Genetic Network Programming with Fuzzy Reinforcement Learning Nodes for Multi-Behaviour Robot Control

A thesis presented in partial fulfilment of the requirements for the

degree of

Masters of Science
In
Computer Science

Massey University – Albany Campus
New Zealand.

Wenhan Wang
2014

Abstract

This research explores a new approach for building a complex intelligent robot multi-behaviour comprising of a variety of intelligent subsystems that are fused together into one hybrid system. The work mainly focuses on integrating reinforcement learning and fuzzy logic with genetic network programming, examining the different architectures, and aims to achieve multi-objective behaviours and alleviate the problem of learning and calibration by repeated interaction with the environment. Different components of the learning algorithm are studied separately and also in combination. They are developed systematically using an increasing level of complexity for robot behaviours. As a test bed, the work investigates how to achieve ball pursuit and wall avoidance behaviours simultaneously, in the realm of the robot soccer game. The training procedure and test environment is designed, as well as a variety of fitness functions are experimented for the multi-behaviour objectives. Furthermore, the novel evolutionary architecture is combined with hill-climbing to accelerate the search for the best individual.

Keywords—robot soccer; multi-behaviour; multi-objectives; genetic network programming; fuzzy logic; reinforcement learning;

Acknowledgments

I would like to express my gratitude to all those who helped me during my research. My deepest gratitude goes first and foremost to Dr. Napoleon Reyes (Supervisor) and Dr. Andre Barczak (Co-Supervisor), for their constant encouragement and guidance. They have walked me through all the stages of this research. Without their consistent and illuminating instruction, this thesis could not have reached its present form. Thanks for the time you spent with me.

My thanks would also go to my beloved parents for their loving considerations and great confidence in me through all these years. I also owe my sincere gratitude to my uncle's family who gave me their help and time in supporting and helping me living in New Zealand.

Table of Contents

Abstract I
Acknowledgments II
List of Figures V
List of Tables VII
List of Pseudo Codes VII
Chapter 1 Introduction 1
1.1. Overview of the Current State of Technology 1
1.2. Research Objectives 2
1.3. Scope and Limitations of Research 2
1.4. Overview of the Problem Domain 3
1.5. Significance of the Research 3
1.6. Research Methodology 4
1.7. Structure of the Thesis Documentation 5
Chapter 2 Review of Related Literature 7
2.1. Fuzzy Logic Control 7
2.1.1. Fuzzy Sets and Membership 7
2.1.2. Algorithm Description 8
2.2. Reinforcement Learning 9
2.2.1. Markov Decision Process 9
2.2.2. General Description 10
2.2.3. Temporal Difference Learning 12
2.3. Genetic Network Programming 18
2.3.1 The Basics of GNP 18
2.3.2 Initialization the GNP 22
2.3.3 Running a GNP Individual 22
2.3.4 Genetic Operators 22
2.4. GNP with Reinforcement Learning 23
2.4.1. Basic Structure of GNP-RL 23
2.4.2. Running a GNP-RL Individual 25
2.5. Summary 26
Chapter 3 Adaptations of the Algorithms for Robot Control: Single Behaviour 27
3.1. Problem Domain Specifications 27
3.2. Algorithm 1: Fuzzy Logic Controller 28
3.2.1. General Architecture 28
3.2.2. Problem-Specific Parameter Settings 29

3.2.3. Experiment Results and Analysis 31
3.2.4. Limitations of the Algorithm 32
3.3. Algorithm 2: Reinforcement Learning with Fuzzy Logic 32
3.3.1 General Architecture 33
3.3.2 Problem-Specific Parameter Settings 35
3.3.3 Results and Analysis 36
3.3.4 Limitations of the Algorithm 40
3.4. Algorithm 3: Genetic Network Programming with Reinforcement Learning 41
3.4.1. General Architecture 41
3.4.2. Problem-Specific Parameter Settings 43
3.4.3. Results and Analysis 45
3.4.4. Limitations of the Algorithm 47
3.5. Summary 47
Chapter 4 Fuzzy-Reinforcement Learning for Robot Multi-behaviour 49
4.1 Introduction 49
4.2 General Architecture 50
4.3 Problem-Specific Parameter Settings 51
4.3.1 Fuzzy Logic Parameters 51
4.3.2 Reinforcement Learning Parameters 52
4.4. Results and Analysis 54
4.4 Limitations of the Algorithm 61
Chapter 5 GNP with Trained Fuzzy-RL Nodes for Learning Multi-Behaviours 63
5.1 General Architecture 63
5.2 GNP with Trained Fuzzy-RL Pseudo Code: Training Phase 66
5.3 Problem-Specific Settings 66
5.3.1. Judgment Nodes 66
5.3.2. Processing Nodes 67
5.3.3. GNP Fitness Function for Integrated Target Pursuit and Wall 69
5.3.4. Parameters for GNP 72
5.3.5. Hill-climbing Algorithm 72
5.4 Results and Analysis 73
5.5 Limitations of the Algorithm 78
Chapter 6 Summary and Future Work 79
Appendix A. Codes for the implementation of Fuzzy-RL 84
Appendix B. Codes for the implementation of GNP with trained Fuzzy-RL nodes ... 88
Reference 96

