

การปรับปรุงความทนทานของโปรแกรมหุ่นยนต์



นาย วรเศรษฐ สุวรรณิก

สถาบันวิทยบริการ

จุฬาลงกรณ์มหาวิทยาลัย

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตรดุษฎีบัณฑิต

สาขาวิชาวิศวกรรมคอมพิวเตอร์ ภาควิชาวิศวกรรมคอมพิวเตอร์

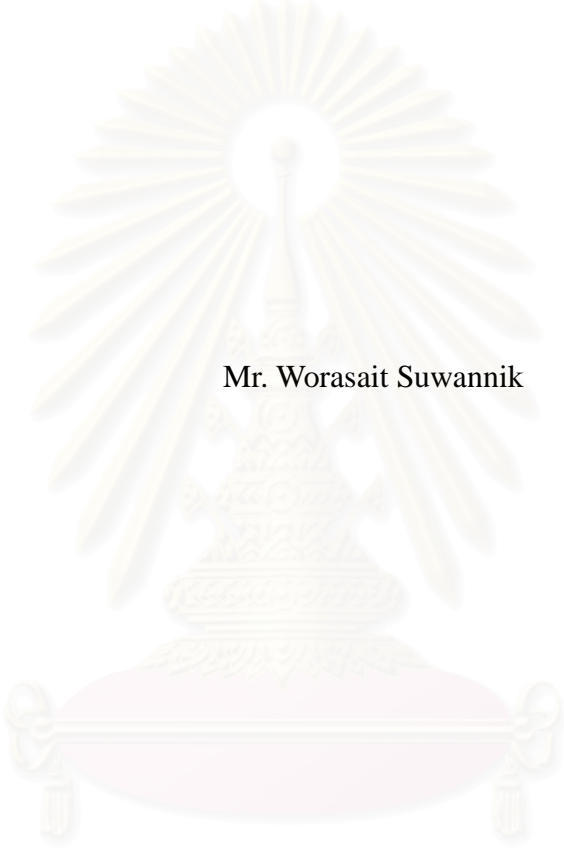
คณะวิศวกรรมศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย

ปีการศึกษา 2546

ISBN : 974-17-3944-3

ลิขสิทธิ์ของจุฬาลงกรณ์มหาวิทยาลัย

IMPROVING THE ROBUSTNESS OF EVOLVED ROBOT PROGRAMS



Mr. Worasait Suwannik

สถาบันวิทยบริการ  
จุฬาลงกรณ์มหาวิทยาลัย  
A Dissertation Submitted in Partial Fulfillment of the Requirements  
for the Degree of Doctor of Engineering in Computer Engineering

Department of Computer Engineering

Faculty of Engineering

Chulalongkorn University

Academic Year 2003

ISBN 974-17-3944-3

Thesis title      IMPROVING THE ROBUSTNESS OF EVOLVED ROBOT PROGRAMS  
By                      Worasait Suwannik  
Field of Study      Computer Engineering  
Thesis Advisor     Associate Professor Prabhas Chongstitvatana, Ph.D.

---

Accepted by the Faculty of Engineering, Chulalongkorn University in Partial Fulfillment of the Requirements for the Doctor's Degree

..... Dean of Faculty of Engineering  
(Professor Somsak Panyakaew, Ph.D.)

#### THESIS COMMITTEE

..... Chairman  
(Assistant Professor Boonserm Kijisirikul , Ph.D.)

..... Thesis Advisor  
(Associate Professor Prabhas Chongstitvatana, Ph.D.)

..... Member  
(Arthit Thongtak, Ph.D.)

..... Member  
(Attawith Sudsang, Ph.D.)

..... Member  
(Associate Professor Manukid Parnichkun, Ph.D.)

วเรศรชลุ สุวรรณิก : การปรับปรุงความทนทานของโปรแกรมหุ่นยนต์ (Improving the Robustness of Evolved Robot Programs). อาจารย์ที่ปรึกษา : รศ. ดร. ประภาศ จงสถิตย์วัฒนา, 102 หน้า. ISBN 974-17-3944-3

วิทยานิพนธ์ฉบับนี้อธิบายการวิวัฒนาการโปรแกรมควบคุมแขนหุ่นยนต์สำหรับการเอื้อมจับ โปรแกรมหุ่นยนต์ถูกวิวัฒนาการโดยใช้กำหนดการเชิงพันธุกรรม การวิวัฒนาการเกิดขึ้นในโลกจำลองและโลกจริง เนื่องจากความไม่แน่นอนในโลกจริง โปรแกรมหุ่นยนต์ไม่สามารถทำงานได้อย่างทนทานเมื่อโอนไปให้กับหุ่นยนต์จริง ทั้งนี้เนื่องจากความแตกต่างระหว่างโลกจำลองและโลกจริง วิธีการปรับปรุงความทนทานของโปรแกรมหุ่นยนต์ที่จะใช้บนหุ่นยนต์จริงได้ถูกนำเสนอสองวิธี วิธีแรกใช้หลายคอนฟิกรูเรชันในการวิวัฒนาการแบบไม่เชื่อมตรงกับหุ่นยนต์ วิธีที่สองวิวัฒนาการโปรแกรมแบบเชื่อมตรงกับหุ่นยนต์ การวิวัฒนาการแบบเชื่อมตรงไม่ต้องการแบบจำลองแต่ใช้เวลามหาศาลเพราะถูกจำกัดด้วยกลไกการทำงานของหุ่นยนต์ เพื่อที่จะเร่งเวลาการวิวัฒนาการ เราเสนอการใช้ฟังก์ชันเมมโมไอซ์เพื่อจำลองของการเคลื่อนที่ของหุ่นยนต์ เมื่อเปรียบเทียบกับการวิวัฒนาการแบบไม่เชื่อมตรงแล้ว ความทนทานของโปรแกรมที่วิวัฒนาการจากหลายคอนฟิกรูเรชันเพิ่มขึ้น 11% การวิวัฒนาการแบบเชื่อมตรงเพิ่มความทนทานถึง 27%

สถาบันวิทยบริการ  
จุฬาลงกรณ์มหาวิทยาลัย

ภาควิชา	วิศวกรรมคอมพิวเตอร์	ลายมือชื่อนิสิต .....
สาขาวิชา	วิศวกรรมคอมพิวเตอร์	ลายมือชื่ออาจารย์ที่ปรึกษา .....
ปีการศึกษา	2546	ลายมือชื่ออาจารย์ที่ปรึกษาร่วม .....

##4171821821 : MAJOR COMPUTER SCIENCE

KEYWORDS : ROBOT PROGRAM, GENETIC PROGRAMMING, ON-LINE LEARNING, ROBUSTNESS

WORASAIT SUWANNIK : IMPROVING THE ROBUSTNESS OF EVOLVED ROBOT PROGRAMS. THESIS ADVISOR : ASSOC. PROF. PRABHAS CHONGSTITVATANA, Ph.D., 102 pp. ISBN 974-17-3944-3

This dissertation describes an evolution of robot arm control programs for a visual-reaching task. The robot programs are evolved using Genetic Programming. The evolution process takes place in simulation and in the real world. Due to uncertainty in the real world, robot programs evolved in simulation are not working robustly when transferred to the real robot. This is caused by the difference between the simulation model and the real world. Two methods are proposed to improve the robustness of a robot program running on a real robot. First, multiple robot configurations are used during off-line evolution. Second, control programs are evolved on-line, that is the evolution procedure takes place in the real world on a real robot. On-line evolution does not require the model but it is very time consuming because of the mechanical speed of a physical robot. To accelerate on-line evolution, we propose a memoized function to store the effect of arm motions. Compared to a naive off-line evolution, the robustness of the program evolved off-line with multiple robot configurations is improved by 11%. The on-line evolution improves the robustness by 27%.

Department	Computer Engineering	Student's signature .....
Field of study	Computer Engineering	Advisor's signature .....
Academic year	2002	Co-advisor's signature .....

## ACKNOWLEDGEMENTS

I would like to express my gratitude to my advisor, Assoc. Prof. Prabhas Chongstitvatana, for his valuable advice and continuous support.



สถาบันวิทยบริการ  
จุฬาลงกรณ์มหาวิทยาลัย

## CONTENTS

	Page
<b>Abstract (Thai)</b> . . . . .	<b>iv</b>
<b>Abstract (English)</b> . . . . .	<b>v</b>
<b>Acknowledgements</b> . . . . .	<b>vi</b>
<b>Contents</b> . . . . .	<b>vii</b>
<b>List of Tables</b> . . . . .	<b>xii</b>
<b>List of Figures</b> . . . . .	<b>xiii</b>
Chapter	
<b>1 INTRODUCTION</b> . . . . .	<b>1</b>
1.1 Background . . . . .	1
1.2 Evolution of Robot Programs . . . . .	2
1.3 Improving the Robustness . . . . .	2
1.4 Dissertation Organization . . . . .	3
<b>2 CONTROL ARCHITECTURES</b> . . . . .	<b>5</b>
2.1 Configuration Space . . . . .	5
2.1.1 C-Space Transform . . . . .	6
2.1.2 Planning in C-Space . . . . .	6
2.1.3 Advantages . . . . .	9
2.1.4 Disadvantages . . . . .	9
2.2 Subsumption Architecture . . . . .	10
2.2.1 Task-Achieving Behavior . . . . .	10
2.2.2 Layering Behaviors . . . . .	10
2.2.3 Advantages . . . . .	11
2.2.4 Disadvantages . . . . .	12
2.3 Neuron Network Controller . . . . .	12
2.3.1 Network Architecture . . . . .	13

## CONTENTS (cont.)

	Page
2.3.2 Training the Network with Genetic Algorithm . . . . .	14
2.3.3 Advantages . . . . .	17
2.3.4 Disadvantages . . . . .	17
2.4 Genetic Programming Program . . . . .	18
2.4.1 Individual Program . . . . .	18
2.4.2 Fitness Measure . . . . .	20
2.4.3 Selection Method . . . . .	21
2.4.4 Genetic Operators . . . . .	21
2.4.5 Advantages . . . . .	22
2.4.6 Disadvantages . . . . .	22
2.5 Conclusion . . . . .	23
<b>3 ISSUES IN EVOLVING A ROBOT CONTROLLER . . . . .</b>	<b>24</b>
3.1 Genetic Programming Issues . . . . .	24
3.1.1 Initial Structure . . . . .	25
3.1.2 Fitness Function . . . . .	27
3.1.3 Fitness Cases . . . . .	28
3.1.4 Operation which Modify the Structure . . . . .	28
3.1.5 Selection Method . . . . .	30
3.1.6 Terminating Condition . . . . .	31
3.2 What To Evolve . . . . .	31
3.2.1 Evolving Robot Controller . . . . .	31
3.2.2 Evolving Robot Morphology . . . . .	33
3.3 Types of Evolution . . . . .	33
3.3.1 Off-line Evolution . . . . .	34
3.3.2 On-line Evolution . . . . .	34



## CONTENTS (cont.)

	Page
3.3.3 Hybrid On-line Off-line Evolution . . . . .	36
3.4 Miscellenous Issues . . . . .	36
3.4.1 Experimental Repeatability . . . . .	36
3.4.2 Judging Successful Behavior . . . . .	37
3.4.3 Incremental Evolution . . . . .	37
<b>4 RELATED WORKS ON IMPROVING ROBUSTNESS . . . . .</b>	<b>38</b>
4.1 Improving Robustness in Simulated Robots . . . . .	38
4.1.1 Mobile Robot . . . . .	39
4.1.2 Robot Arm . . . . .	39
4.2 Improving Robustness of Real Robots . . . . .	39
4.2.1 Capturing Data from the Real World . . . . .	40
4.2.2 Adding Noise . . . . .	41
4.2.3 Training with Multiple Trials . . . . .	41
4.2.4 Adaptive Control Architecture . . . . .	42
4.3 Conclusion . . . . .	43
<b>5 EVOLVING ROBOT ARM PROGRAMS FOR A TARGET REACH- ING TASK . . . . .</b>	<b>44</b>
5.1 The Robot Arm . . . . .	44
5.1.1 Joints and Links . . . . .	44
5.1.2 Vision . . . . .	46
5.2 Robot Task . . . . .	47
5.3 Evolving a Control Program . . . . .	48
5.3.1 Control Program . . . . .	48
5.3.2 Function Set . . . . .	48
5.3.3 Terminal Set . . . . .	49

## CONTENTS (cont.)

	Page
5.3.4 Fitness Function . . . . .	51
5.3.5 Genetic Parameters . . . . .	52
5.4 Result . . . . .	52
5.5 Conclusion . . . . .	52
<b>6 IMPROVING ROBUSTNESS OF ROBOT ARM PROGRAMS . . . . .</b>	<b>54</b>
6.1 Robustness . . . . .	55
6.1.1 Reality Gap . . . . .	55
6.1.2 Real World Reliability . . . . .	55
6.2 Improving the Robustness by Multiple Configurations . . . . .	56
6.2.1 Multiple Configurations . . . . .	57
6.2.2 Genetic Learning . . . . .	59
6.2.3 Experiment . . . . .	60
6.2.4 Result . . . . .	62
6.2.5 Conclusion . . . . .	63
6.3 On-line Evolution . . . . .	63
6.3.1 Memoized Function . . . . .	65
6.3.2 Genetic Learning . . . . .	66
6.3.3 Experiment . . . . .	67
6.3.4 Result . . . . .	68
6.3.5 Conclusion . . . . .	70
<b>7 COMPARING WITH REINFORCEMENT LEARNING . . . . .</b>	<b>72</b>
7.1 Q-learning . . . . .	73
7.2 Applying Q-Learning to Visual-Reaching Task . . . . .	74
7.2.1 A State Space and a Set of Actions . . . . .	75
7.2.2 Reward and Punishment . . . . .	75

## CONTENTS (cont.)

	Page
7.2.3 Discount Factor . . . . .	75
7.2.4 Exploration vs Exploitation . . . . .	76
7.2.5 When Learning is Completed . . . . .	76
7.2.6 Multi-step Q Learning . . . . .	76
7.3 Comparing Q-Learning with Genetic Programming . . . . .	77
7.3.1 Learning Time . . . . .	78
7.3.2 Quality of the result . . . . .	79
7.4 Conclusion . . . . .	80
<b>8 CONCLUSION . . . . .</b>	<b>81</b>
<b>References . . . . .</b>	<b>83</b>
<b>Biography . . . . .</b>	<b>89</b>

สถาบันวิทยบริการ  
จุฬาลงกรณ์มหาวิทยาลัย

## LIST OF TABLES

	Page
5.1 Terminal Set . . . . .	49
5.2 Function Set . . . . .	50
5.3 Genetic parameters . . . . .	52
5.4 Robustness of control programs in various maps . . . . .	53
6.1 Length of each link as seen from the vision system . . . . .	59
6.2 Genetic parameters . . . . .	61
6.3 Robustness of control programs in various maps . . . . .	63
6.4 Genetic parameters . . . . .	67
6.5 Evolution time . . . . .	68
6.6 Robustness of an evolved program . . . . .	70
7.1 Learning time of Q-learning and genetic programming. . . . .	78

สถาบันวิทยบริการ  
จุฬาลงกรณ์มหาวิทยาลัย

## LIST OF FIGURES

	Page
2.1 The configuration space for 2 DOF robot . . . . .	8
2.2 Avoid layer . . . . .	11
2.3 A neuron network controller is normally placed between robot sensors and actuators. . . . .	13
2.4 Sigmoid Function . . . . .	14
2.5 Flowchart for the genetic programming. . . . .	19
2.6 Tree for the expression $\sqrt{x^3}$ . . . . .	20
2.7 An example of mobile robot program. . . . .	20
2.8 Mutating a GP program. . . . .	21
2.9 Recombination of two GP programs. . . . .	23
5.1 An overview of the experimental platform. . . . .	44
5.2 A robot arm seen from its vision . . . . .	45
5.3 Five environments for testing the visual-reaching task. . . . .	47
5.4 An If function . . . . .	50
5.5 An IfAnd function . . . . .	50
6.1 Recall Rate vs Speed up of a memoize function. . . . .	69
7.1 Updating a Q value . . . . .	77
7.2 Trajectory of a robot arm controlled by a policy given by Q learning. . . . .	79
7.3 Trajectory of a robot arm controlled by a GP program. . . . .	80

# CHAPTER I

## INTRODUCTION

### 1.1 Background

Classical approaches to robotics require mathematical models in order to plan the robot operations. Because of this requirement, robots that adopt those approaches have to be well engineered and are normally found in a structured environment. In the cases where proper models are difficult to be obtained, the traditional approaches fail to work.

To generate a robot controller in such situation, two approaches can be used. The first approach required human intuition to program the robot. The second approach use learning algorithms to automatically synthesize a robot controller.

Robot programming can be done at several levels. A low level program provides a direct mapping between sensors and motors. It is analogous to programming a computer with a machine code. A higher level program organizes and co-ordinates low level programs in some manners. A successful example of this approach is Subsumption Architecture proposed by Brooks [4].

However, programming a robot is very different from programming a computer. Traditional program development cycle include designing, programming, testing, and debugging. Programming becomes very difficult for a complex robot task. Testing and debugging is tedious because a robot movement is slow compared to the electronic speed. Debugging a robot program is also very difficult. It is difficult to put the robot into the same situation where a bug occurs.

Instead of programming a robot by hand, we can use learning algorithm to generate a robot controller. Evolutionary algorithms such as Genetic Algorithms [17, 14] or Genetic Programming [24] allow robots to learn. Evolutionary approach is inspired by

natural evolution. The process of evolutionary computation involves fitness evaluation, selection, reproduction, random variation of individuals in population.

## 1.2 Evolution of Robot Programs

Evolutionary approach is used as robot learning because of the following reasons.

- For most problems, verifying the answer is easier than finding the answer. From a user point of view, evolutionary approach is advantage to other approach. The user simply defines a fitness function which evaluates the performance of the solution. Evolutionary approach does the job of finding the answer.
- Using evolutionary approach is more convenient. In a robotic task, the same fitness function can be used with different robots to perform the same task. A roboticist does not have to invent a new robot algorithm for different types of robots.
- Learning is also beneficial in the case when a robot is put to work in the unseen environment. A robot with learning capability is able to deal with uncertainty.

