical results. In *From Animals to Animats 3: Proceedings of the Third International Conference on Simulation of Adaptive Behaviour*, pages 238–245. MIT Press, 1994.

P. Maes and R. Brooks. Learning to coordinate behaviors. In *Proceedings of the Eighth National Conference on Artificial Intelligence(AAAI'90)*, pages 796–802. Boston, MA, August, 1990.

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

M. Mataric and D. Cliff. Challenges in evolving controllers for physical robots. In *Evolutionary Robotics, special issue of Robotics and Autonomous Systems*, 1995.

G. Mayley. The evolutionary cost of learning. In *From Animals to Animats 4: Proceedings of the Fourth International Conference on Simulation of Adaptive Behaviour*. MIT Press, 1996.

A. K. McCallum. *Reinforcemnet Learning with Selective Perception and Hidden State*. Ph. D. dissertation, University of Rochester, 1996.

R. A. McCallum. Overcoming incomplete perception with utile ditinction memory. In *Proceedings of the 10th International Machine Learning Conference*. Morgan Kauffman, 1993.

L.A. Meeden. An incremental approach to developing intelligent neural network controllers for robots. *IEEE Transactions on Systems, Man, and Cybernetics*, 26, 3:474–485, 1996.

J.A. Meyer, P. Husbands, and I. Harvey. Evolutionary robotics: A survey of applications and problems. In *First European Workshop on Evolutionary Robotics*, Paris, France, 1998. Springer.

O. Michel. An artificial life approach for the synthesis of autonomous agents. In *Artificial Evolution*. Springer, 1996.

O. Miglino, H.H. Lund, and S. Nolfi. Evolving mobile robots in simulated and real environments. *Artificial Life*, 2:417–434, 1995.

J.H. Miller. The evolution of automata in the repeated prisoner's dilemma. In *working paper*, Santa Fe Institue, NM, 1989.

Melanie Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, 1996.

F. Mondada, E. Franzi, and P. Ienne. Mobile robot miniaturisation: A tool for invesitigation in control algorithms. In *Proceedings of the Third International Symposium on Experimental Robotics*. Kyoto, Japan, 1993.

J. Moody and C. Darken. Fast learning in networks of locally-tuned processing units. *Neural Computation*, 1:281–294, 1989.

U. Nehmzow and B. McGonigle. Achieving rapid adaptations in robots by means of external tuition. In *From Animals to Animats 3: Proceedings of the Third International Conference on Simulation of Adaptive Behaviour*. MIT Press, 1994.

U. Nehmzow and T. Smithers. Mapbuilding using self-organizing networks in really useful robots. In *From Animals to Animats 1: Proceedings of the First International Conference on Simulation of Adaptive Behaviour*. MIT Press, 1991.

N. Nilsson. A mobile automaton: An application of artificial intelligence techniques. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 233–244, 1969.

N. Nilsson. Teleo-reactive programs for agent control. *Journal of Artificial Intelligent Research*, pages 139–158, 1994.

S. Nolfi. Evolving non-trivial behavior on autonomous robots: Adaptation is more powerful than decomposition and integration. In *Evolutionary Robotics, From Intelligent Robots to Artificial Life*, 1997a.

S. Nolfi. Evolving non-trivial behaviors on real robots: a garbage collecting robot. *Robotics and Autonomous Systems*, 1997b.

S. Nolfi and D. Floreano. *Evolutionary Robotics : The Biology, Intelligence, and Technology of Self-Organizing Machines*. MIT Press, Cambridge, MA, 2000.

S. Nolfi and D. Parisi. Evolving non-trivial behaviors on real robots: An autonomous robot that picks up objects. In *Proceedings of the Fourth Congress of the Italian Association for Artificial Intelligence*. Springer-Verlag, 1995.

P. Nordin and W. Banzhaf. Genetic programming controlling a miniature robot. In E. V. Siegel and J. R. Koza, editors, *Working Notes for the AAAI Symposium on Genetic Programming*, pages 61–67, MIT, Cambridge, MA, USA, 1995. AAAI.

P. Nordin and W. Banzhaf. An on-line method to evolve behavior and to control a miniature robot in real time with genetic programming. *Adaptive Behavior*, 5, 2:107–140, 1996.

P. Nordin and W. Banzhaf. Real time control of a khepera robot using genetic programming. *Cybernetics and Control*, 26 (3):533–561, 1997.

C.W. Omlin and C.L. Giles. Extraction of rules from discrete-time recurrent neural networks. *Neural Networks*, 9(1):41–52, 1996.

C.H. Paradimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithmsa and Complexity*. Prentice-Hall, Englewood Cliffs, New Jersey, 1982.