List of Figures

Figure 2.1 Classic sets and fuzzy sets 8
Figure 2.2 Sample fuzzy sets for distance 8
Figure 2.3 Structure of genetic network programming 18
Figure 2.4 Structure of the gene of a node 19
Figure 2.5 Schematic diagram of the genetic network programming algorithm (training phase) 20
Figure 2.6 Schematic diagram of genetic network programming (testing phase) 21
Figure 2.7 Judgment and Processing Node in GNP 22
Figure 2.8 Processing node and judgment node with sub-nodes 24
Figure 2.9 Schematic diagram of GNP- RL running in the testing phase 25
Figure 3.1 2D simulation environment 28
Figure 3.2 Flowchart of fuzzy logic control system 28
Figure 3.3 Fuzzy logic system design (NL-Negatively Large, NM-Negatively Medium, NS-Negatively Small, ZE-Zero, PS-Positively Small, PM-Positively Medium and PL-Positively Large) 29
Figure 3.4 Angle fuzzy sets 29
Figure 3.5 Distance Fuzzy Sets 30
Figure 3.6 Trace of ball and robot (fuzzy logic controller) 31
Figure 3.7 Schematic diagram of RL with fuzzified input algorithm 33
Figure 3.8 Schematic diagram of the Fuzzy-RL algorithm 34
Figure 3.9 Angle Fuzzy Sets 35
Figure 3.10 Initial part of RL with fuzzified input algorithm 37
Figure 3.11 Results of Algorithm 2a: Average angle from ball every 50 time steps (y-axis = ave. angle; x-axis: 1 unit = 50 time steps). 37
Figure 3.12 Results of Algorithm 2b: Learning phase of the Fuzzy-RL algorithm. 38
Figure 3.13 Results of Algorithm 2b: Average angle from ball every 50 time steps (y-axis = ave. angle; x-axis: 1 unit = 50 time steps). 38
Figure 3.14 The performance after running for a while (RL with fuzzified input algorithm) 39
Figure 3.15 The performance after running for a while (Fuzzy-RL algorithm) 40
Figure 3.16 Schematic diagram of GNP with RL for training phase 41
Figure 3.17 Schematic diagram of GNP with RL for testing phase 43
Figure 3.18 Judgment node settings 43
Figure 3.19 Processing node settings 44
Figure 3.20 Fitness of the best individual (y-axis = fitness; x-axis = generation count)

.....45
Figure 3.21 Performance of the GNP with RL 46
Figure 4.1 Calculation of difference between the heading angle of the ball, and the nearest wall 49
Figure 4.2 Schematic diagram of the Fuzzy-RL algorithm 50
Figure 4.3 Angle from ball Fuzzy Sets 51
Figure 4.4 Angle from wall Fuzzy Sets 51
Figure 4.5 Distance from wall Fuzzy Sets 52
Figure 4.6 Average angle from ball (measured every 500 time steps) during robot training. 54
Figure 4.7 Pre-defined restricted area used in the experiments. The ball is initially placed within the black region depicted in the figure. The white region is the prohibited area. 55
Figure 4.8 Close to the wall counts every 500 time steps 55
Figure 4.9 Trained sample close to wall 1 performance 1 57
Figure 4.10 Trained sample close to wall 1 performance 2 57
Figure 4.11 Trained sample close to wall 2 performance 1 58
Figure 4.12 Trained sample close to wall 2 performance 2 58
Figure 4.13 Trained sample close to wall 3 performance 1 59
Figure 4.14 Trained sample close to wall 3 performance 2 59
Figure 4.15 Trained sample close to wall 4 performance 1 60
Figure 4.16 Trained sample close to wall 4 performance 2 60
Figure 5.1 Modified GNP Individual used in the new algorithm 63
Figure 5.2 Schematic diagram of GNP with trained Fuzzy-RL nodes algorithm 64
Figure 5.3 Judgment node settings 67
Figure 5.4 Absolute angle of the robot relative to the field 67
Figure 5.5 Angle from ball fuzzy sets 70
Figure 5.6 Sample GNP individual with the minimum number of nodes. Note that the algorithm may generate a variety of individuals with different nodes and connections. 73
Figure 5.7 General performance of a good individual 74
Figure 5.8 The performance close to wall 1 75
Figure 5.9 The performance close to wall 2 75
Figure 5.10 The performance close to wall 3 76
Figure 5.11 The performance close to wall 4 76
Figure 5.12 Fitness of top 3 individuals with hill climbing 77
Figure 5.13 Fitness of top 3 individuals without hill climbing 77

List of Tables

Table 2.1 State-action space for RL in GNP-RL algorithm 24
Table 3.1 Fuzzy Associative Memory Matrix for Ball Pursuit: Steering Angle Adjustment 30
Table 3.2 Fuzzy Associative Memory Matrix for Ball Pursuit: Speed Control 31
Table 3.3 State-Action space (States are the truth value from Fuzzy system, Actions are steering angles for the robot) 36
Table 3.4 The state-action space of RL 42
Table 3.5 Performance data of top five individuals 46
Table 4.1 The ID of states for corresponding input combination (Distance from wall is near) 52
Table 4.2 State-Action space (y-axis: ID of RL States, x-axis: actions) 53
Table 5.1 Fuzzy rules for calculating the ball pursuit behaviour fitness. 70
Table 6.1 Comparison of different algorithms 79