This work uses an evolutionary algorithm namely Genetic Programming to synthesize a robot arm control program. The robot program is capable of reaching a target in the environment filled with obstacles. The robot arm is visually guided.

## 1.3 Improving the Robustness

Evolutionary roboticists often evolve a robot program for a particular environment. The result is a program that ties to that particular environment. Therefore, an evolved program may not work even if the environment is slightly changed. The change in environment can occur when a robot program evolved for a simulated environment is transferred

to a robot that works in the real environment. Because of the difference between real world and simulated world, the trajectory of the simulated and the real robot are different. Some successful program in simulation might fail to work in the real world.

In this dissertation, we will show that robustness of a control program evolved in simulation can be improved. To improve the robustness, we trained the program in simulation with multiple robot arm configurations.

The robustness problem aroused from the fact that simulated model is different from reality. However, we used simulation because evolution can be done several order of magnitude faster compared to evolution in the real robot. To avoid the problem of inaccurate model completely, the on-line evolution, i.e. evolution on the real robot, can be used. On-line evolution takes very long time compared to using simulation. Floreano and Mondada [11] evolved an obstacle avoidance behavior on a physical mobile robot in about thirty hours. Chongstitvatana and Polvichai [8] estimates that genetic learning of a robot arm program will take about two thousands hours.

In this dissertation, we will show that learning can be done on-line in a reasonable amount of time. We used a memoized function to allow faster evaluation time. The robustness of a program evolved on-line is improved from the one evolved off-line.

#### 1.4 Dissertation Organization

In the next chapter, Chapter 2 “Control Architectures” describes various approaches used to control the robot arm. Chapter 3 shows several issues in evolving a robot controller. Chapter 4 explains the related work on how the robustness of an evolved controller can be improved. Chapter 5 describes the experiment on evolving a robot arm control program to solve a visually-guided target-reaching task. Chapter 6 discusses two methods to improve the robustness of a control program. The first method is to evolve a control pro-



gram with multiple robot arm configurations. The second method is to evolve the control program on-line. Chapter 7 compares the performance of Q learning with Genetic Programming. Chapter 8 concludes the dissertation.



สถาบันวิทยบริการ  
จุฬาลงกรณ์มหาวิทยาลัย

## CHAPTER II

### CONTROL ARCHITECTURES

Controlling a multi-link arm robot is a difficult computational problem. The robot controller has to have knowledge about the relationship between the robot arm and the environment. The simplest method is that the robot has to know its kinematics. However, having such knowledge is not enough to control a robot in a visual-reaching task. To generate a robot arm controller required a planner to generate the sequence of motions that can lead to a desired configuration.

This chapter presents several approaches to control a robot arm to reach a target in an environment that filled with obstacles. It starts with a geometrical Configuration Space approach. Next, Behavioral Based approach is discussed. Then, the biologically inspired Neural Network is explained. Lastly, Genetic Algorithms and Genetic Programming can also be used to produce a robot arm controller.

#### 2.1 Configuration Space

Robot algorithms differ from computer algorithms in the major ways. Computer algorithms operate in perfect information environment. The effect of each command is known and documented whereas a robot algorithm operates on a model of the robot and the real world. In many cases, the model is difficult to find.

Some robot algorithm has to operate in real-time. Each robot movement takes time to move from one place to another. Each robot movement also differs in the time taken. For example, moving the arm by 90 degrees would take less time than moving the arm by 10 degrees for nine times. Most computer algorithms do not require such a strict timing.

A motion planner is one of robot algorithms. It gives a robot a plan, which is a sequence of motions that a robot can execute to reach a goal. More specifically, we are

interested in finding a plan for a robot arm in a known static workspace.

### 2.1.1 C-Space Transform

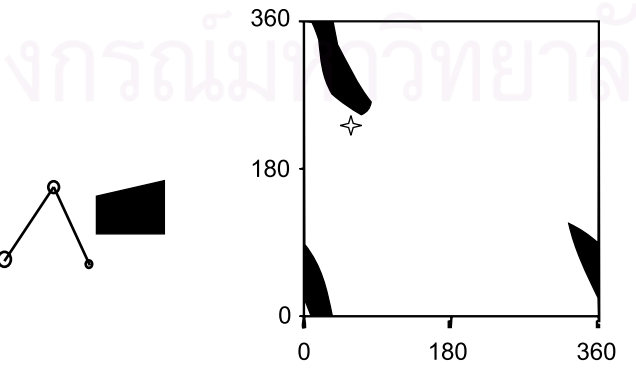
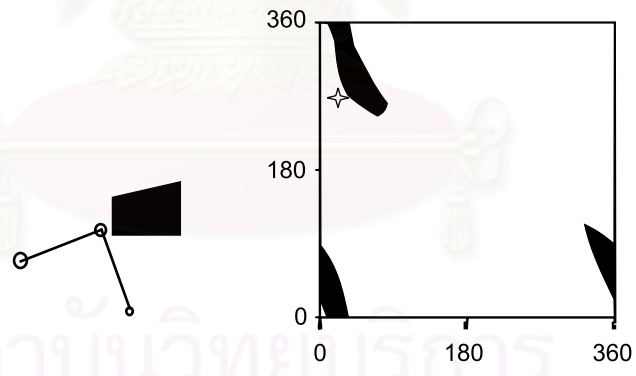
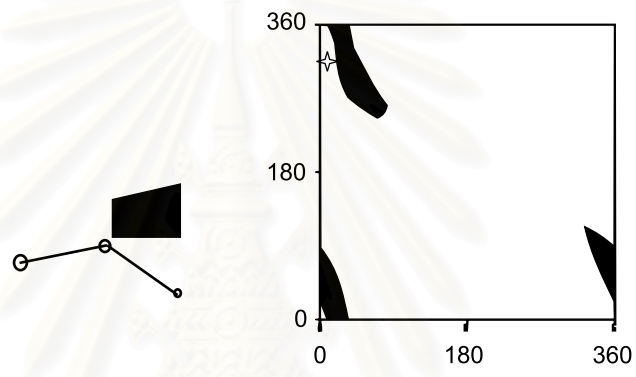
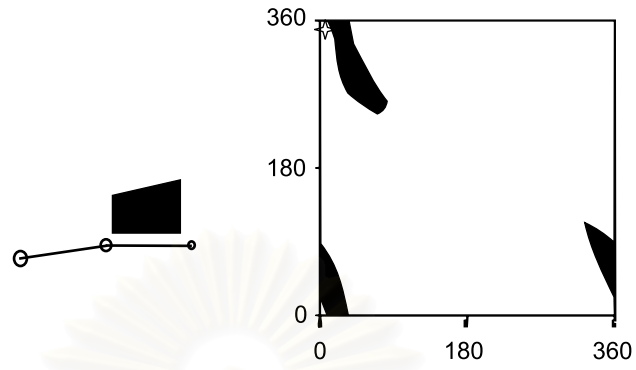
To control a robot arm to reach a target position in an environment filled with obstacles, a roboticist used a technique call a Configuration Space to plan a robot movement. Geometrical techniques are applied to transform a 3D space into a configuration space [28]. A configuration space is the set of all robot configurations. After the transformation, the robot will become a point in a configuration space. Thus make it easier to search for a plan.

The left area of Figure 2.1 shows a two DOF robot arm. The right area shows its Configuration Space. It shows the arm configuration based on the angle of the first joint in the X axis and the angle of the second joint in the Y axis. Any configuration of the robot arm in the left window corresponds to the point in the Configuration Space. A collision free path in a configuration space corresponds to a collision free path in the workspace.

### 2.1.2 Planning in C-Space

Planning in C-Space raises discretization issues. A typical geometric planner might discretize a continuous collision-free C-Space to build a free-space graph. If discretization resolution is too coarse, planning is likely to fail. If the resolution is too fine, planning would take too much time.

A well-known search algorithm can be applied to a free-space graph. Each node in a graph corresponds to a collision-free range. Each link in a graph indicates that the ranges are adjacent in a configuration space. Adjacency in configuration space means that two ranges are reachable by a unit change in exactly one joint parameter.



สถาบันวิทยบริการ  
จุฬาลงกรณ์มหาวิทยาลัย

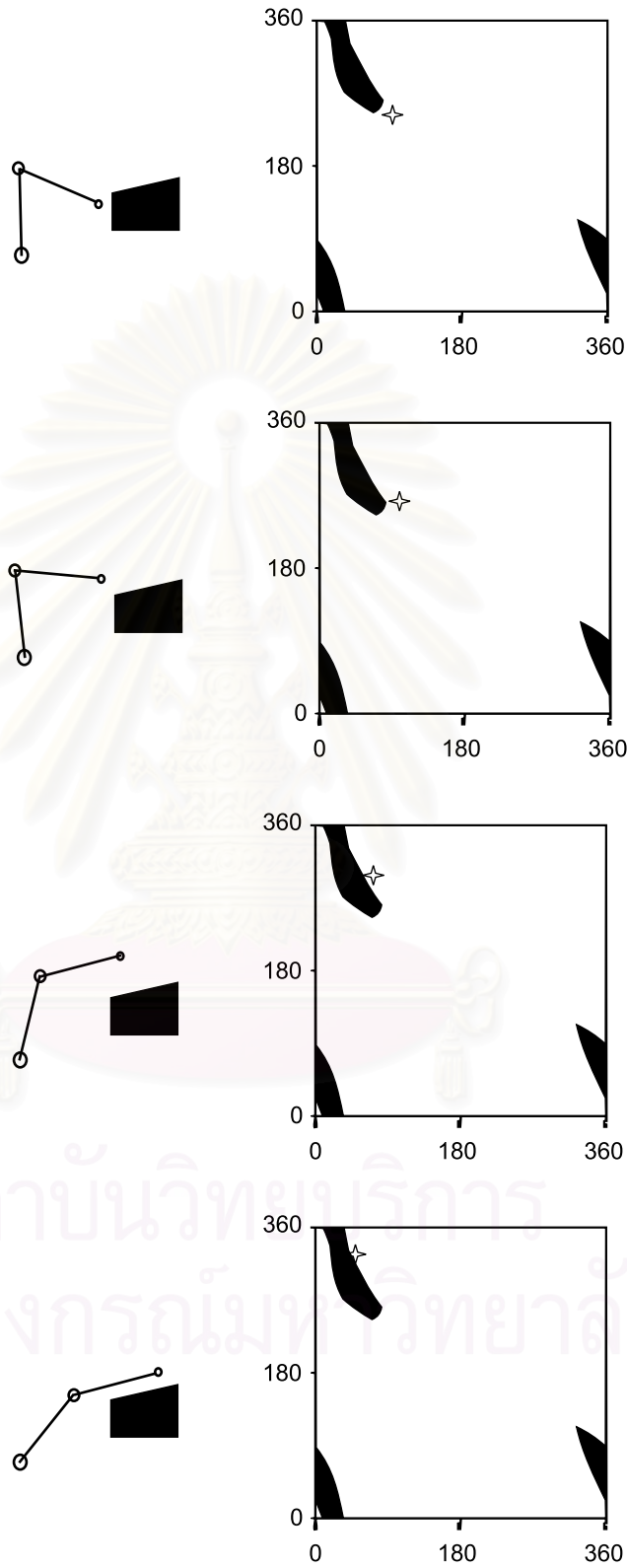


Figure 2.1: The configuration space for 2 DOF robot

### 2.1.3 Advantages

A configuration space planner is a complete algorithm. Meaning that it will find a plan if exists. Moreover, if there is no plan, the planner is able to notify the user that no plan exists.

Compare to other architecture discussed in this chapter, Configuration Space planning is the only complete algorithm.

A configuration space planner is a task planner. Thus, it is able to find a plan for any know polyhedral obstacles.

### 2.1.4 Disadvantages

A configuration space has the number of dimensions equal to the robot's degree of freedom. The higher the dimension, the longer it took to find a plan. A proof provided by Reif [15] is a strong evidence that any complete algorithm required exponential time regard to the robot's DOF. Therefore, if the robot has a high DOF, it is impossible to plan using a complete algorithm. However, a probabilistic planner mostly be able to provide a plan for a high DOF [22].

A probabilistic planner creates a roadmap graph in a configuration space by samples the configurations. The graph is a roadmap of free configuration. The planner plans in the roadmap instead of the whole configuration space. The performance of the algorithm is not guarantee. However, the probabilistic algorithm will be useful if it is proved that the chance that it can find a plan increased quickly as the number of samples increases.

The algorithm required a strict level of certainty. The model of the robot and environment must be modeled very accurately to avoid any damage that causes when a robot hits an obstacle or itself. Calibration is needed to match the model with real world.

## 2.2 Subsumption Architecture

Conventional approach to robotics decomposed the task into functions. For example, the C-Space approach discussed in the previous section decomposes the task into modeling, planning, and acting. Each function is executed by a centralized processing unit. It forwards its output to the next function synchronously.

Subsumption approach is an alternative approach in controlling a robot. This approach is proposed by Brooks [4]. In subsumption approach, the robotic problem is decomposed into task-achieving behaviors. Each task-achieving behavior operates locally and asynchronously. It only communicates by suppression and inhibition.

### 2.2.1 Task-Achieving Behavior

A task-achieving behavior is created by a layer of control. Each layer is composed of a network of finite state machines. Each finite state machine runs asynchronously. It communicates by passing a message to another finite state machine. The finite state machine is driven by the message it receives. The state of the machine can be changed if time has been expired.

Figure 2.2 shows a layer of control that makes a robot avoid obstacles. The leftmost state machine receives input from a sonar sensor. The rightmost state machine outputs to motors. The middle state machines do the job of determining if the robot should stop or should turn away from the obstacles.

### 2.2.2 Layering Behaviors

To simplify the task of building a complex robot, a behavior is built incrementally. After a layer is built and tested, a more complex behavior can be built upon an existing layer. The new layer can add more functionality to a robot while still maintaining a

functionality of the lower layer.

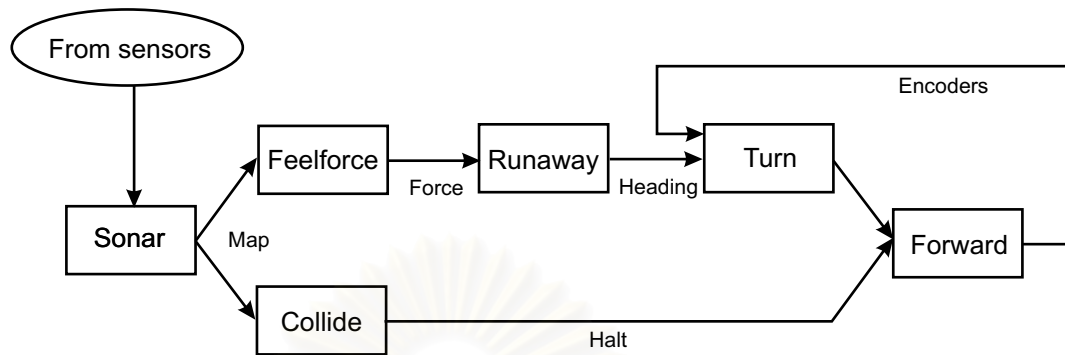


Figure 2.2: Avoid layer

Because each finite state machine is operated asynchronously, a message can be in conflict. Suppression and inhibition is a mechanism that can resolve a conflict. A higher layer can suppress messages to a lower layer module. A message from a higher layer can suppress a message from a lower for a specified time period. A higher layer can also inhibit output from a lower layer module.

### 2.2.3 Advantages

A subsumption robot has the following advantages.

- Subsumption architecture allows complex behavior to be built incrementally. The complexity of building a complex behavior is thus reduced. Each layer can be tested and debugged until it can display desired behavior.
- The robot can work in the real world. This is due to the quick reaction and ability to handle uncertainty.
- Simple processing unit, suitable for a low power mobile robot.
- No complex world model is required. A robot uses the world as its own model.
- This approach uses human intuition to program a robot. In many cases, it is



shown that certain knowledge that human possesses is more tailored to a task that the robot has to work on. Those kinds of knowledge are physical properties of the robot and the environment. The designer used that knowledge to directly solve the problem.

#### 2.2.4 Disadvantages

There are two disadvantages of using this approach. They are scalability and resolving a conflicting behavior.

- The first disadvantage is about scalability. If the robot has a lot of sensors, it is very difficult to monitor or make guess which sensors should be involved in creating a certain robot behavior. Moreover, it is not known that how many layers can be built upon the existing behaviors. The interaction of the behavior will be beyond a designer's imagination.
- The other disadvantage is about resolving a conflict behavior. The connections of each primitive augmented finite state machine are designed by human. As for simple behavior, where the goal of each finite state machine conforms to each other, the robot will work without any problem. However, the problem occurs when the goal of two or more equal priority behaviors are conflicted to each other.

#### 2.3 Neuron Network Controller

In the same way as the architect build the building by mimicking a structure found in nature, robotic researcher can create a robot by drawing analogy from biological creature [16]. Natural evolution creates a robust system that can stand to noise or adapts to change easily. The reason we need this property is because we need the robot to work in the real environment. Most of the animals use nervous system as their control structure.

A neuron network is a network of neurons connected via synapses. A neuron can be regarded as a function with received inputs from the incoming synapses. In general, if the input signal is strong, they can excite the neuron such that the neuron will fire a signal to its neighbors. If the input signal is low, the neuron can inhibit the incoming signal. A network of neurons can be regarded as a network of simple processors. A neuron computed by firing a signal at different level to its neighbor.

### 2.3.1 Network Architecture

An artificial neuron network simulates the biological nervous system. Each neuron receives inputs as the weighted sum of the input signals. The output of each neuron is a function of the weighted sum. We normally used a sigmoid function as a neuron function. The sigmoid function is shown in Figure 2.4.