R. Pfeifer and C. Scheier. *Understanding intelligence*. MIT Press, Cambridge, Mass., 1999.

M. J. D. Powell. Radial basis functions for multivariable interpolation: a review. In J. C. Mason and M. G. Cox, editors, *Algorithms for approximation*, pages 143–167. Oxford University Press, 1987.

A. Ram, R. Arkin, G. Boone, and M. Pearce. Using genetic algorithms to learn reactive control parameters for autonomous robot navigation. *Adaptive Behavior*, 2(3):277–305, 1994.

C. Reynolds. Evolution of corridor following in a noisy world. In *From Animals to Animats 3: Proceedings of the Third International Conference on Simulation of Adaptive Behaviour*, pages 402–410. MIT Press, 1994.

J. Rissanen. Modeling by shortest data description. *Automatica*, 14:465–471, 1978.

J. Rissanen. *Stochastic Complexity in Statistical Inquiry*. World Scientific, Singapore, 1989.

S. M. Ross. *Introduction to Probability and Statistics for Engineers and Scientists*. Academic Press, San Diego, CA, 2nd edition, 2000.

R. Salomon. Increasing adaptivity through evolution strategies. In *From Animals to Animats 4: Proceedings of the Fourth International Conference on Simulation of Adaptive Behaviour*. MIT Press, 1996.

J.D. Schaffer. Multiple objective optimization with vector evaluated genetic algorithms. In *Proceedings of the International Conference on Genetic Algorith,s and their Applications*, pages 93–100, 1985.

C. Scheier and D. Lambrinos. Adaptive classification in autonomous agents. In *Proceedings of European Meeting on Cybernetics and Systems Research*, 1995.

A. Silva, A. Neves, and E. Costa. Genetically programming networks to evolve memory mechanism. In *Proceedings of Genetic and Evolutionary Computation Conference*, 1999.

H. A. Simon. *The Sciences of the Artificial*. MIT Press, Cambridge, Mass., 1985 edition, 1981.

K. Sims. Evolving virtual creatures. In *Proceedings of the ACM Siggraph*, pages 25–39, 1994.

A. P. Singh, T. Jaakkola, and M. I. Jordan. Model-free reinforcement learning for non-Markovian decision problems. In *Proceedings of the 11th International Machine Learning Conference*. Morgan Kauffman, 1994.

T. Smith. *Adding Vision to Khepera: An Autonomous Robot Footballer*. Master's thesis, School of Cognitive and Computing Sciences, University of Sussex, 1997.

W. Stolzmann. An introduction to anticipatory classifier systems. In *Learning Classifier Systems, Lecture Notes in Computer Science vol. 1813*, pages 175–194, 2000.

R. S. Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Proceedings of the Machine Learning Conference*, 1990.

R. S. Sutton. Planning by incremental dynamic programming. In *Proceedings of the Eighth International Workshop on Machine Learning*, pages 353–357, 1991.

R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, Mass., 1998.

A. Teller. The evolution of mental models. In *Advances in Genetic Programming*. MIT Press, 1994.

A. Thompson. Evolving electronic robot controllers taht exploit hardware resources. In *Advances in Artificial Life: Proceedings of the Third International Conference on Artificial Life*, pages 640–656. Springer-Verlag, 1995.

Karl von Frisch. *The Dance Language and Orientation of Bees*. Belknap Press of Harvard University Press, Cambridge, 1967.

S. D. Whitehead. *Reinforcemnet Learning for the Adaptive Control of Perception and Action*. Ph. D. dissertation, University of Rochester, 1992.

S. D. Whitehead and D. H. Ballad. Learning to perceive and act by trial and error. *Machine Learning*, 7:45–83, 1991.

S. W. Wilson. ZCS: A zeroth level classifier system. *Evolutionary Computation*, 2(1):1–18, 1994.

S. W. Wilson. Classifier fitness based on accuracy. *Evolutionary Computation*, 3(2):149–175, 1995.

B. Yamauchi. Dynamical neural networks for mobile robot control. Memorandum Report AIC-033-93, Naval Research Laboratory, Washington, 1993.

B. Yamauchi and R. Beer. Integrating reactive, sequential, and learning behavior using dynamical neural networks. In *From Animals to Animats 3: Proceedings of the Third International Conference on Simulation of Adaptive Behaviour*, pages 382–391. MIT Press, 1994a.

B. Yamauchi and R. Beer. Sequential behavior and learning in evolved dynamical neural networks. *Adaptive Behavior*, 2(3):219–246, 1994b.

L. A. Zadeh. Fuzzy sets. *Information and Control*, 8:338–353, 1965.