List of Pseudo Codes

Pseudo code 1: TD(0) algorithm (Sutton, et al., 2012) 12
Pseudo code 2: Sarsa (On-Policy) algorithm (Sutton, et al., 2012) 13
Pseudo code 3: Q-Learning (On-Policy) algorithm (Sutton, et al., 2012) 13
Pseudo code 4: TD(λ) algorithm (Sutton, et al., 2012) 16
Pseudo code 5: SARSA(λ) algorithm (Sutton, et al., 2012) 17
Pseudo code 6: Q(λ) algorithm (Sutton, et al., 2012) 17
Pseudo code 7: Reward function for RL with FLS 36
Pseudo code 8: Reward function for GNP-RL 44
Pseudo code 9: Reward function for ball pursuit and wall avoidance 54
Pseudo code 10: GNP with Trained Fuzzy-RL 66
Pseudo code 11: Fitness function for speed control behaviour 69
Pseudo code 12: Fitness function for wall avoidance 70
Pseudo code 13: Final fitness function 71
Pseudo code 14: Hill-climbing algorithm 72

Chapter 1
Introduction

1.1. Overview of the Current State of Technology

A large number of studies have been made on automatic design of behaviour sequences for agents, such as the sequence to carry out some tasks in the virtual world. The experiments of creating artificial life aiming to realize the behaviours of ants or fishes are a good example, as well as the planning for real mobile robots which have a simple object in the real world. Many models used to express such behaviour sequences for agents have been proposed. Many of these models use fuzzy logic, supervised leaning, reinforcement learning, and evolutionary optimization techniques such as Genetic Algorithm, Evolution Strategy and Genetic Programming. As a result, there are still some common problems that are hard to be addressed, such as enormous computational cost during training time, expert pre-knowledge required or poor ability in terms of adjustability in dynamic environments.

Reinforcement learning is still a powerful algorithm for robot control, and several modifications and combinations of reinforcement learning with other algorithms show a strong potential for improvement. Some of these developments have sophisticated framework, and some of them run extremely fast (Pitoyo, et al., 2009). Recently, a graph structure based evolutionary algorithm Genetic Network Programming was proposed (Katagiri, et al., 2000). Within twelve years, this algorithm has been tested and combined with other algorithms, showing a solid capability to deal with dynamic environments. The latest research combined Genetic Network Programming, Fuzzy Logic and Reinforcement Learning together, and presented a strong potential to overcome weaknesses mentioned above. It has been tested by a wall following robot and the result was much better than conventional methods in dynamic environments (Mabu, et al., 2011).

1.2. Research Objectives

The main objective of this research is to give agents the ability to learn a complex behaviour by themselves in dynamic environments, by using hybrid algorithms. To the best knowledge of the author, the combination of genetic network programming, fuzzy logic and reinforcement learning has not been tested yet to achieve a complex multi-behaviour through interacting with the environment. This research aims to study the aforementioned algorithms and improve and extend them for learning more complex multi-behaviour operations.

Specific Objectives

1. Develop, test and analyse a simple robot behaviour using a cascade of fuzzy logic control systems.
2. Develop, test and analyse a simple robot behaviour using reinforcement learning.
3. Develop, test and analyse a simple robot behaviour using reinforcement learning and fuzzy logic system.
4. Develop, test and analyse a simple robot behaviour using genetic network programming with reinforcement learning.
5. Find other possible learning schemes to achieve multiple behaviour operations for a robot.
6. Develop, test and analyse multiple behaviour functionality with fuzzy-reinforcement learning algorithm.
7. Develop, test and analyse the genetic network programming with fuzzy-reinforcement learning nodes algorithm to achieve multiple behaviour operations for a robot.

1.3. Scope and Limitations of Research

1. All experiments should run in a simulation environment. There are limitations to the physics involved in the simulations.
2. The experimental subject is a simulated bi-wheel robot, which is only able to control the speed and steering angle by controlling the velocity of the two wheels independently.

3. The testing field in the simulation environment complies with the FIRA Roboworld Cup standard soccer field.
4. The multiple behaviours for the robot in the tests are limited to three simple tasks: target pursuit, speed control and wall avoidance.

1.4. Overview of the Problem Domain

The problem domain is limited to the FIRA micro-soccer robot. It only has two wheels and it is a popular test bed for artificial intelligent programs. The hybrid algorithm will be tested in a simulation environment where all sizes are the same as the real robot soccer. This research focuses on developing multiple behaviour operations for a bi-wheel robot. Two different behaviours are tested, ball pursuit (with speed control) and wall avoidance. The simulation program updates the status of the ball and robot every time step. All the simulations in this thesis only consider the position, speed and direction of the objects. Thus the angle and distance from ball to robot, or the angle and distance from wall to robot can be obtained as input values for the robot control algorithm. Moreover, the control system is able to adjust the speed and the direction of the robot directly.