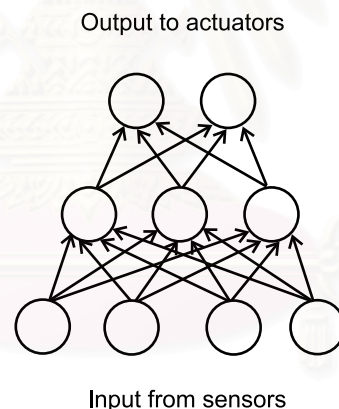


Figure 2.3: A neuron network controller is normally placed between robot sensors and actuators.

As shown in Figure 2.3, the neuron network is placed between robot sensors and actuators. The input signal from the sensor connected to neuron in an input layer. The output from the network drives an actuator. Inside the network, neurons can be connected in various ways. The behavior of a neuron network depends on the structure of the network, the way neurons communicate, and the weight of the synapses.

The simplest form of network architecture is multilayered feed forward neuron net-

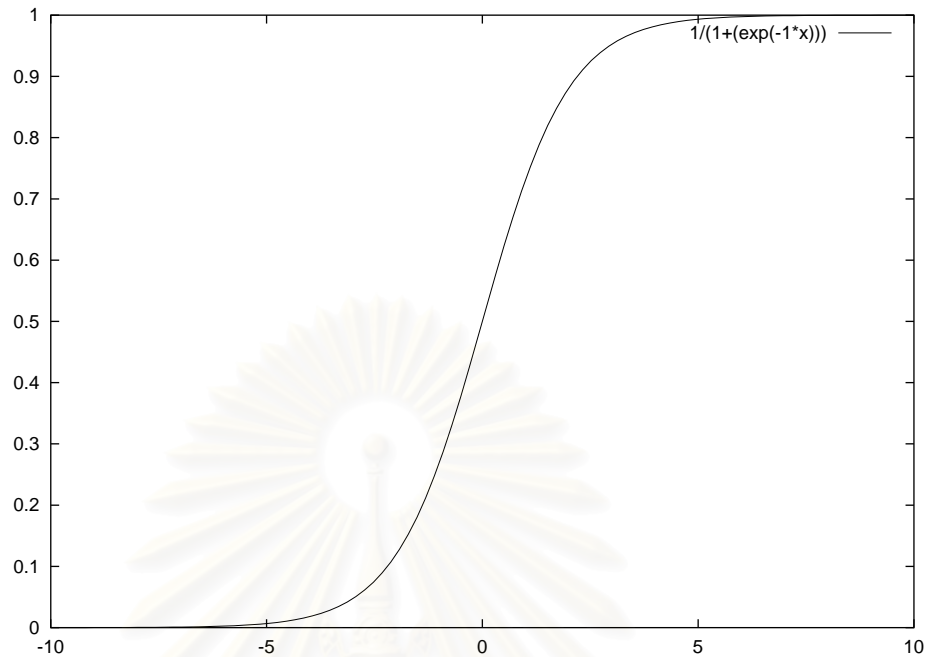


Figure 2.4: Sigmoid Function

work. Normally, there are three layers in the network: input, hidden, and output layers. Outputs from the units in input layers connect to hidden units. Outputs from hidden units connect to the output units. The signal is fed from the input unit to output unit in forward direction only.

### 2.3.2 Training the Network with Genetic Algorithm

A neuron network can be viewed as a computer. As for a simple feed forward neuron network an artificial neuron network can be programmed by changing the synaptic weights. Backpropagation is an algorithm that can be used to train the network. It used the concept of gradient descent. It tries to reduce the local error of each individual synaptic weight. The global error is reduced after several epoch of local error reduction.

However, using Backpropagation algorithm might raise some problems. The algorithm might not be able to escape from a local minima. Moreover, Backpropagation al-

gorithm is a supervised learning. There are many cases that the teachers cannot supervise the learner.

An alternative approach to Backpropagation is to use Genetic algorithm (GA) to evolve neuron network weights. GA operates on fixed length binary string. Before the evolution begins, we have to encode neural network weights into a binary string. Each individual string encodes a set of neuron network weights.

Besides evolving network weights, we can evolve the network structure. As for simplicity, we normally fixed a structure of the network and evolved only neuron network weights. But how can we determine a structure of the neuron network such as how many layer or how many hidden unit. As a rule of thumb, a network of more layers is typically able to perform more complex task. However, several layers mean more nodes and more time to learn.

The following subsection will describe in more detail about how to use GA to evolve a vector of network weight.

### Individual

Neural weights encoded to chromosome which are adjusted during the evolution. In the standard GA, an individual is a fixed length binary string. However, to evolve a vector of network weights, we encode the weights into a vector of real numbers. Those real numbers are attached with synapses. The behavior produced by network depends on the synaptic strength.

### Fitness Function

Every network is evaluated with a fitness function. The fitness of a network depends on how well it can perform the task.

## Selection

GA uses a selection method to select an individual to produce offspring. There are many ways to select an individual. The idea is that a highly fit individual has higher chance to be selected. The following are some of selection method that can be applied to Genetic Programming.

- Fitness Proportionate Selection – Probability that an individual is selected is equal to its fitness divided by the sum of all fitness.
- Tournament Selection – A small group of individual in population is randomly chosen. The fittest individual in the group is selected.
- Elitist Selection – An individual in the elite group (e.g., top ten) in population is randomly selected.
- Rank Selection – Instead of selecting an individual based on fitness value, rank selection selects individual based on its rank in population.

## Genetic Operators

Standard genetic operator operates on a binary string. To modify a vector of real values, we modify operators a little.

- Reproduction – This operator simply duplicates the parent's chromosome to the child's. Even though there is no progress made by applying this operator, there is no deteriorating. The job of this operator is to keep a highly fit individual in the population.
- Mutation – A gene in child's chromosome is randomly changed with mutation probability. The real value genes are slightly increased or decreased. A

mutated individual might have a higher or lower fitness than the parent.

- Crossover – This operator combines two parent's chromosomes together. There are many ways to combine two chromosomes. For example, we can combine two chromosomes uniformly. However, for some problem, combining two network chromosomes might not make much sense.

### 2.3.3 Advantages

Using neuron network as a control architecture is preferable because of the following reasons.

- Neural Network is a structure that can stand to noise. It can produce robust behavior in the real world.
- Using an artificial neural network allows us to test the idea from biology. Moreover, we can borrow ideas from biology to improve the performance of artificial neural network.
- Research in Neural Network is matured. There are many theory and techniques that can be applied to solve a robotic problem.
- The architecture is ideally coupled with the environment and the robot through simple feedback loop. It allows fast interaction for the real-time response.

### 2.3.4 Disadvantages

Designing a network structure requires trial and error.

A neuron network is very difficult to analyze. The pattern of firing neuron signal occurs at a very low level and almost does not make any sense to any human.

## 2.4 Genetic Programming Program

Genetic Programming (GP) can be used to automatically generate a computer program. Because it was inspired by Genetic Algorithm, its algorithm is similar to Genetic Algorithm. Figure 2.5 shows the flowchart of GP. The difference is that the individual in GP population is a LISP program. The difference in the individual caused the genetic operator to operate differently. GP was introduced by Koza [24].

Genetic Programming can be used to evolve a controller that successfully transferred to a real robot. GP can represent a problem in various levels. For example, a sensor and actuator level, or a behavioral level. Lee et al. [27] applied GP to evolve behavior primitives and behavior arbitrators for mobile robots. They used GP to evolve primitive behaviors. After that, GP was used again, in the different level, to evolve a control program that selected the primitive behaviors in order to generate a new and more complex behavior. Chongstitvatana and Polvichai [8] evolved a robot arm control program using GP. They solved the problem of reaching targets in environments that contained obstacles.

### 2.4.1 Individual Program

Individual in GP population is a program tree. The size of a program tree is not fixed. In theory, GP has a freedom to find a solution of any size. This is beneficial to the user of GP because there is no need to estimate the size of the solution in advance.

Figure 2.6 shows how GP might express the expression  $\sqrt{x^3}$  given only SquareRoot, \*, and x primitives. Each tree consists of primitives from a function set and a terminal set. Terminal primitives locate at leaf nodes of the tree. Function primitives locate at inner nodes of the tree.

Figure 2.7 shows an example of a robot program. Here a terminal set consists of MoveForward, TurnLeft, TurnRight. A function set consists of IfClose and IfAway. The example program will check if the robot is close to obstacle. If it is close, the robot will avoid the obstacle by turning left. If not, the program will check if it is heading away from

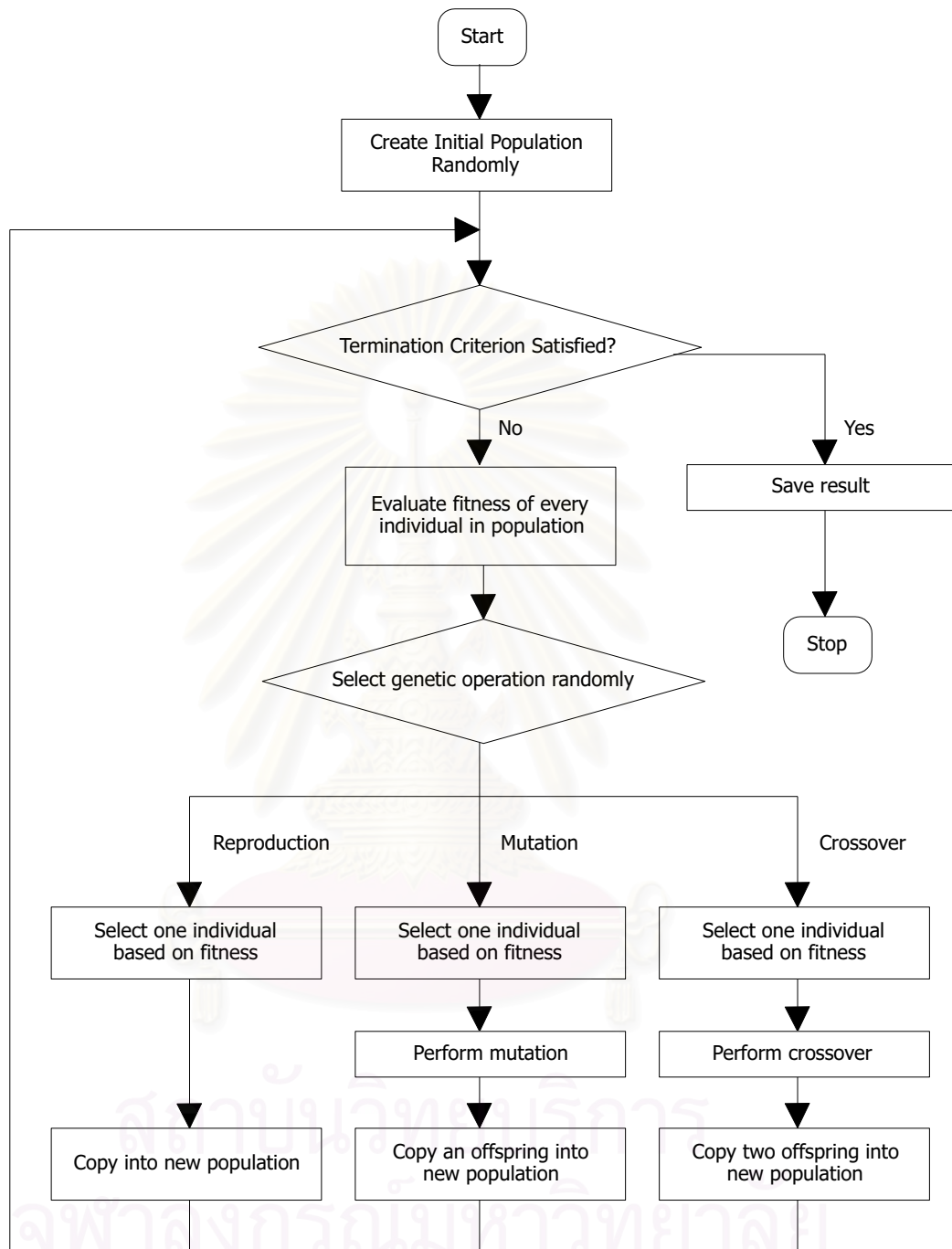


Figure 2.5: Flowchart for the genetic programming.



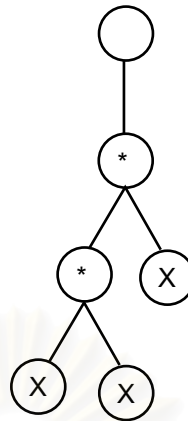


Figure 2.6: Tree for the expression  $\sqrt{x^3}$ .

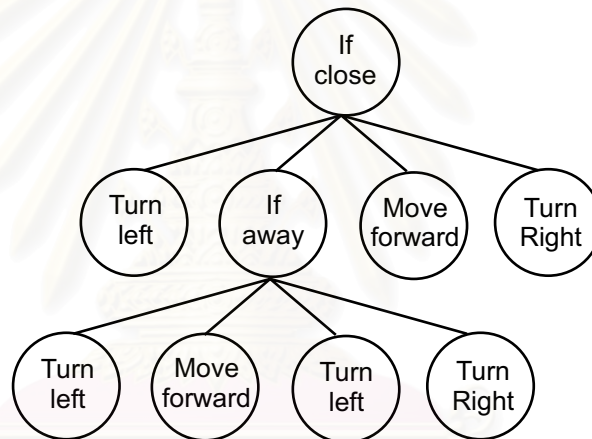


Figure 2.7: An example of mobile robot program.

the destination. If it is moving away from the destination, it will turn right (with a hope that it can compensate with left turning when close to obstacle.) If it is moving toward the destination, it will move forward to that direction. This program will be repeated until the robot reaches its destination.

#### 2.4.2 Fitness Measure

In nature, a highly fit individual has more chance of surviving and reproduces offspring. Natural selection is the survival of the fittest. Fitness is also a driving force of

Genetic Programming. A program tree is run and its performance is evaluated. A fitness function gives rise to a program. Writing a fitness function for GP to generate a robot program is much easier than writing a robot program.

### 2.4.3 Selection Method

GP can use selection method as in GA.

### 2.4.4 Genetic Operators

After being selected from population, an individual (i.e, a program tree) would be applied one of genetic operators. In Koza's original book on GP [24], the primary genetic operators are Darwinian's reproduction and crossover. The reproduction operator simply copies the selected individual to the next generation. The crossover operator exchanges subtrees of two individuals and thus gives two off-springs. In the book, the reproduction probability and the crossover probability were set, as the default values, to 0.1 and 0.9 respectively. The secondary genetic operators are mutation, permutation, editing, encapsulation, and decimation.

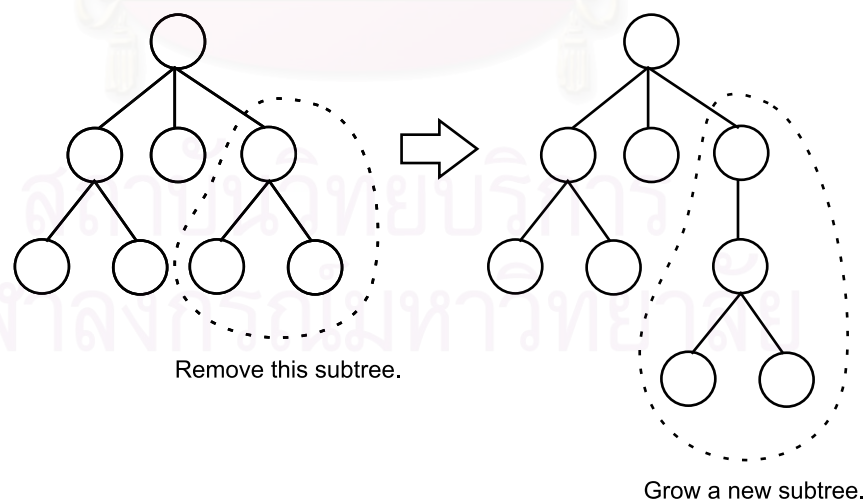


Figure 2.8: Mutating a GP program.

Because the structure of individuals in GP and GA are different, genetic operators

in GP is different from GA. Koza's mutation operator randomly removes a subtree and randomly grows a new subtree. Koza's recombination operator swaps subtrees from two parent trees. Mutation and recombination operations are shown in figure 2.8 and figure 2.9 respectively. Nevertheless, sometime different genetic operators make the search faster.

#### 2.4.5 Advantages

Genetic Programming received input in symbolic form. This makes it easier to help GP in producing an answer compared to neuron network which we almost have no idea how to help evolution finding the answer. GP also produced output in symbolic form. It is beneficial to analyze a behavior of a robot [40].

Genetic Programming can evolve programs at a various level. Lee et. al. evolved behavior primitives and behavior arbitrator using GP [27]. They evolved two behavior primitives using GP. Behavior primitives use low level. After that, GP is used again to evolve a behavior arbitrator which does the job of activating or deactivating behavior primitives in order to perform a more complex task.

#### 2.4.6 Disadvantages

The strength of GP depends on choosing a right terminal or a function set. If primitives in both sets are not sufficient to solve a problem, then GP will never be able to find the answer. In the extreme case, we can use machine code to evolve a control program [40]. However, evolving at a machine level might take times. High level terminals or functions are designed based on human judgement. It may help finding the answer quicker but it might prohibit the autonomy of a robot.

GP is not a method that animal uses as a control structure. Therefore, any result from neuro biology cannot be applied to GP.