1.5. Significance of the Research

This research presents a novel architecture for combing Fuzzy logic, reinforcement learning and genetic network programming. The new architecture inherits the flexible attribute from genetic network programming and adds complex processing nodes into the system (this research uses Fuzzy-Reinforcement nodes), making the new framework to be able to deal with complex multiple robot behaviours. The ball pursuit and wall avoidance behaviour for robot was achieved, as well as an efficient training methodology, involving a fitness function. As compared to the other works (Mabu, et al., 2011) for robot control using the GNP algorithm, the experiments in this research proves that the new architecture has strong adaptability mechanisms for learning multi-objectives.

1.6. Research Methodology

1. Study cascaded fuzzy logic systems.
2. Write a program to test a Fuzzy logic system for achieving a simple task.
3. Study Reinforcement Learning Algorithm.
4. Write a program to test Reinforcement Learning for achieving a simple task.
5. Study Genetic network programming.
6. Write a program to test Genetic network programming for achieving a simple task.
7. Follow the existing genetic network programming with reinforcement learning scheme to implement ball pursuit behaviour for robot.
8. Combine fuzzy logic with reinforcement learning. Write a program to test ball pursuit behaviour for robot.
9. Improve the performance of fuzzy logic with reinforcement learning for ball pursuit.
10. Write a program to achieve a complex multi-behaviour by using fuzzy logic with reinforcement learning.
11. Write a program to achieve a complex multi-behaviour by using genetic network programming with Fuzzy-reinforcement learning nodes.
12. Test and compare the GNP with Fuzzy-RL nodes scheme with Fuzzy-RL scheme.
13. Through experiments, characterise all the aforementioned algorithms, in terms of performance efficiency, flexibility and adaptability.

1.7. Structure of the Thesis Documentation

This document is comprised of 6 chapters. Chapter 2 provides a theoretical framework and detailed algorithms of fuzzy logic control system, reinforcement learning and genetic network programming.

Chapter 3 presents three different algorithms to accomplish implementing a single robot behaviour – ball pursuit (with speed control). Detailed implementation and results are discussed in this chapter, as well as the limitation of each algorithm.

Chapter 4 presents the Fuzzy-RL algorithm for achieving multi-behaviour for robot. Ball pursuit and wall avoidance are implemented and tested. Detailed implementation and testing results also shows in this chapter.

Chapter 5 presents the novel architecture - GNP with trained Fuzzy-RL nodes for multi-behaviour. Ball pursuit and wall avoidance are implemented and tested, in contrast with Chapter 4, it is able to distinguish different walls for different actions.

Lastly, chapter 6 summaries the whole document, and identifies promising areas of research worthy of conducting future works.

Chapter 2
Review of Related Literature

2.1. Fuzzy Logic Control

Fuzzy logic has a very long history, the term is introduced in 1965 by Lotfi A. Zadeh (Zadeh, 1965) with the proposal of fuzzy set theory and it has been studied even earlier. Fuzzy logic has been applied to many field and wildly used in industry as a control system. It is a form of many-valued logic or multi-value logic, it gives an approximation rather than fixed and exact.

2.1.1. Fuzzy Sets and Membership

For crisp sets, an element x in the universe X is either a member of some crisp set A or not. This binary issue of membership can be represented mathematically as below,

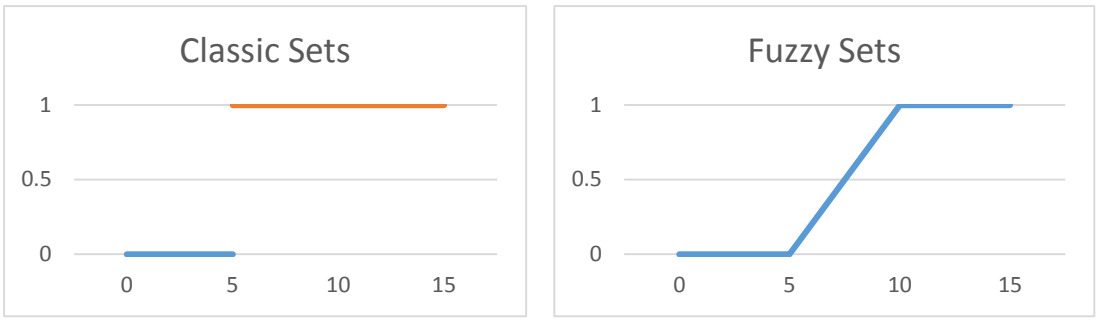
$$X_A(x) = \begin{cases} 1, x \in A \\ 0, x \notin A \end{cases} \quad \textit{(1) (Berenji, et al., 1992)}$$

Zadeh extended the notion of binary membership to accommodate various "degrees of membership" on the real continuous interval [0, 1]. The membership function embodies the mathematical representation of membership in a set, and the notation used throughout this text for a fuzzy set is a set symbol with a tilde underscore, say $\underline{A}$, where the functional mapping is given as

$$\mu_{\underline{A}}(x) \in [0,1] \quad \textit{(2)}$$

The symbol $\mu_{\underline{A}}(x)$ (Ross, et al., 2002) is the degree of membership of element x in fuzzy set $\underline{A}$.

For example, to describe the distance between two objects, it can be presented as (a) by using classic sets and (b) by using fuzzy sets in Figure 2.1.



(a) (b)

Figure 2.1 Classic sets and fuzzy sets

2.1.2. Algorithm Description

Fuzzy control can be divided into three steps: fuzzification, rule evaluation and defuzzification.

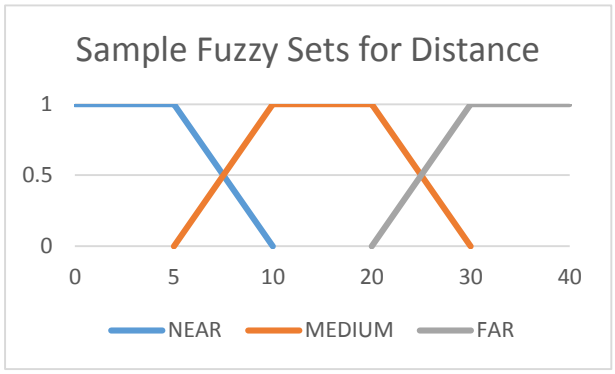


Figure 2.2 Sample fuzzy sets for distance

In the first step fuzzification, the real input value will be mapped to a truth value in the 0 to 1 range by each membership function. Use Fig. 2.2 for instance, suppose the input value is 25. Thus the truth values for distance can be written as follows.

$$\mu NEAR(25) = 0$$
$$\mu MEDIUM(25) = 0.5$$
$$\mu FAR(25) = 0.5$$

Fuzzy logic usually use IF THEN statements for rules, and these rules will take truth values to the final fuzzy value.

For instance:

Rule 1: If distance is NEAR then low speed

Rule 2: if distance is MEDIUM then medium speed

Rule 3: if distance is FAR then high speed

Now the firing degree of low speed is 0, the firing degree of medium speed is 0.5 and the firing degree of high speed is 0.5.

Suppose the value of low speed is 1, medium speed is 5 and high speed is 10. In the last step defuzzification, it uses a centroid method to reach the final output.
Centre of mass formula:

$$COM = \frac{\sum_{i=1}^{N} m_i x_i}{\sum_{i=1}^{N} m_i} \quad \textit{(3) (Berenji, et al., 1992)}$$

Thus the output is:

$$output = \frac{0 * 1 + 0.5 * 5 + 0.5 * 10}{0 + 0.5 + 0.5} = 7.5$$

2.2. Reinforcement Learning

Reinforcement learning addresses the problem of how an autonomous agent that senses and acts in its environment can learn to choose optimal actions to achieve its goals. The problem, due to its generality, covers tasks such as learning to control a mobile robot, learning to optimize operations in factories, and learning to play board games. The major difference between reinforcement learning and other forms of machine learning is that learner is not told which actions to take, but discover which actions yield the most reward by trying them. This section introduces the basic schematics of reinforcement learning and two well-known algorithms, Sarsa (Rummery, et al., 1994) and Q-Learning (Watkins, et al., 1992).

2.2.1. Markov Decision Process

In machine learning, the environment is typically formulated as a Markov decision process (MDP) (Bellman, 1957) and reinforcement learning is not an exception. Markov decision processes (MDP), named after Andrey Markov, provide a mathematical framework for modelling decision making in situations where outcomes are partly random and partly under the control of a decision maker. The essence of MDP is that a future state is determined only by the current state, because the current state already contains sufficient information for determine the next state.
Markov Decision Process is defined by four elements **S**, **A**, **R**, **P**. (Faria, Gedson, et al., 2000)
S is a finite set of states
A is a finite set of actions
$\mathbf{R}^{\mathbf{a}}_{\mathbf{ss'}}$ is the immediate reward received after taken action **a** transition from current

state **s** to next state **s'**.

$\mathbf{P^a_{ss'}}$ is the probability that action **a** in state **s** at time t will lead to state **s'** at time **t+1**.

2.2.2. General Description

MDP can solve the optimal policy for choosing actions by using Dynamic Programming if the R function and P function are known. Reinforcement Learning is used to learn the optimal policy when P function and R function are unknown. The four main elements (Bonarini, et al., 2009) of a reinforcement learning system are: a **policy**, a **reward function**, a **value function**, and, optionally, a **model of the environment**.

A **policy** defines the learning agent's way of behaving at a given time (when selecting actions).

A **reward function** defines the goal in a reinforcement learning problem. It maps each perceived state (or state-action pair) of the environment to a single number, a reward.

A **value function** specifies what is good in the long run, whereas a **reward function** indicates what is good in an immediate sense.

A **model of the environment** is something that mimics the behaviour of the environment. (e.g., a simulation program (although that in itself is NOT the model...))