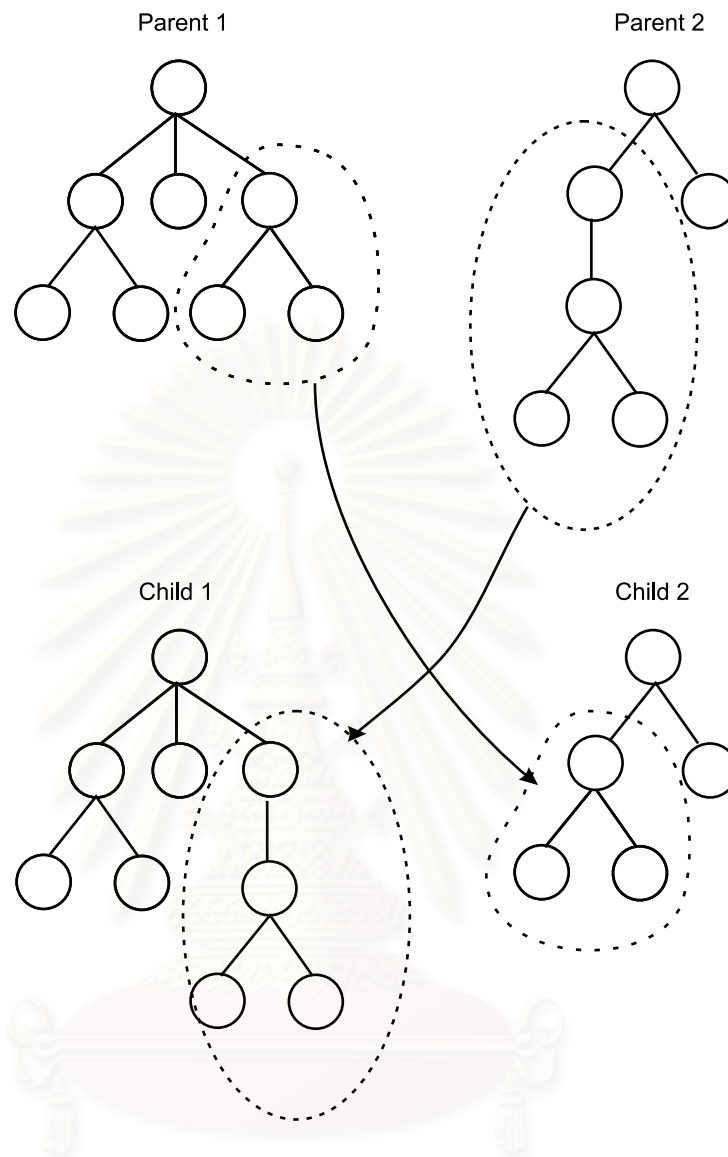


Figure 2.9: Recombination of two GP programs.

## 2.5 Conclusion

This chapter explains four robot control architectures namely Configuration Space, Subsumption Architecture, Neuron Network, and Genetic Programming. Each type of architecture can be used to control a mobile robot as well as a robot arm. Each type has its own strength and weakness. As for the robot task we are concerned, we concentrate on using evolutionary approach to build a controller for the robot task. The next chapter discusses how to apply evolutionary approach to solve robotic problems.

## CHAPTER III

### ISSUES IN EVOLVING A ROBOT CONTROLLER

We focus on applying an evolutionary technique, namely Genetic Programming, to produce a robot arm controller. Even though evolutionary process can automatically produce a robot controller, researchers have to select appropriate evolution parameters. The right choice will help evolutionary technique to find the answer (i.e., a robot controller) in reasonable amount of time. This chapter discusses each choice in details.

This chapter is organized as follows. We first describe issues related to using Genetic Programming in synthesizing a program. However, evolutionary method can also be used to evolve a robot morphology (e.g., sensor position, wheel size). Then, we describe three types of evolution namely off-line, on-line, and hybrid evolution. Finally, we discuss miscellaneous issues regarding on procedure for evolving a robot controller.

#### 3.1 Genetic Programming Issues

We use Genetic Programming as method for synthesizing a robot arm control program. An overview of Genetic Programming was described in the last chapter. In this section, we describe Genetic Programming in more details in the following perspectives.

- Initial structure.
- Fitness measure.
- Selection method.
  
- Operation which modify the structure.
  
- Terminating condition.

### 3.1.1 Initial Structure

The structure of a program is tree-like. Each node of a program tree is an executable unit selected from a function set and a terminal set. Before the evolution starts, the first generation of program trees is randomly created. A randomly created tree can be any size. To limit memory consumption, we normally specify a program size (in nodes) or the depth of a tree.

- **Function and Terminal Set** – GP constructs a program tree by choosing functions and terminals from a function and a terminal set. Both set have closure property. The closure property states that every function in the function set is able to accept any value return from any function in a function set or any terminal in a terminal set. Both set has to be closed because Genetic Programming can arbitrary cut any branch of a tree and connects or grows a new subtree at any random point.

Both set also has to be sufficient. In other words, a function and a terminal should contain items sufficient to solve a program. Otherwise, Genetic Programming cannot find the answer. For example, in the domain of Boolean function, AND and NOT function are sufficient to represent any Boolean function. However, AND and OR function are not sufficient. Therefore, if we want to find a Boolean function using Genetic Programming, we cannot use the function  $f = \{ \text{AND}, \text{OR} \}$ .

Functions and terminals represent the type of a solution. It can be said that GP operates at different levels of representation. Reynolds reported that changes in representation caused changes in difficulty in finding a solution [47]. In his experiment, he compared the effort (i.e., difficulty) in finding controllers for a robot with unconstrained sensors (i.e., roving eyes) and a constrained one. The result showed that the constrained representation (or simple represen-

tation) reduced the difficulty in finding a solution.

- **Code Size and Depth** – Genetic Programming creates a program in the first generation almost randomly. We normally determine the initial code size or code depth to bound the memory consumption of program population. In some cases, we limit the program size to give Genetic Programming a hint of how complex the solution will be.

The code depth as well as the size depend on the complexity of the problem. In the extreme, it is obvious that the program of depth 1 (without any `PROCn` functions) cannot solve a difficult problem. But larger code consumes memory.

Soule and Foster [48] showed that the program size grew to protect the destructive effect from standard crossover. In their experiment, the size, the depth, the active node were observed during the evolution. They also noticed that a program tree tended to become sparse.

- **Population Size** – The larger number of population increases chance in finding a result. The more complex the problem is, the larger the population size should be [24]. However, the amount of time taken to evaluate the entire population would be increased when there are more population. The question is what the appropriate size for solving a problem is. The experiment conducted by Gathercole and Ross [13] shows that small population size (e.g., 50), with an appropriate selective pressure, over many generations performs better than large population size (e.g., 2,000) over few generations. They noticed that, for each generation, small population size will give a small increment in fitness value. In a survey of Parallel GAs [6], Cantú wrote that different sizes of population evolve in the different ways.

### 3.1.2 Fitness Function

Evolutionary searches are guided by fitness functions. Selected individual are able to reproduce, sexually or asexually. Without a fitness measure, evolutionary algorithms are not different from random search.

Designing a fitness function requires insight and trial-and-error. As for a difficult goal, a designer might divide the goal into several subgoals. Thus, it is easier for the evolutionary algorithm to find the result. However, by tailoring fitness function into subgoals eliminates the autonomy of evolution. The resulting controller is biased with the designer.

Urzelai and Floreano [53] proposed a framework which can describe, assess, and compare fitness function. It was defined by three dimensions.

- **Functional-Behavioral dimension.** A functional fitness is measured based on the component that makes the robot functions. A behavioral fitness is based on the components that measure the behavior of a robot. In other words, a functional fitness evaluates the cause of a behavior while a behavioral fitness evaluates the effects of a behavior.
- **External-Internal dimension.** A robot can measure an internal fitness function by itself whereas an external observer is needed to measure an external fitness function. An external fitness function is easier to design. However, it is not always possible to evolve a control program in the real robot using an external fitness measure.
- **Explicit-Implicit dimension.** This dimension refers to any decision and any constraint human explicitly imposed to a fitness function. The more constraint human imposed in the function, the more explicit the fitness function is. In real life, the fitness function is not stated explicitly. Thus, implicit fitness function



is in accordance with autonomous of the robot.

A fitness function has to be carefully designed. For example, if the robot task is to avoid the obstacle, one might set a fitness function as the sum of the distance from the robot to obstacles. The higher the sum results in the higher the fitness. This fitness might create a robot that simply runs away from obstacles and then stops when it is furthest from all obstacles.

### 3.1.3 Fitness Cases

A fitness case represents a situation that a computer program must be dealing with. In some cases, a fitness case represents an initial condition that a program starts with. A fitness is normally measured as a sum or an average of fitness cases.

In most problems, there are an infinite number of instances. A result program should be able to be applied to all instances of the problem. Too few fitness cases result in overfitting: the program cannot deal with more general cases. However, it is impossible to run all individual programs against all fitness cases. Therefore, an appropriate number of fitness cases must be chosen. Teller and Andre [52] proposed a strategy for choosing the number of fitness cases.

### 3.1.4 Operation which Modify the Structure

Evolution is progress when the fit individual is modified. The modification can be done sexually (i.e., crossover) or asexually (i.e., mutation.)

- Crossover. Angeline [1] showed that the standard crossover was not a GP building block engine by an empirical prove. In his paper, he compared the performance of the standard crossover and two headless chicken crossover methods—the strong one and the weak one. Headless chicken crossover selects

one individual from the entire population as a parent and mates the individual with a randomly created tree. He found that the performance of the headless chicken crossover was comparable to the standard crossover. This showed that it did not matter which one would be selected as the crossover partner. However, we cannot yet conclude that the standard crossover is not influential in finding an answer. This is because the numbers of the benchmarks used in this study are few.

Poli and Langdon [42] proposed a GP schema theory to explain how GP with one-point crossover works. One-point crossover is different from the standard crossover that the crossover points of both parents are at the same position in the tree. An experiment is also conducted to study the schemata in the real GP population. The result showed that the one-point crossover was not as disruptive as the standard crossover. Therefore, one-point crossover converges faster than the standard one.

- Mutation. Mutation randomly picks a node in a tree, removes the node and its subtrees, and replaces the deleted subtree with a new randomly created subtree. Koza did not put an emphasis on mutation. He categorized this genetic operator as a secondary genetic operator. In his book, the mutation probability is set to 0.0 and the operator is briefly considered only in two sections. He gave a reason that the occasional usefulness of mutation in GA (i.e., to preserve diversity) was inapplicable in GP because functions and terminals rarely disappeared from the entire population. Another reason was that crossover operated in the similar way as point mutation. However, Luke and Spector [29] reported that mutation performed better than crossover when there were less population and more generations. They also gave a combination of the genetic operator suitable for some problems in their paper. Those combination are as follows: 20-30 % mutation rate, 60-70% crossover rate, and 10% repro-

duction rate. However, with the same reason as the experiment on crossover conducted by Angeline in the previous subsection, we cannot yet conclude that the standard crossover is not influential in solving a problem.

In evolutionary learning system, mutation is like a double-edge sword [46]: The higher mutation rate increases the rate of adaptation and increases the chance of destructive mutation. Also, if the mutation rate is 100%, then the search method is random and the selection method will play an important role in finding the answer.

### 3.1.5 Selection Method

A selection method is considered an important factor in finding a result. An individual is being selected based on its fitness. There are many types of selection methods. For instance: roulette wheel selection, elitism, and tournament selection. In roulette wheel selection or fitness-proportionate selection, the chance of being selected is proportional to an individual's fitness. Elitism selects a group of best individuals in the entire population. Tournament selection selects the best individual of the group of random population. The size of the group (i.e., tournament size) specified the selection pressure: The smaller size, the lower selection pressure. Putting more selection pressure causes the population converges (i.e., become identical) faster. An experiment conducted by Gathercole and Ross [13] shows that the selection pressure should be varied according to the population size.

In some problems, it is hard to measure the fitness of an individual. Such problems include art, music, and literature. A user-driven selection is needed. However, since a human selection takes times, modeling a human strategy by extracting important information after he evaluates an individual helps speed up the evolution process [41].

### 3.1.6 Terminating Condition

Evolution can be seen as never ending process. However, in practice, genetic programming has to be terminated. The termination criterion for evolving a robot program is when a robot can display the desired behavior. In the case where we cannot find the solution after evolution is run for a long time or no progress is made, we should terminate the evolution.

## 3.2 What To Evolve

Natural evolution produces both brain and body for animals. Analogously, artificial evolution can be used to generate a robot controller as well as a robot body.

### 3.2.1 Evolving Robot Controller

Programming a robot by hand to produce a nontrivial behavior is very difficult. Evolutionary Algorithms can be used to produce a robot controller. Majority of work in the field of Evolutionary Robotics is about producing a robot controller. Because of the generality of evolutionary algorithms, evolutionary algorithms can be applied to produce virtually any type of robot controller. The two popular one are neural network controller and Genetic Programming.

Various parameters such as network weights, neuron types, the type of communication, or a network structure are involved in successful behavior of Neuron Network controller. We can encode those parameters as genotype and use evolutionary algorithm to evolve it. However, in general, Evolutionary Robotist often used Genetic Algorithm to encode the network weights. All other parameters are predetermined by human.

Besides a neuron network controller, Genetic Programming is successful in producing a robot control program. The program has a tree-like program structure. Evolving a

control program using GP has different issue from evolving a set of network weights. Since our work focus on using Genetic Programming to evolve a robot arm controller, issues regarding the use of Genetic Programming will be discussed in more detail in this chapter.

Robot controller can be classified into two types: reactive and non-reactive controllers. Both neuron network controller and GP program can be both types. As for neuron network controller, the type of the controller depends on how we connect the network. For a GP program, the type of the controller depends on the primitives in function set and terminal sets.

- **Reactive.** A reactive controller is a controller which a robot action depends on the current sensor value. In other words, sensor stimulates the robot's action. A feed forward neuron network provides this type of controller. As for genetic programming, the type of a controller depends on a function and a terminal set. A reactive controller can produce behaviors such as obstacle avoidance [11], exploration and homing, and a visually guided robot.
- **Non-reactive.** A non-reactive controller motor output is not a function of the current sensor value. In fact, this type of controller does not require any sensor input to produce a behavior. A non-reactive behavior can 'remember' an event that occurred in the past. The behavior might depend on the current sensor value and the past event. The behavior might also depend on the current sensor value and robot's internal representation. The representation can be anything such as a flag in a program, geometrical model, augmented finite state machine, a recurrent neuron network, etc.

The set of reactive controller is a subset of the set of non-reactive controllers. Normally, a non-reactive controller can perform more complex task because it can avoid the ambiguous situation where the same sensor input

needs different action. However, a reactive controller can get out of the ambiguous situation using an active perception strategy. An active perception strategy is the process of using motor actions to select sensory patterns such that they are easy to be discriminated [38].

### 3.2.2 Evolving Robot Morphology

The performance of the robot depends not only on the robot controller but also on the robot body plan. The body plan includes robot body size, wheel radius, sensor placement, etc. By encoding these properties into the genotype, Evolutionary Algorithms can be used to produce a robot physical properties together with the robot controller. In some cases, increasing the size of search space by adding physical properties can reduce the search time.

Lund, Hallam, and Lee [30, 26] argued that true Evolvable Hardware (EHW) should evolve all the hardware system since the electronic part is dependent on the other parts that form the system. They evolved a robot body with its controller (i.e., a brain as they called) at the same time. The body part genome is a string of floating point numbers. The brain part is a program tree. The evolution mechanisms used are GA and GP respectively. They analyzed the effect of different body plan in some problems.

Cliff et. al. [9] evolved a visually guided mobile robot. The genotype of the robot encodes both the physical positioning of the photo sensor on the robot body.

### 3.3 Types of Evolution

Artificial evolution that occurred in simulation is called off-line evolution. The one that take place in the real world is called on-line evolution. This section discusses the advantages and disadvantages of each type. The hybrid off-line on-line combines the strength of both types.

### 3.3.1 Off-line Evolution

Evolutionary Algorithms involved thousands of fitness evaluation. To evolve a control program for robot, single evaluation might take 1-10 minutes depends on robot's mechanical speed. Evolving a control program in a computer gives a quick result. Moreover, using simulated robots is cheaper than building a physical robot. A simulated robot can be evaluated continuously without having to maintain or repair.

Simulation is useful for studying some fundamental problems in evolutionary robotics. To study the possibility of an intelligent robot in solving a problem, evolving a control program in a simulator is a affordable choice. Reynolds [47] used GP to evolve a control program for an obstacle avoidance robot. He co-evolved a control program and obstacle topology. Koza et al. [25] showed that the time-optimal control strategy can be found by using GP. Fitness function brings about such a strategy. Calderoni and Marcenac [5] also used GP to evolve a collective behavior.

The best result obtained from off-line evolution can be tested on a physical robot. However, the result is likely not robust due to the difference between the simulated world and the real world. The difference is called a simulation-reality gap. Crossing the gap is not trivial because of two major reasons [20]. First, it is difficult to model any aspect of the real world accurately. Second, it is very difficult to include every aspects of the reality in simulation.

Chapter 5 will discuss some methods that can be used to improve the robustness of a robot controller evolved from simulation.

### 3.3.2 On-line Evolution

Evolving a robot controller in a simulator has an advantage of reducing time. However, the difficulty is how to make a realistic simulator so that the robots in simulation

and in the real world behave in the same way. In order to avoid problems in modeling a real world in simulation, evolving a controller in a real robot is an alternative way. Using a real robot in evolving a control program takes a great deal of time more than evolving in a simulator.

Although experimental result in simulation is useful, Brooks pointed that when using simulation, we might solve a problem that does not exist in the real world or the solution of the problem cannot be used in the real world [3]. This is because a simulated robot is usually oversimplified. Many aspects of the real world such as noise, uncertainty, mass, friction, and inertial forces are ignored. Brooks recommended discarding simulation model and using a real robot as its model.

On-line evolution occurred in on a physical robot in a physical environment. Even though it is an artificial evolution, it is similar to natural evolution in the sense that it occurs in the real world. The tremendous amount of time in on-line evolution leads to several problems [34]. This might be a reason that there are a few attempts to evolve a robot controller on-line. Floreano and Mondada evolved an obstacle avoidance behavior for a mobile robot [11]. The behavior of neuron network controller converges in about thirty hours. Dittrich and his colleagues evolved a controller for a robot with arbitrary structures [10].

There are other problems aside of the problem of waiting time. Due to the time overhead, the battery of the robot will be drained out. A robot part will be broken before the evolution is completed. Moreover, because the robot has to evolve on-line in the physical environment, a robot might be crashed with the working environment. Therefore, we have to provide some safety to prevent our robot from being damaged.



### 3.3.3 Hybrid On-line Off-line Evolution

Hybrid evolution offers a complementary strength of on-line and off-line evolution. It can give the faster evolution time and resulted in robust behavior of the robot. Hybrid evolution starts in simulation until a behavior of the robot converges. After that, the population is transferred from simulation to the real robot. Evolution continues in the real world until the physical robot can express the desired behavior.

Lund and Miglino [32] evolved an obstacle avoidance behavior for a Khepera robot using hybrid evolution. The first 200 generations are evolved in simulation. The last 20 generations are evolved in the real environment. They obtained 98% of reduction in evolution time. The fitness function reach the optimal strategy at the generation 60 in simulation but the fitness can be improved when evolve in the real robot. The behavior of the result after 20 generations in the real world match with the best result from 200th generation in simulation.

## 3.4 Miscellenous Issues

### 3.4.1 Exeperimental Repeatability

Artificial Evolution is a stochastic process. Each run starts from randomly selected population and ends when terminating criterion is met. The solution is not necessary the same for every run. Find a solution that works for one time does not mean that the evolutionary setting can always give the solution. To ensure statistically correct, evolutionary process has to be repeated for several runs.

When the result is tested in simulation without noise, the result would be the same no matter how many times we repeat the test. However, when the resulting controller is tested with a real robot, the result is not the same for every run. Several works in evolutionary robotics selected best individual to be tested and analyzed. This is clearly

not sufficient. The large number of testing makes experiment more creditable but it would take very long time to conduct.

### 3.4.2 Judging Successful Behavior

Evolutionary roboticists express their robot objective using a fitness function. However, satisfying a fitness function does not mean that the objective is satisfied. Many of the time, roboticists have to judge the success of the robot behavior by themselves. Due to the empirical nature of the work in this field, the behavior of a robot is normally tested with one or few testing environment. This is because the overhead of physical experiment. The result is lacking in data.

### 3.4.3 Incremental Evolution

Some behaviors are very complex hence are very difficult to evolve from random population. Instead of evolving a complex controller from scratch, it is better to evolve the behavior incrementally. Incremental evolution starts by evolving a population that can perform a less difficult task. After that, the population is used to produce a more difficult task. Incremental evolution allows evolutionary algorithms to search in a refined search space.

## CHAPTER IV

### RELATED WORKS ON IMPROVING ROBUSTNESS

An evolved controller is often brittle when dealing with an environment that it has never seen during the evolution. The brittleness of the robot program can be observed by looking at a mobile robot's trajectory. For example, Nolfi et al. [37] described an experiment conducted on a mobile Lego robot. A neural network controller was evolved in a simulation. When transfer to the physical world, the trajectories of the real robot differed significantly from that of the simulated robot.

The brittleness of a robot program can be seen obviously by failing to complete its task in an unseen environment. Chongstitvatana and Polvichai [44] used GP to evolve a robot arm program in a simulator. Some successful program, when transferred to the real world, failed to reach the target.

Obviously, robustness is one of the most important issues the evolutionary robotics researchers have to deal with. Several researchers proposed a method of improving the robustness of a robot controller. This chapter reviewed related work on improving the robustness of simulated robots and real robots.

#### 4.1 Improving Robustness in Simulated Robots

Simulation is a useful tool to study some aspects of robotics. It provides faster experiment time. It is much cheaper to construct a simulation than building a real robot. In this section, we reviewed the study of the robustness on simulated robot. A controller obtained from evolutionary algorithms cannot work well when the environment is changed.

This section reviewed experiment on improving the robustness of simulated robots.

#### 4.1.1 Mobile Robot

Perturbation in the simulated environment could improve the robustness of mobile robot control programs [7, 39]. In their experiment, the task of the mobile robot is to move to the target in environment filled with obstacles. The robustness can be improved by injecting perturbation during evolution of a GP program. Simulated environment is perturbed by moving numbers of obstacles. The result showed that introducing perturbation during evolution can improve robustness. The analysis showed that a program can reuse the experience acquired during evolution.

Prateetongkum and Chongstitvatana [45] added a probabilistic function to improve robustness in a navigation problem. The task of the simulated mobile robot is the same as [39]. They used GP to evolve a robot program. The function set included a probabilistic function. The result showed that adding the probabilistic function can improve the robustness of an evolved program. The analysis showed that a program tree with a probabilistic function has more path variety.

#### 4.1.2 Robot Arm

Suwannik [49] evolved a robot arm control program using GP. When the program is tested with different target position, even in a small range, the program fails to reach a target. The robustness of the control program is improved by evolving a program with multiple runs. In each run, a target is randomly placed within a small range. The result shows that multiple runs can improve the robustness of a simulated robot arm control program.

### 4.2 Improving Robustness of Real Robots

Even though a simulated robot is useful in studying several aspects of robotics, a simulated robot is very different from a real robot. Many properties that are occurred in

the real world are not included in simulation. Experiment on a simulation might not be valid unless it is tested with the real robot. However, the more serious problem is that a controller obtained from simulation cannot work well in the real world.

There are four major techniques to improve the robustness of the program evolved from simulation. The first is to build a realistic controller by sampling data from the real world. The second technique adds noise to simulation to improve the robustness. The third technique trains a program with multiple environments. The fourth technique uses special type of control architecture which can resist to change. These techniques can be combined.

#### 4.2.1 Capturing Data from the Real World

Realistic simulation can be built by capturing sensory data and motor effect from the real world. The sampling data is used in simulation.

Lund and Miglino [32] recorded sensor and motor responses from the real robot from different part in an environment. The data is used in simulation where they evolved a controller. After 200 generations of evolution in simulation, they transferred a control program to the real mobile robot. They are able to evolve an obstacle avoidance behavior in the real environment.

Lund and Hallam [31] evolved an exploration and homing behavior. A robot has to explore as much area as possible and keep coming to its home as to charge its battery. The researchers recorded the sensor data by moving a mobile robot to various distance from the light (i.e., virtual charging station) and rotating the robot to different orientation. The real world trajectory is very different from the simulated one. However, the task is accomplished.

#### 4.2.2 Adding Noise

The real world is full of noise. A sensor gives different values from the same position. The same motor command does not drive the robot to exact same position. Simulation can be more realistic when noise is added to simulation. The amount of noise should be at the same level as occurred in the real world.

Jakobi [21] wrote that, in order to cross a reality gap, an appropriate level of noise had to be inserted into a simulation. His experiment shows that if the simulation has no noise as if the robot is in the ideal world, the trajectory path of the robot in the simulator is completely different from the one in the real world. While those trajectories are corresponded to each other in a model with an appropriate level of noise (i.e., a Gaussian distribution with standard deviation equals to the one derived from the experiment). However, if there is too much noise, the robot fails to work as intended.

Miglino et al. [35] built a simulation by recording a sensor data and motor effect from a real robot. They also tested the performance of an evolved controller obtained from three simulations. These simulations have different form of noise in sensor readings. They found that a 'conservative noise' gave the best result. The conservative form of noise makes the robot misinterpret the distance with respect to a randomly selected axis.

#### 4.2.3 Training with Multiple Trials

To evolve a controller that can resist changing, we can 'train' a robot controller with multiple trials. Each trial can be different environment, different robot, or both.

Lee et al. [27] evolved a box-pushing robot. The behavior is complex so that they had to decompose it into subbehaviors and evolved a program for each subbehavior independently. They evolved a behavior arbitrator to activate a suitable behavior. They achieved robustness by training controllers in multiple trials. The robot is randomly placed

in different location and orientation.

Jakobi showed that a complex simulator is not necessary when evolving a controller for a robot with a complex behavior [19]. He evolved a controller for an Octopod (eight-legged robot) in a type of simulation called minimal simulation. The idea of the minimal simulation is that, when evolving a controller for a complex behavior, anything that does not lead to a successful behavior does not have to be taken into account in the simulation. For example, he did not care whether the legs will clash or the belly is dragged along the ground since those behaviors do not lead to a successful behavior. The evolved controller is successfully transferred to the real Octopod.

#### 4.2.4 Adaptive Control Architecture

One of the advantages of using evolutionary approach is that it can learn virtually anything. We can use evolutionary approach to learn any type of robot controller, no matter how strange a controller is.

Floreano [12] evolved an adaptive controller for a mobile robot. Instead of evolving connection strength for a neural network controller, they evolve adaptive rule based on Hebbian learning. The synaptic weights of the robot are randomly initialized. Their values are changed over time by adaptive rules encode in the genotype. This type of control architecture allows fast adaptation time. It also requires less generation and smaller population to evolve. The task of the robot is light switching. A robot switches a light on by moving to one side of the wall. When the light is on, the robot moves to the opposite side of the wall, where the light is placed, and stays there for the rest of its life.

Ishiguro et. al. [18] evolved a controller for a box pushing task. The controller is constructed based on the idea of neuromodulator. A neuron can diffuse a specific type of neuromodulator. After a synapse received a diffused neuromodulator, it will change

its weight according to its own genetically-determined interpretation. The resulting controller evolved from simulation can perform the task robustly on the real robot.

### 4.3 Conclusion

This chapter reviews related work on improving robustness of a evolved controller. Simulated robots need some technique to improve robustness when dealing with changing environment. Real robots need those techniques in order to work successfully in the real world (i.e., change from simulated environment to the real environment.)



สถาบันวิทยบริการ  
จุฬาลงกรณ์มหาวิทยาลัย



## CHAPTER V

### EVOLVING ROBOT ARM PROGRAMS FOR A TARGET REACHING TASK

The goal of the experiment is to evolve a robot arm control program to solve a visually-guided target reaching task. This chapter explains details of the experiment. First, we describe the robot platform. Then, we explain the robot task. Finally, we discuss how to apply Genetic Programming to synthesize a robot program to solve the robot task.

#### 5.1 The Robot Arm

Figure 5.1 shows the overview of the experimental platform. The robot arm system consists of a PC executing a robot program evolved by Genetic Programming technique. The robot arm has three joints made of position-controlled servos. It can move only in a plane. A CCD camera located above the arm provides a robot vision. The vision system monitors the distance between the robot's end effector and the target and checks whether the robot hit any obstacles.

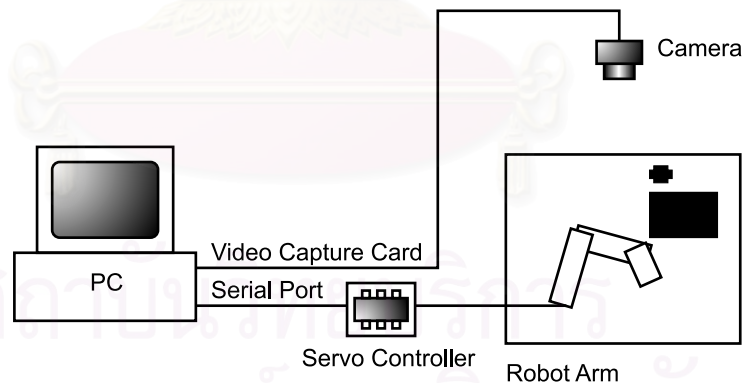


Figure 5.1: An overview of the experimental platform.

##### 5.1.1 Joints and Links

Figure 5.2 shows the robot arm seen from its own vision system. Each joint of the arm is made of a servo used in a hobby radio controlled airplane or car. Every joint uses different model of servo. They are varied in size and capability. The largest servo is

located at the shoulder joint. The smallest one is located at the wrist joint. The smaller servo locates at the end to reduce the weight in order to prevent oscillation because of a servo feedback.

The links of the robot arm are made of wood. The length of the link from shoulder to elbow, from elbow to wrist, and from wrist to the end effector are 25:31:15 respectively. The overall length is scaled down from Polvichai's master thesis [43]. However, the ratio of the length is similar to the robot arm in his thesis.

The joints of the robot arm are driven by different servos. A servo is controlled by a mini servo controller board. The board is connected to a PC via a serial port. The format of the command to the servo board is `servo number , position , sync`. The board can control 4 servos. The positional resolution is from 0 to 0xFF. We did not use such a high resolution. Instead we divided each joint into 60 steps. The synchronization byte is always 0xFF. The command is sent to the port using a call to a dynamic link library. The library hides complicate detail about the serial port API.

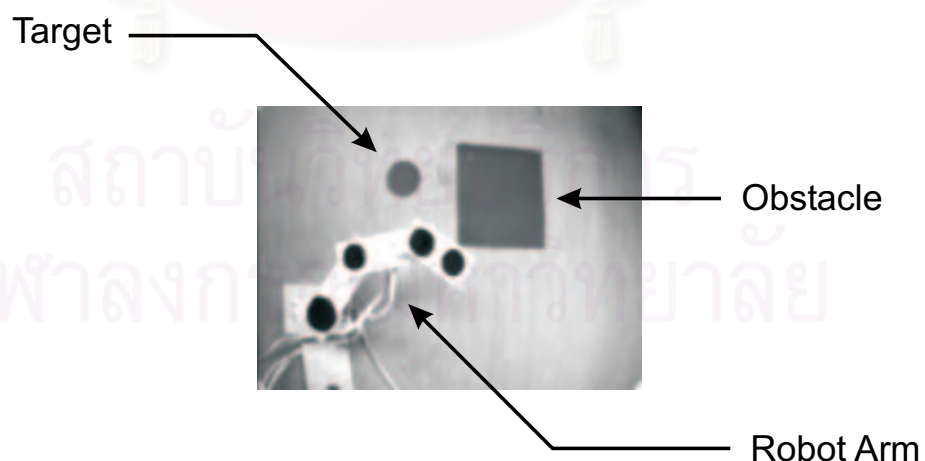


Figure 5.2: A robot arm seen from its vision

### 5.1.2 Vision

A CMOS camera provides a robot vision. The video signal is sent to a PCI video card. We used the Video for Windows API to capture the video image. The captured image is thresholded before each joint is detected. The joints are marked with circular black velvet cloth. The position of each joint and the robot's end effector are the centroids of the circular marks.

The initial configuration of the robot arm is to stretch to the right. Therefore, the robot knows that the shoulder joint is located at the leftmost and the end effect is located at the rightmost. The elbow and the wrist are in between. While the robot is moving, the robot can see only four blobs in the visual field. It can know which blob belong to which joint by tracking each single robot movement.

After the vision system detects three joints and one end effector, it assumes that there is a link between two joints. The links connected the joints are five pixel width.

For each image, four blobs are detected. However, some robot configuration can be out of the visual field. We prevent this to happen by creating virtual obstacles around the border of the visual field. The robot arm cannot move out of the visual field because it cannot move pass the virtual wall.

The obstacles can be detected by several ways. Since the image detection algorithm can measure the size of the detected blob, we can use an obstacle which size is bigger than the joint mark. The big blob is the obstacle. An alternative is to can paint the obstacle a bit lighter or darker than the joint marks. If we threshold the captured image at different level, the vision routine is able to tell which blob is the obstacle or joint marks.

The control loop is as follows. First, the robot starts from stretching to the right. Second, an image detection and tracking routine provides the position of every joint and

the end effector. Then, a program tree is executed. If the arm movement command is executed, the program waits until the movement is completed. If the end effector is on the target or the maximum number of movement is reached, the program stops. Otherwise, the process is repeated starting from the second step.

## 5.2 Robot Task

The task of the robot is to reach a target in an environment filled with obstacles in a reasonable amount of time. A program does not have to avoid the obstacle completely. A robot arm is allowed to touch obstacles.

Due to the stochastic nature of Genetic Programming, we cannot guarantee that it will find an answer for every instance of the problem. We conducted experiment on five instances of the problem as shown in Figure 5.3. In each sub figure, a white circle is the target. A black rectangle is an obstacle.

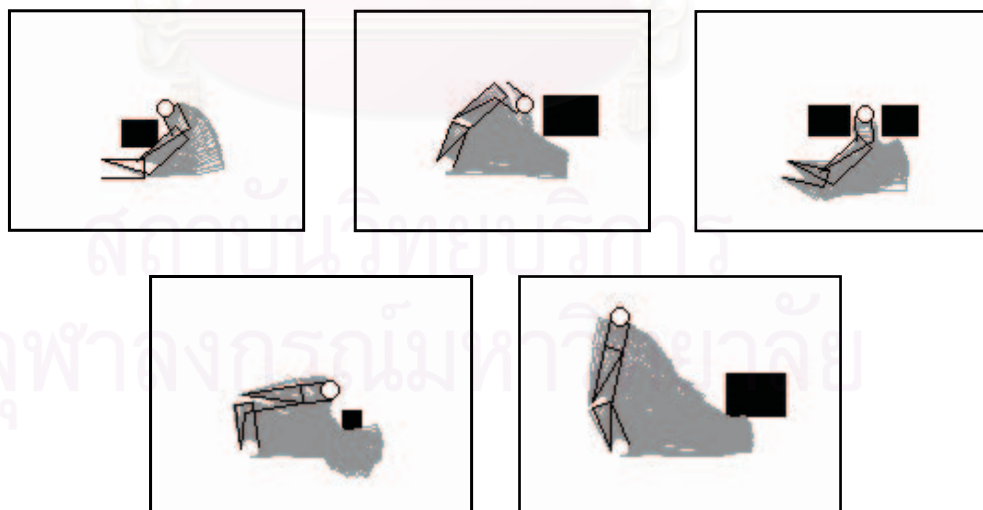


Figure 5.3: Five environments for testing the visual-reaching task.

## 5.3 Evolving a Control Program

### 5.3.1 Control Program

A program starts running from the root node. Because the inner node controls the flow of the program using conditional, few terminal nodes are executed for each program run. We have to iterate the program several times until a specific number of robot movement is reached or a robot reaches the target.

GP used primitives in a terminal and a function set to construct a robot program. Table 5.1 and 5.2 show the primitives in the terminal set and the function set respectively. The terminals set contains robot motion commands and sensing primitives. A motion command moves a specific joint one step. A sensing primitive senses if any of the robot's links hits an obstacle or whether the end effector is closer to the target. The function set contains basic control flow primitives, which are `If`, `IfAnd`, `IfOr`, and `Not`.

### 5.3.2 Function Set

The diagram of `If` is shown in Figure 5.4. This function evaluates the first child. If the result of the evaluation is true, it will evaluate the second child and return the value from the second child. Otherwise, it will evaluate the third child and return its value.