A policy, **π**, is a mapping from each state, $\boldsymbol{s \epsilon S}$, and action, $\boldsymbol{a \epsilon A}$, to the probability **π(s,a)** of taking action **a** when in state **s**. Informally, the value of a state **s** under a policy **π**, denoted $\mathbf{V^{\pi}(s)}$, is the expected return when starting in **s** and following **π** thereafter. For MDPs, $\mathbf{V^{\pi}(s)}$ is defined formally as

$$V^{\pi}(s) = E_{\pi}\{R_t | s_t = s\} = E_{\pi}\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s\} \quad \textit{(4) (Sutton, et al., 2012)}$$

Where $E_{\pi}\{\}$ denotes the expected value given that the agent follows policy **π**, and **t** is any time step.

$$V^{\pi}(s) = E_{\pi}\{R_t | s_t = s\}$$

$$= E_{\pi}\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s\right\}$$

$$= E_{\pi}\left\{r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} | s_t = s\right\}$$

$$= \sum_{a} \pi(s,a) \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma E_{\pi}\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+2} | s_{t+1} = s'\right\}\right]$$

$$= \sum_a \pi(s,a) \sum_{s'} P^a_{ss'} [R^a_{ss'} + \gamma V^{\pi}(S')] \quad \textit{(5) (Sutton, et al., 2012)}$$

The equation above is the Bellman equation for $\mathbf{V^{\pi}}$. It expresses a relationship between the value of a state and the values of its successor states.
In reinforcement learning $\mathbf{R^a_{ss'}}$ and $\mathbf{P^a_{ss'}}$ are unknown, so it is impossible to update the value function by using the equation above. By using **Monte Carlo Methods** (Metropolis, et al., 1949), keeping the policy unchanged and keeping training the system repeatedly, one can use the estimated value function as below:

$$V(s_t) \leftarrow V(S_t) + \alpha[R_t - V(S_t)] \quad \textit{(6) (Sutton, et al., 2012)}$$

where $\mathbf{s_t}$ is the state visited at time **t**, $\mathbf{R_t}$ is the reward after time t and α is a constant parameter.
This estimated value function will eventually be an approximation of equation (5).

There are three common policies (Sutton, et al., 2012) used for action selection. The aim of these policies is to balance the trade-off between exploitation and exploration, by not always exploiting what has been learnt so far.
ε-greedy - most of the time the action with the highest estimated reward is chosen, called the greediest action. Every once in a while, say with a small probability ε, an action is selected at random. The action is selected uniformly, independent of the action-value estimates.
ε-soft - very similar to ε-greedy. The best action is selected with probability 1 - ε and the rest of the time a random action is chosen uniformly.
softmax - one drawback of ε-greedy and ε-soft is that they select random actions uniformly. The worst possible action is just as likely to be selected as the second best.

Softmax remedies this by assigning a rank or weight to each of the actions, according to their action-value estimate. A random action is selected with regards to the weight associated with each action, meaning the worst actions are unlikely to be chosen. This is a good approach to take where the worst actions are very unfavourable.

On-Policy (Poole, et al., 2010) Temporal Difference methods (introduced in next section) learn the value of the policy that is used to make decisions. The value functions are updated using results from executing actions determined by some policy. On the other hand, **Off-Policy** (Poole, et al., 2010) methods can learn different policies for behaviour and estimation.

2.2.3. Temporal Difference Learning

1-step TD Prediction TD(0) (Sutton, et al., 2012)

Temporal Difference (TD) Learning methods can be used to estimate these value functions. If the value functions were to be calculated without estimation, the agent would need to wait until the final reward was received before any state-action pair values can be updated. Once the final reward was received, the path taken to reach the final state would need to be traced back and each value updated accordingly (Monte Carlo Methods). On the other hand, with TD methods, an estimate of the final reward is calculated at each state and the state-action value updated for every step of the way. Expressed formally:

$$V(s_t) \leftarrow V(s_t) + \alpha[r_{t+1} + \gamma V(S_{t+1}) - V(S_t)] \quad (6)$$

where $\mathbf{r_{t+1}}$ is the observed reward at time **t+1**. The pseudo code of TD(0) algorithm is shown as Pseudo code 1.

Pseudo code 1: TD(0) algorithm (Sutton, et al., 2012)

1. Initialize V(s) arbitrarily, π to the policy to be evaluated
2. **Repeat** (for each episode)
3. Initialize s
4. **Repeat** (for each step of episode)
5. a $\leftarrow$ action given by π for s
6. Take action a; observe reward, r, and next state, s'
7. $V(s) \leftarrow V(s) + \alpha[r + \gamma V(s') - V(s)]$
8. $s \leftarrow s'$
9. **Until** s is terminal

In the actual application, V(s) cannot be acquired directly, and Q(s, a) is introduced as an estimation of V(s).

The value of taking action **a** in state **s** under a policy **π** is defined as below, denoted **Q^{π}(s,a)**, as the expected return starting from **s**, taking the action **a**, and thereafter following policy π:

$$Q^{\pi}(s,a) = E_{\pi}\{R_t | s_t = s, a_t = a\} = E_{\pi}\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a\} \quad (7)$$

Q^{π} is called the action-value function for policy **π**.