The diagram of `IfAnd` is shown in Figure 5.5. This function evaluates its third child if the logical AND result of the first two children is true. Otherwise, it evaluates the fourth child. `IfOr` operates in the same manner except that its logical operator is OR.

The `NOT` function evaluates its only child. It returns the negation of the result returned from its third child.

Table 5.1: Terminal Set

Terminal Name	Operation
Closer	Return true if the last movement resulted in less distance from the end effector to the target. Otherwise, return false.
Sp, Ep, Wp	Rotate a shoulder, elbow, or wrist one step clockwise. Return true if the operation is successful. Otherwise, return false.
Sn,En,Wn	Rotate a shoulder, elbow, or wrist one step counterclockwise. Return true if the operation is successful. Otherwise, return false.
Farther	Return true if the last movement resulted in more distance from the end effector to the target. Otherwise, return false.
HitP	Return true if the last move is clockwise and the movement hit an obstacle. Otherwise, return false.
HitN	Return true if the last move is counterclockwise and the movement hit an obstacle. Otherwise, return false.
OnTarget	Return true if the end effector is on the target (i.e., the distance between them is less than or equal to 2 units). Otherwise, return false.
See	Return true if there is no obstacles between the end effector and the target. Otherwise, return false.

### 5.3.3 Terminal Set

Items in the terminal set are listed in Table 5.1. The terminal set contains arm movement and sensing primitives.

Table 5.2: Function Set

Function name	Operation
If, IfAnd, IfOr	Basic control flow functions.
Not	Evaluate its child and return the negation.

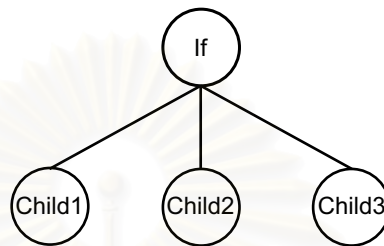


Figure 5.4: An If function

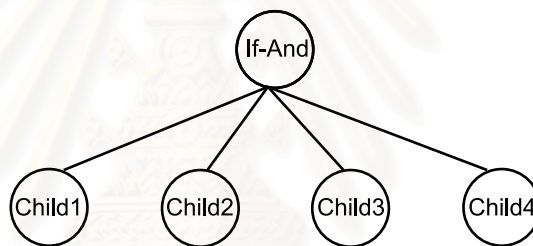


Figure 5.5: An IfAnd function

SP, SN, EP, EN, WP and WN are arm movement primitives. The prefix S, E, and W stand for shoulder, elbow, and wrist respectively. The suffix P and N stand for positive and negative direction respectively. P moves the joint clockwise and N moves the joint counterclockwise by one step. A single movement is in a small step to prevent any damage caused by hitting obstacles. The movement primitives return false if the move hits with any obstacle or reaches its limit. For a successful move, it will return true.

There are two types of arm sensing primitives. The output of the first type depends on the current input. The first type of primitives includes OnTarget and See. We can determine if the end effector is on the target by looking from the captured image. We can also determine that there is no obstacle blocking a line from the end effector to the target.

The output of the second type depends on the current input and the previous movement. `HitP`, `HitN`, `Closer` and `Farther` are of this type.

#### 5.3.4 Fitness Function

Genetic Programming can be considered as a search method guided by a fitness function. A fitness function evaluates the performance of a robot. In evolutionary robotics, a fitness function gives rise to the behavior of the robot. In the experiment, each robot program was given a limited amount of time to be executed in the simulation before its fitness was evaluated. Since the task is to reach a target, the fitness of a robot program can be simply defined as the ability to get close to the target at the end of the run. The fitness function  $f_{single}$  defined as follow.

$$f_{single} = \begin{cases} 1000 & \text{if } d < 2 \\ 1000 - d & \text{otherwise} \end{cases}$$

where  $d$  is the distance (in pixels) between point the end effector and the target at the end of the run.

Our fitness function is simplified from Polvichai [43]. The intention is to eliminate any effect of tuning fitness function that might be resulted in improving the robustness of the control program in the real world. According to Urzelai and Floreano [53], this definition of the fitness function put very much of the effort on the genetic search. By measuring just a distance at the end of each run, the search might stuck at a local minima. However, the problem does not exist in our case. Even though our test cases consist of an instance of target reaching that the robot has to move its end effector away from the target before it can reach the target, GP was able to find the answer to the problem.



### 5.3.5 Genetic Parameters

The genetic parameters listed in Table 5.3 are fixed for all runs. The evolution will stop if the solution is found or it reaches the maximum generations. In the latter case, the evolution will be rerun. These genetic parameters are just good enough for GP to learn the task in reasonable amount of time. We did not try to optimize the genetic parameters.

Table 5.3: Genetic parameters

Name	Value
Population	1,000 programs
The method used for generating the first generation	Grow method with depth limit 4
Crossover rate	80%
Reproduction rate	10%
Mutation rate	10%
Selection method	Tournament selection with tournament size 7
Maximum generation	30 generations

## 5.4 Result

The program did not work robustly when transferred to the real robot. Table 5.4 shows the robustness of the control program in various maps. The robot can reach the target for 79% on average.

## 5.5 Conclusion

This chapter explains the evolution of a robot arm control program. We will see in the next chapter that a robot program evolved from simulation using this setting is not robust. The next chapter presents two techniques that can improve the robustness of an evolved control program.

Table 5.4: Robustness of control programs in various maps

Instance	Robustness of a control program evolved with one configuration
A	86
B	92
C	81
D	84
E	54
Average	79

สถาบันวิทยบริการ  
จุฬาลงกรณ์มหาวิทยาลัย

## CHAPTER VI

### IMPROVING ROBUSTNESS OF ROBOT ARM PROGRAMS

Evolutionary method quickly finds a shortcut in the interaction between the robot and the environment and producing a controller. The resulting solutions are therefore simple and efficient. However, the evolved solution that ties tightly with a specific environment failed to work when there is a change in the environment. This is not the weakness of this approach when compared to other approaches. The change in an environment is also a problem for other approaches as well (e.g., traditional approaches). This is due to the fact that we cannot or did not include the changes at the design time.

There are two types of environmental changes. First, the environment is changed. For example, any object in the environment is moved. Or, the environment is changing from simulation to the real world. Many evolutionary roboticists often evolved a robot controller in simulation as to speed up the evolution time. The solution from simulation is then transferred to the real robot. An evolved control program that is working in the simulator is likely to fail in the real world. The main reason is because it is very difficult, if not impossible, to model the real world accurately. Simulated world is different from the real world.

Even when the environment is static, the robot does not sense and act in the environment the same way every time. This is due to noise in the real world and uncertain effect of motor commands. From these reasons, the robot controller would 'feel' that the environment is changed as well.

In this chapter, we describe a method to improve the robustness in the aspect of both kinds of changes. This chapter firstly describes our definition of robustness are discussed. Next, two methods to improve a robustness of program in the real world are discussed. The first method, a control program is evolved in simulation. To improve the robustness in

the real environment, the population of control programs are trained with multiple robot arm configurations [50]. The second method eliminated the robot model completely by learning the task on-line. Memoized function is used to speed up the on-line learning time [51].

## 6.1 Robustness

### 6.1.1 Reality Gap

The trajectory of the program in simulation will be the same every time we run the program. This is because everything is deterministic. The trajectory of the same robot program running in the real world is different from the one in simulation.

The experiment concentrates on improving the robustness of a robot arm control program. Our first definition of robustness is “the ability to deal with changes in environment”. The change occurred when a controller that evolved in simulation is transferred to the real robot. The simulated environment and the working environment is greatly different. This is intentional; we did not modeled the robot arm as realistic as possible.

### 6.1.2 Real World Reliability

The experimental platform is uncalibrated. The links of the robot arm are made of wood. The joints of the robot arm are made of different servos which have only estimated position control. The camera is uncalibrated. The position and the length of the robot are obtained through a robot's CCD camera. The camera lens has a lot of distortion. The vision subroutine outputs the different result even the robot does not move. The light caused noise in vision system.

With the above reasons, the trajectory of the robot is not the same for every run. Every program is running in the same environment. However, they produced different

results. Some program even failed to reach a target. The second definition of robustness is therefore means that “the robot can perform the repetitive task with a high number of successes.”

## 6.2 Improving the Robustness by Multiple Configurations

The visually-guided target reaching task is described in the previous chapter. Genetic Programming is used to evolve a solution program. By the time consuming nature of evolution, thousands of fitness evaluations are performed prior to the emergence of the desired behavior. With slow mechanical movement of the robot, artificial evolution of a control program on the real robot would take days or months to complete in the real world. Using a computer to simulate the movement of the robot can greatly speed up the evolution time.

When using GP to evolve a control program for a target-reaching task, we found that a control program evolved from a simulation with single robot configuration did not work well on the real robot. Many aspects of the real robot and the real world have not been simulated with enough accuracy. For example, the lens distortion in the vision system had not been included in our simulation. The result is that events that occur in the real world do not have any corresponding events in the simulation.

A robust robot program can perform its task in the real world despite the partial knowledge of the real world in the simulator used to evolve the program. In the experiment, a robot learnt to perform the task in a simulator. The simulator is deliberately built such that it is only an approximation of the real world in three aspects.

- The lens distortion is not included in simulation. The CCD camera used in the experiment has high distortion due to the close up effect. We did not model any lens distortion in our simulation. Therefore, the simulated vision system

sees the arm and the environment as undistorted.

- The motion of each joint is not calibrated with the real robot. We estimated the degree of each step of each link. First, we align a link to the X axis in the vision system. Then we move the link and count the number of steps until the link aligns to the Y axis in the vision system. The degree of each step we used in simulation is equal to 90 divided by the number of steps.
- The noise in locating the position of each joint due to the vision system is not included in simulation. The vision system locates each joint by recognizing the circular marker and calculates its centroid. The light, which caused uneven reflection in the scene, together with the lens distortion resulted in up to 2 pixels noise in reporting the position of each joint.

### 6.2.1 Multiple Configurations

We believe that by adding those aspects to the simulation model the robustness of the real robot arm will be improved. However, rather than improving the accuracy of the simulation, we proposed an alternative. The alternative is to evolve a robot program that does not depend on aspects simulated inaccurately. The important aspect that makes the robot fail to work is that the length of the arm is distorted due to the camera lens distortion. Therefore, we propose to eliminate this dependency by evolving a robot program with multiple configurations.

We conjecture that a program that can control every robot arm configurations from any point to reach a target in a simulation will be working very robustly in the real world. The intuition is that at each state of the robot arm in real world will have a corresponding state in the simulation. And thus, if a program is restarted for every state of the real robot arm, a robot will finally reach a target position.

It is very difficult, if not impossible, to find such a program. However, it gave us some intuition that a program that can control more configurations to reach a target in simulation is likely to be working more robustly in the real world. There are some issues in training the robot arm with multiple configurations. The first issue is how many configurations are needed to train the robot. The second issues is what robot configurations will be included in simulation.

#### How Many Configuration?

It is clear that more configurations resulted in more simulation time. If a robot program is evolved with  $n$  configurations, it would take about  $n$  times more than evaluating a single configuration. Moreover, it is more difficult to find a solution that can work with multiple configurations. The result is that it would take more than linear time increment to find a solution for multiple configurations. In some configuration, it is impossible to reach a target because the arm cannot avoid obstacles or there is no configuration that can move the tip to the target.

In the experiment, we decided to use three robot configurations. This is to make the problem not too difficult for GP to find the answer for every robot arm configuration.

#### What Configurations to be Included?

With the set up that we used, the major discrepancy between the simulator and the real world is the distortion of the size of the arm caused by the lens. The length of each link varies as the arm moved in the visual field but the simulator had no knowledge about this variation. We decided to choose this aspect for the evolution process to evolve a robot control program that was independent of it. A program is evolved to control three different simulated robot arms to perform the task. Those arms are varied in length. The first arm is the same as that used in the first experiment. The second and the third arm are little

shorter and little longer than the first arm respectively.

The first configuration is the configuration that is seen by the vision system from the starting position. The second configuration is the configuration that the length of the arm is seen longest from the vision system. The third is the shortest one seen from the vision system. The length of each link is shown in Table 6.1.

Table 6.1: Length of each link as seen from the vision system

1st link	2nd link	3rd Link
23	29	14
25	31	15
27	33	16

### 6.2.2 Genetic Learning

In this section, artificial evolution took place in simulation and the resulting program is transferred to the real robot. Because GP's probabilistic nature, GP will produce different result program for every run. A number of runs are needed to ensure statistically reliable results. Therefore, we repeated the run 10 times for each instance of the task. The results are averaged over 10 programs generated from GP.

The following subsections describe how we applied GP to the robot learning problem.

#### Terminal and function set

The function and terminal set are the same as in the previous chapter. The terminal set contains sensing and arm movement primitives. The function set contains basic control flow primitives. The terminal set and the function set are listed in Table 5.1 and Table 5.2 respectively.



## Fitness function

The fitness function  $f_{single}$  that can synthesize a program that is able to control a single robot arm to perform the task is defined as follows.

$$f_{single} = \begin{cases} 1000 & \text{if } d < 2 \\ 1000 - d & \text{otherwise} \end{cases}$$

where  $d$  is the distance (in pixels) between the robot's end effector and the target.

For the program evolved with multiple configurations, the fitness function is the summation of the fitness function  $f$  evaluated from each configuration. The program is accepted as a solution when all three arms can use it to reach the target. The fitness function  $f_{multiple}$  is defined as follows.

$$f_{multiple} = \sum_{config=1}^3 f_{single_{config}}$$

## Genetic parameters

The genetic parameters listed in Table 6.2 are fixed for all runs. The evolution will stop if the solution is found or it reaches the maximum generations. In the latter case, the evolution will be rerun. These genetic parameters are just good enough for GP to learn the task in reasonable amount of time. We did not try to optimize the genetic parameters.

### 6.2.3 Experiment

The task of the real robot arm is to reach the target while avoiding the obstacles. The robot control program was evolved in simulator. After the evolution is completed, the control program is transferred and tested with the real robot.

Table 6.2: Genetic parameters

Name	Value
Population	1,000 programs
The method used for generating the first generation	Grow method with depth limit 4
Crossover rate	80%
Reproduction rate	10%
Mutation rate	10%
Selection method	Tournament selection with tournament size 7
Maximum generation	30 generations

As shown in Figure 5.3, five instances of the problem were created as the representatives of the problem. They varied in target positions and obstacle positions. A circle in each picture is a target. Black rectangles are obstacles. To make the experiment easier to conduct, the obstacles and the target are not captured from the robot vision. This simplification does not have any effect on the measurement because both types of control programs will be tested with the same benchmark.

The robustness of a robot program evolved with single configuration and one that evolved with multiple configurations are compared. The partial knowledge of the real world is compensated by making a control program that did not rely on a robot configuration.

After the control program is obtained from simulation, the robustness of robot programs was measured in the real world. The robot is allowed to move not more than a specific number of steps. The robustness is measured in the real world as the percentage of time the real robot can successfully reach the target. The arm can reach the target if the distance between the tip of the arm and the center of the target is less than or equal to 5 units. They are measured against five instances of the target-reaching problems shown in

Figure 5.3. Those instances are the same as that the robot had learnt. The different is that a real robot arm is used to run the control program in this measuring phase.

Each program ran on the real robot arm 10 times. A run that has errors will be discarded. The first type of error is the vision system. An error was occurred when the robot vision cannot detect all the joints. When a robot hits the target, it generally bounces back to the previous collision free configuration. However, sometime the vision system still detected that the robot still hits the obstacle. This type of run was also discarded.

#### 6.2.4 Result

The genetic learning time is limited to 30 generations. For a program that can solve one simulated arm, it normally took less than that to evolve a successful controller. However, in some cases, 30 generations are not enough to find the control program. For a controller that could control three simulated arms to reach a target, it took more effort to find a solution program. Effort also depends on the difficulty of an instance of the problem. Some instances are more difficult to evolve a successful program.

The real world is noisy. The robot arm did not move exactly the same trajectories even it was using the same program. There are errors from several sources. First, the model of the robot is not accurate. The length of arm seen from the vision system is varied due to the lens distortion. The joint step of the robot arm is just the estimation of the real joint step. Second, the robot vision detected different joint position due to variation in lighting and shadow.

Table 6.3 shows the robustness measured from each map. The robustness value varied from map to map. The average robustness of the proposed method is 90 %. Training with multiple configurations can improve the robustness by 10 %. In the first two instances, a program trained with this technique almost never misses a target.

Table 6.3: Robustness of control programs in various maps

Instance	Robustness of a control program evolved with one configuration	Robustness of a control program evolved with three configurations
A	86	100
B	92	99
C	81	90
D	84	85
E	54	75
Average	79	90

### 6.2.5 Conclusion

GP can create a robot control program in simulation. A control program evolved from simulation is not robust in the real world. The reason is because simulation is not accurate. However, evolving a control program that does not rely on any inaccuracy in simulation can create a robust control program. We identified the main reason that causes the program to fail. To improve the robustness, we evolved a control program that does not rely on the robot configuration.

In this section, a robot arm control program is evolved in simulation. The advantage of using simulation is that the evolution can be done very fast. However, the simulation is required and human intuition about what aspect of the real world needed to be independent of is required. In the next section, we will show that evolving entirely on the real robot can avoid both requirement and still improve the robustness of the robot control program when tested on the real world.

### 6.3 On-line Evolution

Natural evolution is a very slow process. Progress has been made over thousands of generations, which take billions of years. However, with the speed of today's computer,

thousands of generations in off-line artificial evolution might take only few hours. Off-line evolution of robot controllers can be done very fast because it occurs in simulation. However, the controller obtained from simulation is not likely to work robustly when transferred to the real robot. This is due to the inaccurate model of the physical world.

To overcome this shortcoming, the evolution can be performed with a real robot in the physical world. Artificial evolution that occurred in the real world is called on-line evolution. On-line evolution eliminated the requirement for a model completely. The robot controller is evolved on the real robot which can be regarded as its best model.

Evolving a robot program in the real world in real time is similar to the evolution process found in nature. Thus, on-line evolution is very time consuming. Chongstitvatana and Polvichai [8] estimates that genetic learning of a robot arm program will take about two thousands hours.

Artificial evolution can be sped up by using the following techniques.

- Tailoring evolutionary algorithms to suit the problem we are interested in. According to "No Free Lunch" theorem, there is no best evolution algorithm for every problem.
- Changing the genetics parameters or operators. For example, increasing the population size can increase the chance of finding a solution but it might slow down genetic search as well.
- Reducing the time used in evaluating an individual. For example, Aporntewan [2] proposed that using hardware as evaluator instead of using software to simulate hardware can speed up hardware evolution by 36 times.

Our approach falls into the last category. The bottleneck is at the robot performance

evaluation. If the robot is allowed to run at the maximum step of 300 steps and each step took 1 second. It would take 5 minutes for evaluating one program. If there are 1,000 programs in one generation, it would take about 83 hours to complete one generation. From this reason, it is natural to reduce the evaluation time.

The evaluation time can be reduced by letting the robot learn the effect of its action and reuse the learned result while evolving a controller for the task. We used memoized function to learn the motor effect of the robot movement.

### 6.3.1 Memoized Function

To evaluate an individual on-line, each robot motion is performed in the physical world. However, many of these motions are repetitive hence their effects are already known. If these effects are stored and reused, then a large number of actual motions can be eliminated. We propose a memoized function to store the effects of robot motions.

Memoized function or memo function is a function that remembers which arguments it has been called with and the result returned. If the function is called with the same arguments again, it will return the result from its memory rather than recalculating it. The same principle is found at the hardware level in computer architecture. A cache is used to store recently accessed memory locations.

The memoized function receives joint angles as the arguments. It outputs the positions of all joints. The implementation of the memoized function can hold every possible combination in the joint space. Since each joint of the robot arm has 60 discrete steps, there are a  $60^3$  or 216,000 entries. However, for larger size of configurations, we do not have to allocate all entries to implement the function. The technique of virtual memory can be used. The memoized function can be implemented with a much smaller physical memory.

### 6.3.2 Genetic Learning

The robot arm control program is evolved on the real robot. A program is evolved on-line using the same terminal set and the function as in the previous experiment. The terminal set and the function set are shown in Table 5.1 and Table 5.2 respectively.

The fitness function for on-line evolution is shown below. It is the same as the fitness function for producing a controller for a single robot arm configuration.

$$f_{single} = \begin{cases} 1000 & \text{if } d < 2 \\ 1000 - d & \text{otherwise} \end{cases}$$

where  $d$  is the distance (in pixels) between the end effector and the target.

Genetic parameters are listed in Table 6.4. Most parameters are the same as the previous experiment except the population size and the terminating condition.

We choose the different population size because of the following reason. Before we evolved the robot program on the physical robot, an experiment in simulation is conducted to test the performance of the approach. It is found that the population size affected the on-line evolution time. Among tested population size, the size of 200 individuals gives the fastest evolution time.

The terminating condition for the previous section is when the maximum generation reaches 30 generations. In this experiment, the evolution is continued until no progress has been made during 10 consecutive generations. This is because it is expensive for the evolution to waste the time without making any progress. If the evolution stops without a solution, it will be rerun with the memoized function that its memory has already been filled. We included time for a run that cannot find the solution program.

Table 6.4: Genetic parameters

Name	Value
Population	200 programs
The method used for generating the first generation	Grow method with depth limit 4
Crossover rate	80%
Reproduction rate	10%
Mutation rate	10%
Selection method	Tournament selection with tournament size 7
Maximum generation	No limit. Evolution is terminated when there is no progress for 10 consecutive generation.

### 6.3.3 Experiment

This experiment has two goals. The first goal is to evolve a robot program that can perform a visual reaching task on-line. The on-line evolution must be sped up such that it will give a solution within an acceptable amount of time. The second goal is to compare the performance of a robot program evolved on-line with the one evolved off-line.

The simulation gives the estimation of the time that the robot learned the task with and without using the memoized function. Different sets of genetic parameter were used in simulation. The best parameter set was used in on-line learning. The estimated time and the exact time spent in on-line evolution using the memoized function are compared.

Five instances of a target-reaching task similar to the one in the previous section were used. They are shown in Figure 5.3. The reason that we did not use the results from the previous section is because the old camera was broken. We have to replace it with the new one. Because the new camera is not the same model as the old one, it sees the robot and the environment differently.



We compare the time evolved on-line using memoized function with the estimated time evolved on-line without using any speed up technique. After evolution generated the solution program, the program was tested on the same environment as it was evolved. The robustness of the program evolved on-line and the program evolved off-line are compared.

#### 6.3.4 Result

As shown in Table 6.5, a physical robot can learn the task on-line in less than one hour on average when using the memoized function. The performance of the approach is compared with naive on-line evolution. Since it took a considerable amount of time for a robot to learn the task on-line without using the memoized function, learning time is estimated by using simulation. The estimation assumed that each movement of a simulated robot took one second to complete similar to the physical robot. When compare the estimated time with the actual on-line evolution time, the average speed up is about 23 times.

Table 6.5: Evolution time

Problem	With Memoize (hours)	Without Memoize (hours)	Speed up (times)	Recall (%)	Memory used (%)
A	0.51	5.00	9.64	89.52	0.90
B	0.64	6.33	7.80	85.65	1.13
C	1.19	25.32	20.93	94.82	2.20
D	1.28	132.68	59.09	93.45	2.31
E	0.93	34.81	17.18	81.91	1.79
Average	0.91	40.83	22.93	89.07	1.67

The recall rate is very high. Each recall means the required data is found in memory, which means that the robot does not have to move in the physical world. Less than 3 percent of the memory allocated for the memoized function was used. It is likely that this approach can scale to higher DOF robot. Normally, genetic search does not search the entirely new space everytime. The first generation filled more entries compared to later

generations. This is because the offspring is likely to be in the same configuration as its parents is. The new configuration implies the exploration of the offspring.

As show in Figure 6.1, the percentage of recall and speed up are exponentially related. Starting from zero recall, the speed up is equal to one (i.e., no speed up.) The speed up grows much larger when the recall rate is above 95 percent. A controller for a more complex task might be able to evolve on-line in a reasonable amount of time if the recall rate is very high. Again, simulation can be used to test the feasibility of this approach.

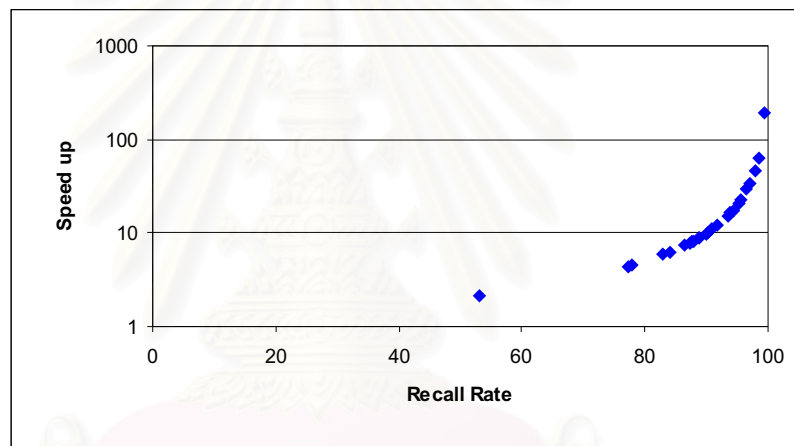


Figure 6.1: Recall Rate vs Speed up of a memoize function.

The robustness of a robot program is the percentage of times the robot can successfully perform the task. The robustness of a program evolved off-line (i.e., in simulation) is compared with a program evolved on-line with the memoized function (i.e., in the real world). The result is shown in Table 6.6. The average robustness of a program evolved on-line is 27% higher than the one evolved off-line. The robustness depends on the similarity between the environment that is used to evolve robot programs and the environment that the robot programs actually work. The on-line evolution occurred in the actual environment that the robot programs work as opposed to the off-line evolution which used the simulation. Therefore, a program evolved on-line evolution tends to have higher robust-

ness.

Look at Table 6.3 and Table 6.6. Notice that the robustness of the program evolved off-line with single configuration in the previous section is higher than the one evolved off-line in this section even when the test cases are similar. The reason is that we test the control program more rigorously than the previous section. In the previous section, the program can reach the target if the tip is in the ranges 5 pixels away from the target. In this section, the tip has to be within 2 pixels away from the target.

It is found that the failed program moved to the configuration that is not found in memory many times more than the successful program does. In other words, the failed program moved to the configuration it was not learned during the evolution. We hypothesize that this is due to noise in the system that leads the robot to move to unknown configurations. One way to improve the robustness is to add the same level of noise to the memoized function when evolving a control program.

Table 6.6: Robustness of an evolved program

Problem	Programs evolved off-line	Programs evolved on-line with the memoized function
A	82	98
B	26	76
C	82	82
D	56	60
E	28	92
Average	55	82

### 6.3.5 Conclusion

In this section, learning a robot task was divided into two parts. A robot learned the visual reaching task using Genetic Programming. A robot learned its interaction with the environment using a memoized function. On-line evolution allows robot to learn without

requiring any robot model. The memoized function helps speed up the on-line evolution. By using this approach, learning a visual-reaching task can be speed up by about 23 times.

The evolved robot program works robustly in the testing environment. To achieve higher level of robustness, the same amount of noise as observed in the real world should be added to a memoized function. Moreover, the memoized function can be replaced by another learning algorithm that tolerates noise.

The size of required memory for implementing memoized function grows exponentially with the degree of freedom of the robot. However, the memoized function can be implemented with the technique of virtual memory as it is found that the percent of usage is low in the experiment.

From this experiment, we propose the framework of synthesizing a robot control program on-line. Synthesizing a robot program on the real robot without using a model can be divided into two parts. The first part is to learn the sensorimotor effect of the robot. The second part is to learn the task.

If a realistic simulation model of the robot and the environment are available, the model can replace the learner of the first part. However, if the goal is learn the task on-line, the learner must learn the model.

The second part is to learn how to perform the task. It can be achieved using various techniques. Here Genetic Programming is used to learn the robot task.

## CHAPTER VII

### COMPARING WITH REINFORCEMENT LEARNING

An agent that adopts reinforcement learning approach learns by interacting with a dynamic environment [23]. An agent interacts with an environment by trial-and-error. It learns by receiving a feedback from the environment. A good action is reinforced with a reward while a bad action is suppressed by a punishment. After a sequence of actions, the agent can learn a policy that can achieve its goal.

The agent perceives the environment it is situated in. The state of the agent  $s$  is discretized into a set of states  $S$ . At each state, there are some possible actions the agent can perform. After performing an action  $a$ , the state that the agent perceived is changed. An agent learns from the interaction with the environment. It performs an action and received a feedback. A feedback is given to the agent as a reward and punishment. A reward is given to a desirable action.

It is not always a good strategy to choose an action that maximizes the immediate reward. Often, the eventual reward is delayed. Thus, the agent faces the credit assignment problem. It has to determine what action in the sequences of actions deserved to be given the credit when the eventual reward is received.

The result of reinforcement learning is a policy. A policy is a decision making function that maps a state to an action. An agent follows a policy to achieve its goal. An optimal policy can maximize the total reward received from any starting state.

Reinforcement learning can be applied to wide variety of problems. It can be applied to robotics problems, playing board games, planning, and function approximation [36].

A nice property of reinforcement learning is that it does not require input/output pairs. Most learning algorithm requires input/output pairs as training examples. Lacking

the desirable output make the learning more difficult. The learner has only current state and reward as learning information.

Reinforcement learning is suitable for an on-line learning for two reasons. First, it does not require a model. Second, learning is incremental. The agent can learn while its performance is evaluated.

In many cases, the state space of the agent is too large to be fitted in computer memory. In this case, a state generalization method can be applied to reduce the size of the state space. A state generalization technique can improve the efficiency of the learner. A similar set of state are expected to have the same reward when an agent performing an action.

### 7.1 Q-learning

The result of learning is a policy. However, learning a policy is difficult because the training data is not presented in the form of state and action pair. Instead, the feedback received from learning is the reward gains after performing an action. Q-learning can learn a policy indirectly by learning the evaluation function.

Q learning algorithm stores all possible state-action pairs (s,a) in a Q table. Each entry in the table is called a Q value. A Q value is the highest discounted cumulative rewarded of agent performs an action a when it is at state s. Choosing an action with the highest Q value is similar to choosing the optimal policy.

The basic Q learning algorithm is listed below.

1. Initialize all Q values to 0.
2. Do Forever:
  - a. Let s be the current world state.
  - b. Choose an action a with the maximum  $Q(s, a)$ . (choose random action sometimes.)

c. Let  $t$  be the new world state after performing the action  $a$ .

d. Let  $r$  be the immediate reward.

e. Update the stored utility values using the update rule:

$$Q(t, a) = Q(t, a) + \beta(r + \gamma e(t) - Q(t, a))$$

where  $e(t)$  is the maximum  $Q(t, b)$  over all actions  $b$ .

Q-learning can be applied to robotic applications. Mahadevan used Q-learning to teach a behavior-based robot to push boxes [33]. Lin applied Q-learning to train a mobile robot to dock a battery charger. Salganicoff trained a robot arm to pick up an object.

## 7.2 Applying Q-Learning to Visual-Reaching Task

The robot task in this thesis is to reach a target while avoiding obstacle. The model of the robot, such as the length of each link, is not required in order to learn the task. The camera located directly above the robot can observe the interaction of the robot with its environment.

The following ingredients are needed to be set up before applying the Q-learning algorithm to any learning problems.

- Determining a state space  $S$  and set of actions  $A$ .
- Determining a reward function.
- Determining a discount factor.
- Specifying an exploration/exploitation strategy.

In the cases of visual reaching task, some more decision needed to be made.

- How many steps before it should explore the new search space?

- When the learning is completed?
- How many steps should the Q-value propagate back?

### 7.2.1 A State Space and a Set of Actions

The robot arm has three joints. The state of a robot arm is described by three numbers. Each number is the index of each joint. For each joint, there are 60 discrete positions. That makes the total of 216,000 possible states. Each state has six possible actions because its joint is allowed to move clockwise and counterclockwise in a small step. Therefore, there are 1,296,000 entries in the Q table. This size of Q table can fit in a computer memory easily.

### 7.2.2 Reward and Punishment

In the step 2.d of Algorithm 1, the reward and punishment (i.e., negative reward) was given to the robot. It will be punished by 5 points if the action causes the robot to move away from the target. A robot will also be punished by 10 points if the certain move causes the robot to hit an obstacle or any part of the robot to move out of the visual field. The 100-point reward is given when a specific action causes the robot to arrive at the target position.

### 7.2.3 Discount Factor

The discounting factor  $\gamma$  defines how much expected future reward affects the current decisions. Low  $\gamma$  means pay little attention to the future or try to maximize immediate reward. High  $\gamma$  means that potential future rewards have a major influence on current decisions. The discount factor used in the experiment is 0.9.



#### 7.2.4 Exploration vs Exploitation

In the step 2.b of Algorithm 1, the learner chooses the action with maximum  $Q(s,a)$ . If the robot always chooses the best action, the learning will not occur. The performance of the robot will not improve. On the other hand, if the robot randomly chooses the action, the previously learnt result is not exploited. Therefore, we have to balance the exploration and exploitation. For a single state, there are formal techniques that give the optimal balance between the exploration and exploitation.

However for our cases, we deliberately chose 30 percent exploitation (i.e., choosing the best action) and 70 percent exploration (i.e., choosing a random action). In fact, the ratio between exploitation and exploration changes over time. During the first stage of learning, the learner has to explore. While during the last stage of learning, the learner should exploit the previously learnt result.

#### 7.2.5 When Learning is Completed

Step 2 of Q learning algorithm indicates that algorithm learns the task forever. However, in practice, the learning should be stopped when the robot finds a path to the target. Since there is no ways to tell if the condition is met except to test it. The test is done by moving the robot to the starting position and let the robot move for a specific number of moves. If the robot can arrive the target, the learning is then complete. The test is performed once in awhile during learning.

#### 7.2.6 Multi-step Q Learning

To speed up learning, the learning result should be propagated to the remote steps. We store a history of actions and update the Q values in the reverse chronological order. As found in Figure 7.1, we found that using the depth of 1,000 gives the best result. This approach is called multi-step Q learning as discussed in [33].

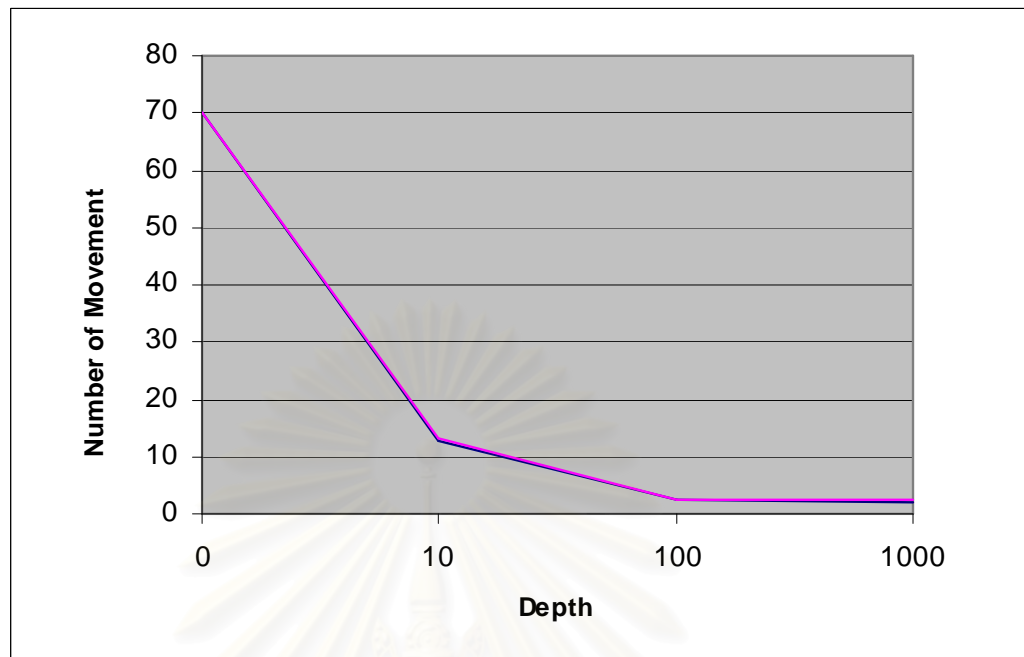


Figure 7.1: Updating a Q value

### 7.3 Comparing Q-Learning with Genetic Programming

Genetic programming and Q-learning are similar in that they are model-free learners. We compare the result of learning the robot task using GP and Q-learning in the following aspects.