Therefore, two implementable TD(0) algorithm show as follows (Pseudo code 2, Pseudo code 3):

Pseudo code 2: Sarsa (On-Policy) algorithm (Sutton, et al., 2012)

1. Initialize Q(s,a) arbitrarily
2. **Repeat** (for each episode)
3. Initialize s
4. Choose a from s using policy derived from Q (e.g. ε-greedy)
5. **Repeat** (for each step of episode)
6. Take action a, observe r, s'
7. Choose a' from s' using policy derived from Q (e.g. ε-greedy)
8. $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma Q(s',a') - Q(s,a)]$
9. $s \leftarrow s'; a \leftarrow a'$
10. **Until** s is terminal

Pseudo code 3: Q-Learning (On-Policy) algorithm (Sutton, et al., 2012)

1. Initialize Q(s,a) arbitrarily
2. **Repeat** (for each episode)
3. Initialize s
4. **Repeat** (for each step of episode)
5. Choose a' from s' using policy derived from Q (e.g. ε-greedy)
6. Take action a, observe r, s'
7. $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma Q(s',a') - Q(s,a)]$
8. $s \leftarrow s'; a \leftarrow a'$
9. **Until** s is terminal

n-step TD Prediction TD(λ) (Sutton, et al., 2012)

In Monte Carlo backups the estimate value function is updated in the direction of the complete return:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots + \gamma^{T-t-1} r_T \quad (8)$$

where T is the last time step of the episode.
In one-step backups the target is the first reward plus the discounted estimated value of the next state:

$$R_t^{(1)} = r_{t+1} + \gamma V_t(s_{t+1}) \quad (9)$$

So, the n-steps target is:

$$R_t^{(n)} = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots + \gamma^{n-1} r_{t+n} + \gamma^n V_t(s_{t+n}) \quad (10)$$

The increment to $\mathbf{V_t(S_t)}$ (the estimated value of $\mathbf{V^{\pi}(s_t)}$ at time **t**), due to an n-step backup of $\mathbf{s_t}$, is defined by

$$\Delta V_t(S_t) = \alpha[R_t^{(n)} - V_t(s_t)] \quad (11)$$

where α is a positive step-size parameter, as usual.
The increments to the estimated values of the other states are

$$\Delta V_t(s) = 0 \; for \; all \; s \neq s_t \quad (12)$$

In on-line updating, the updates are done during the episode, as soon as the increment is computed. In this case

$$V_{t+1}(s) = V_t(s) + \Delta V_t(s) \quad (13)$$

However, n-step TD methods are rarely used because they are inconvenient to implement. Computing n-step returns requires waiting n steps to observe the resultant rewards and states. For large n, this can become problematic, particularly in control applications.

Backups can be done not just toward any n-step return, but toward any average of n-step returns. For example, a backup can be done toward a return that is half of a two-step return and half of a four-step return:

$$R_t^{ave} = \frac{1}{2}R_t^{(2)} + \frac{1}{2}R_t^{(4)} \quad (14)$$

Any set of returns can be averaged in this way, even an infinite set, as long as the weights on the component returns are positive and sum to 1.

The TD(λ) algorithm can be understood as one particular way of averaging n-step backups. This average contains all the n-step backups, each weighted proportional to λn-1, where 0≤λ≤1. A normalization factor of 1-λ ensures that the weights sum to 1. The resulting backup is toward a return, called the λ-return, defined by

$$R_t^{\lambda} = (1-\lambda)\sum_{n=1}^{T-t-1}\lambda^{n-1}R_t^{(n)} + \lambda^{T-t-1}R_t \quad (15)$$

If λ=0, then the overall backup reduces to its first component, the one-step TD backup.

If λ=1, then the overall backup reduces to its last component, the Monte Carlo backup.

On each step **t**, it computes an increment for state visited

$$\Delta V_s(s_t) = \alpha[R_t^{\lambda} - V_t(s_t)] \quad (16)$$

The increments to the estimated values of the other states are

$$\Delta V_t(s) = 0 \; for \; all \; s \neq s_t \quad (17)$$

Eligibility Trace (Barto, et al., 1983)

An eligibility trace is associated with each state. The eligibility trace for state **s** at time **t** is denoted. On each step, the eligibility traces for all states decay by γλ, and the eligibility trace for the one state visited on the step is incremented by 1:

$$e_t(s) = \begin{cases} \gamma\lambda e_{t-1}(s) & if \; s \neq s_t \\ \gamma\lambda e_{t-1}(s) + 1 & if \; s = s_t \end{cases} \quad (18)$$

This kind of eligibility trace is called an accumulating trace because it accumulates each time the state is visited, then fades away gradually when the state is not visited. The traces indicate the degree to which state is eligible for undergoing learning changes should a reinforcing event occur. The reinforcing events are the moment-by-moment one-step TD errors. For example, the TD error for state-value prediction is

$$\delta_t = r_{t+1} + \gamma V_t(s_{t+1}) - V_t(s_t) \quad (19)$$

The global TD error signal triggers proportional updates to all recently visited states, as signalled by their nonzero traces:

$$\Delta V_t(s) = \alpha \delta_t e_t(s) \;\; for\; all\; s \in S \quad (20)$$

Pseudo code 4: TD(λ) algorithm (Sutton, et al., 2012)

1. Initialize V(s) arbitrarily and e(s)=0, for all s∈ S
2. **Repeat** (for each episode)
3. Initialize s
4. **Repeat** (for each step of episode)
5. a ← action given by π for s
6. Take action a; observe reward, r, and next state, s'
7. $\delta \leftarrow r + \gamma V(s') - V(s)$
8. $e(s) \leftarrow e(s) + 1$
9. For all s:
10. $V(s) \leftarrow V(s) + \alpha \delta e(s)$
11. $e(s) \leftarrow \gamma \lambda e(s)$
12. $s \leftarrow s'$
13. **Until** s is terminal

In Pseudo code 4, if λ=0 it becomes completely same with TD(0) algorithm, and if λ=1 this turns out to be just the right thing to do to achieve Monte Carlo behaviour.