- Learning time – We compare the time it took to learn the task. In fact, we compare the estimated time of performing Q-learning on the real robot and real-time for on-line evolution of a robot program.
- Quality of the result – We compare the trajectories of the robots that learned by Q learning and Genetic Programming.

### 7.3.1 Learning Time

Evolutionary learning and reinforcement learning are very time-consuming. Table 7.1 shows the comparison of the learning time of Genetic Programming and Q-learning.

This task can be learned by Q learning in about 2.7 million moves. Learning this task using Genetic Programming is twice faster. The evolution complete after 1.3 million moves. Assuming that each discrete movement of a real robot arm took 1 second, Q-learning will be completed in about 31 days on the real robot. Whereas, it would take about 16 days to evolved the robot program for the same task.

To speed up on-line learning time, we use memoized function to remember the effect of each movement. After applying the memoized function, Genetic Programming can learn the task in 4 hour while Q-learning can learn it in 43 hours. Genetic Programming make more efficient use of memoized function by having higher percent recall and use less entries in the memoize function. The percent recall is the percentage of time that the specific robot configuration found in the memory and thus does not have to make a real movement.

The row "Entries in memoized function" shows the total configuration the robot had been visited during learning. Q learning explored about 72 percent of all possible configuration while GP explores about 7 percent.

Table 7.1: Learning time of Q-learning and genetic programming.

	Q Learning	Genetic Programming
Real move	2,742,600	1,339,569
Recall	2,586,773	1,323,945
Percent Recall	0.94	0.99
Entries in memoried function	155,827	15,624
Percent Explore	0.72	0.07
Time (hours)	43.29	4.34

### 7.3.2 Quality of the result

The robot arm will stop when it reaches the target. We limit the number of movement to 300 steps. Within the limited steps, the trajectory of the result in simulation appears to be different. The trajectory of the end effector of a Q-learning and GP robot are shown in Figure 7.2 and Figure 7.3. The robot that runs a GP program hits the obstacles before they can avoid it. The robot that use Q-learning did not hit any obstacle. Therefore, it has smoother trajectory. This can be explained by the fact that, in Q learning, the action that causes the robot to hit the obstacle will be punished. That is the reason why the trajectory of the Q-learning avoid hitting any obstacle. In Genetic Programming, hitting an obstacle is not penalized in a fitness evaluation. Therefore, Genetic Programming does not generate a program that completely avoid the obstacles.



Figure 7.2: Trajectory of a robot arm controlled by a policy given by Q learning.

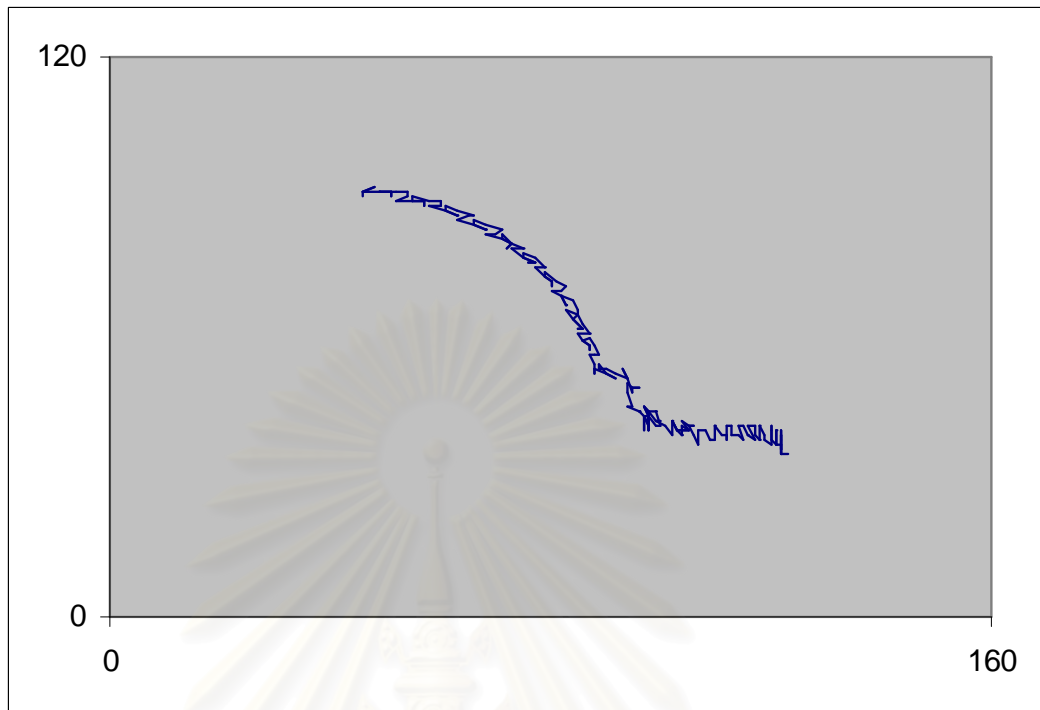


Figure 7.3: Trajectory of a robot arm controlled by a GP program.

#### 7.4 Conclusion

In this chapter, we compare the learning performance of Genetic Programming and Q learning. Both algorithms can learn the robot task without a robot model. To learn to perform a task, Q learning takes 10 times longer than GP. However, Q learning gives a result of higher quality. The trajectory is smoother and has fewer movements.

## CHAPTER VIII

### CONCLUSION

The main contributions of this thesis are as follows.

- The causes of fragile behavior when transferring the robot program evolved in simulation to the real world are identified and validated with the experiment.
- The improvement of robustness of robot programs generated by evolution is suggested and two methods are proposed.
- Off-line evolution using multiple robot configurations is proposed and validated with the experiment.
- On-line evolution is proposed with an important algorithm incorporated to make the method practical for experimentation. The results are validated with the experiment.
- The result from on-line evolution is analyzed and results in the suggestion of a framework for further study that the on-line evolution can be separated into two parts which can be implemented using different learners.

These contributions are elaborated and expanded in the following paragraphs.

Robot arm control programs for a visual-reaching task can be evolved off-line and on-line. The off-line evolution requires the model of a robot and the environment. Inaccurate models result in fragile behavior of a real robot. The robustness of a program evolved off-line can be improved by training the program with multiple robot configurations. However, this approach required the designer's intuition to decide what aspect of the real world that the robot program should be tolerant. Training the robot program to tolerate unimportant changes would not improve the robustness. For example, when

transferring a program that is evolved to tolerate change in target position in simulation [49] to a real robot arm, the result is not robust as a program that tolerates change in the robot configuration as reported in this thesis.

As opposed to off-line evolution, on-line evolution of a robot program does not require any model. On-line evolution is very suitable for a robot or an environment which models are difficult to obtain. For example, a robot created from servos connected randomly or Random Morphology robots as proposed by Dittrich et. al. [10]. However, on-line evolution is very time consuming. By using memoized function, the on-line evolution of a visual-reaching task can be sped up by about 23 times. It took less than one hour to evolve a control program on-line. Moreover, the resulting program works robustly in the testing environment.

The on-line evolution of a robot controller can be separated into two parts. The first part is to learn the robot and the world models. The sensory-action learner can be implemented as a function. A simplest learner is a memoized function. The performance of the memoized function can be measured as its ability to recall the effect of motions without having to move a robot physically. The second part is to learn how to perform the task. Genetic Programming is used to learn a visual-reaching task. Two learners are independent of each other.

## REFERENCES

- [1] Angeline, P. J. Subtree crossover: Building block engine or macromutation? In Proceedings of the Second Annual Conference Genetic Programming (1997) : 9-17.
- [2] Apornthewan, C. and Chongstitvatana, P. An on-line evolvable hardware for learning finite-state machine. In Proceedings of International Conference on Intelligent Technologies (2000) : 125-134.
- [3] Brooks, R. Artificial life and real robots. In Proceedings of European Conference on Artificial Life (1991) : 3-10.
- [4] Brooks, R. Intelligence without reason. In Proceedings of 12th International Joint Conference on Artificial Intelligence (1991) : 569-595.
- [5] Calderoni, S. and Marcenac, P. Genetic programming for automatic design of self-adaptive robots. In Proceedings of the First European Workshop on Genetic Programming (1998).
- [6] Cantu-Paz, E. A survey of parallel genetic algorithms. IlliGAL 97003 (Revised version of IlliGAL 95007). (1997).
- [7] Chongstitvatana, P. Using perturbation to improve robustness of solutions generated by genetic programming for robot learning. In Journal of Circuits, Systems and Computer 9, (1999) : 133-143.
- [8] Chongstitvatana, P. and Polvichai, J. Learning a visual task by genetic programming. In Proceedings of IEEE/RSJ International Conference on Intelligent robots and systems (1996) : 534-540.
- [9] Cliff, D.; Harvey, I.; and Husbands, P. Explorations in evolutionary robotics. In Adaptive Behaviour (1993) : 71-108.
- [10] Dittrich, P.; Burgel, R.; and Banzhaf, W. Learning to move a robot with random morphology. In Proceedings of First European Workshop on Evolutionary Robotics (1998) : 165-178.



- [11] Floreano, D. and Mondada, F. Evolution of homing navigation in a real mobile robot. In IEEE Transactions on Systems, Man and Cybernetics, Part B, 26 (1996) : 396-407.
- [12] Floreano, D. and Urzelai, J. Evolution of plastic control networks. In Autonomous Robots, 11 (2001) : 311-317.
- [13] Gathercole, C. and Ross, P. Small populations over many generations can beat large populations over few generations in genetic programming. In Proceedings of Genetic Programming (1997) : 111-118.
- [14] Goldberg, D.E. Genetic Algorithms in Search, Optimization and Machine Learning (n.p.): Addison-Wesley, 1989.
- [15] Halperin, D.; Kavradi, L.E.; and Latombe, J.C. Robot algorithms. In Algorithms and Theory of Computation Handbook (1999) : 21-1-21-21.
- [16] Harvey, I.; Husbands, P.; Cliff, D.; Thompson, A.; and Jakobi, N. Evolutionary Robotics: the Sussex Approach. In Robotics and Autonomous Systems (1997) : 205-224.
- [17] Holland, J. Adaptation in Natural and Artificial Systems. (n.p.): University of Michigan Press, 1975.
- [18] Ishiguro, A.; Tokura, S.; Kondo, T.; Uchikawa, Y.; and Eggenberger, P. Reduction of the gap between simulated and real environments in evolutionary robotics: a dynamically-rearranging neural network approach. In Proceedings of IEEE Systems, Man, and Cybernetic (1999) : 239-244.
- [19] Jakobi, N. Running across the reality gap: Octopod locomotion evolved in a minimal simulation. In Proceedings of Evorob98 (1998).
- [20] Jakobi, N. Minimal Simulations for Evolutionary Robotics. Ph.D. Thesis, School of Cognitive and Computing Sciences, University of Sussex, 1998.
- [21] Jakobi N.; Husbands P.; and Harvey I. Noise and the reality gap the use of simulation in evolutionary robotics. In Proceedings of the 3rd European Conference on

Artificial Life (1995) : 704-720.

- [22] Kavraki, L. E. and Latombe, J. C. Probabilistic roadmaps for robot path planning. In Practical Motion Planning in Robotics: Current Approaches and Future Directions (1998) : 33-53.
- [23] Kaelbling, L.P., Littman, M.L., and Moore, A.W. Reinforcement Learning: A Survey. In Journal of Artificial Intelligence Research (1996) : 237-285.
- [24] Koza, J. R. Genetic Programming: On the Programming of Computers by Means of Natural Selection. (n.p.): MIT Press, 1992.
- [25] Koza, J. R.; Bennett, F. H.; Keane, M. A.; and Andre, D. Automatic programming of a time-optimal robot controller and an analog electrical circuit to implement the robot controller by means of genetic programming. In Proceedings of IEEE International Symposium on Computational Intelligence in Robotics and Automation (1997).
- [26] Lee, W.; Hallam, J.; and Lund, H. H. A hybrid GP/GA approach for co-evolving controllers and robot bodies to achieve fitness-specified tasks. In Proceedings of the 1996 IEEE International Conference on Evolutionary Computation (1996).
- [27] Lee, W.; Hallam, J.; and Lund, H. H. Applying genetic programming to evolve behavior primitives and arbitrators for mobile robots. In Proceedings of the 1997 IEEE International Conference on Evolutionary Computation (1997) : 501-506.
- [28] Lozano-Perez, T.; Jones, J.; Mazer, E.; and O'Donnell, P. Handey: a robot task planner. MIT Press, 1992.
- [29] Luke, S. and Spector, L. A comparison of crossover and mutation in genetic programming. In Proceedings of Genetic Programming 1997 (1997) : 240-248.
- [30] Lund, H. H.; Hallam, J.; and Lee, W. Evolving robot morphology. In Proceedings of the 1997 IEEE International Conference on Evolutionary Computation (1997) : 197-202.
- [31] Lund, H. H. and Hallam, J. Evolving sufficient robot controllers. In Proceedings of

- the 1997 IEEE International Conference on Evolutionary Computation (1997) : 495-499.
- [32] Lund, H. H. and Miglino, O. From simulated to real robots. In Proceedings of IEEE 3rd International Conference on Evolutionary Computation (1996).
- [33] Mahadevan S. and Connell J. Automatic Programming of Behavior-based Robots using Reinforcement Learning. In National Conference on Artificial Intelligence (1991) : 768-773.
- [34] Mataric, M.J. and Cliff, D. Challenges in evolving controllers for physical robots. In Evolutional Robotics, special issue of Robotics and Autonomous Systems Vol. 19, (1996) : 67-83.
- [35] Miglino, O.; Lund, H.; and Nolfi, S. Evolving mobile robots in simulated and real environments. In Artificial Life (1995) : 417-434.
- [36] Mitchell T. Machine Learning. (n.p.): McGraw-Hill, 1997.
- [37] Nolfi, S.; Floreano, D.; Miglino, O.; and Mondada, F. How to evolve autonomous robots: Different approaches in evolutionary robotics. In Proceedings of Artificial Life IV (1994) : 190-197.
- [38] Nolfi, S. Power and the limits of reactive agents. In Neurocomputing Vol. 49, (2002) : 119-145.
- [39] Nopsuwanchai, R. and Chongstitvatana, P. Analysis of robustness of robot programs generated by genetic programming. In Proceedings of 1st Asian Symposium on Industrial Automation and Robotics (1999) : 211-215.
- [40] Olmer, M.; Nordin, P.; and Banzhaf, W. Evolving real-time behavioral modules for a robot with GP. In Proceedings of the 6th International Symposium on Robotics And Manufacturing (1996) : 675-680.
- [41] Poli, R. and Cagnoni, S. Genetic programming with user-driven selection: Experiments on the evolution of algorithms for image enhancement In Proceedings of Genetic Programming (1997) : 269-277.

- [42] Poli, R. and Langdon, W.B. A review of theoretical and experimental results on schemata in genetic programming. In Proceedings of the First European Workshop on Genetic Programming (1998).
- [43] Polvichai, J. Visually-guided reaching by genetic programming. Master Thesis, Computer Engineering, Chulalongkorn University, (in Thai) 1995.
- [44] Polvichai, J. and Chongstitvatana, P. Visually-guided reaching by genetic programming. In Proceedings of 2nd Asian Conf. on Computer Vision (1995).
- [45] Prateptongkum, M. and Chongstitvatana, P. Improving the robustness of a genetic programming learning. In Proceedings of 3rd Annual National Symposium on Computational Science and Engineering (1999) : 301-305.
- [46] Reiser, P. Evolutionary computation and the tinkerer's evolving toolbox. In Proceedings of the First European Workshop on Genetic Programming (1998).
- [47] Reynolds, C.W. The difficulty of roving eyes. In Proceedings of the First IEEE Conference on Evolutionary Computation (1994) : 262-267.
- [48] Soule, T. and Foster, J. A. Code size and depth flows in genetic programming. In Genetic Programming (1997) : 313-320.
- [49] Suwannik, W. and Chongstitvatana, P. Improving the robustness of evolved robot arm control programs generated by genetic programming. In Proceedings of International Conference on Intelligent Technologies (2000) : 149-153.
- [50] Suwannik, W. and Chongstitvatana, P. Improving the Robustness of Evolved Robot Arm Control Programs with Multiple Configurations. In Proceedings of Asian Symposium on Industrial Automation and Robotics (2001) : 87-90.
- [51] Suwannik, W. and Chongstitvatana, P., On-line Evolution of Robot Programs using a Memoized Function, In IEEE International Conference on Mechatronics and Machine Vision in Practice (2002).
- [52] Teller, A., and Andre, D. Automatically choosing the number of fitness cases: The rational allocation of trails. In Genetic Programming (1997) : 321-328.

- [53] Urzelai, J. and Floreano, D. Evolution of adaptive synapses: Robots with fast adaptive behavior in new environments. In Evolutionary Computation (2001) : 495-524.



สถาบันวิทยบริการ  
จุฬาลงกรณ์มหาวิทยาลัย

## Biography

Worasit Suwannik was born on the 25<sup>th</sup> of September 1973. His birthplace is Supanburi, Thailand. He had studied in that province until the primary school. After that, he came to Bangkok and had studied at Suankularb College for 5 years. He received a bachelor's degree in computer engineering from Chulalongkorn University. He received a master's degree in computer science from Vanderbilt University.



สถาบันวิทยบริการ  
จุฬาลงกรณ์มหาวิทยาลัย