Almost any temporal-difference (TD) method, such as Q-learning or Sarsa, can be combined with eligibility traces to obtain a more general method that may learn more efficiently. The SARSA(λ) algorithm and Q(λ) algorithm are shown as follows (Pseudo code 5, Pseudo code 6).

Pseudo code 5: SARSA(λ) algorithm (Sutton, et al., 2012)

1. Initialize Q(s,a) arbitrarily and e(s,a)=0, for all s,a
2. **Repeat** (for each episode)
3. Initialize s,a
4. **Repeat** (for each step of episode)
5. Take action a; observe reward, r, and next state, s'
6. Choose a' from s' using policy derived from Q (e.g. ε-greedy)
7. $\delta \leftarrow r + \gamma Q(s', a') - Q(s, a)$
8. $e(s, a) \leftarrow e(s, a) + 1$
9. For all s,a:
10. $Q(s, a) \leftarrow Q(s, a) + \alpha\delta e(s, a)$
11. $e(s, a) \leftarrow \gamma\lambda e(s, a)$
12. $s \leftarrow s'; \;\; a \leftarrow a'$
13. **Until** s is terminal

Pseudo code 6: Q(λ) algorithm (Sutton, et al., 2012)

1. Initialize Q(s,a) arbitrarily and e(s,a)=0, for all s,a
2. **Repeat** (for each episode)
3. Initialize s,a
4. **Repeat** (for each step of episode)
5. Take action a; observe reward, r, and next state, s'
6. Choose a' from s' using policy derived from Q (e.g. ε-greedy)
7. $a^* \leftarrow \arg max_b Q(s', a') \; (if \; a' ties \; for \; the \; max, then \; a^* \leftarrow a')$
8. $\delta \leftarrow r + \gamma Q(s', a') - Q(s, a)$
9. $e(s, a) \leftarrow e(s, a) + 1$
10. For all s,a:
11. $Q(s, a) \leftarrow Q(s, a) + \alpha\delta e(s, a)$
12. If $a' = a^* \; then \; e(s, a) \leftarrow \gamma\lambda e(s, a)$
13. Else $e(s, a) \leftarrow 0$
14. $s \leftarrow s'; \;\; a \leftarrow a'$
15. **Until** s is terminal

2.3. Genetic Network Programming

2.3.1 The Basics of GNP

Phenotype of the GNP

The GNP uses a directed graph as its phenotype (Katagiri, et al., 2000). It contains three types of nodes, one start node, judgment nodes and processing nodes. The basic structure is shown in the Fig. 2.2. The only function of the start node is to determine which node to execute first, and the rest nodes never direct to the start node. The judgment node provides the ability to judge the environment. It has multiple connections to other nodes, and typically using if-then statements to process the input values in order to decide which connection to select. On the other hand, the processing node represent the actual action of an agent, e.g. a steering angle of a robot. It only has one connection to another node.

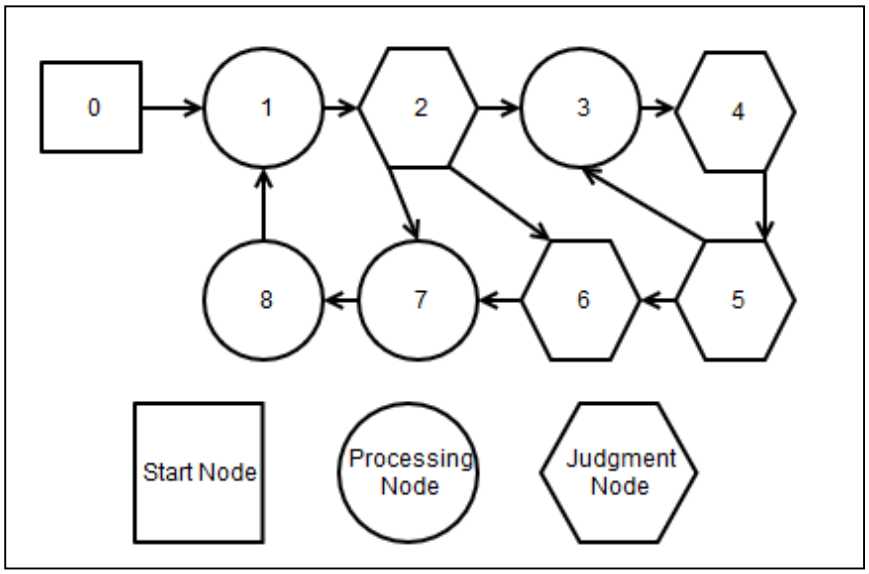


Figure 2.3 Structure of genetic network programming

GNP has time delays (Katagiri, et al., 2000) on executing judgment nodes and processing nodes. It can be considered as how much time consumed by executing the node, because when applying the GNP in the real world, the agent needs time to judge the environment and take actions. Processing nodes and judgment nodes may have different time delays, and it needs a maximum time to determine how many nodes can be executed in one time step of a simulation program or a real world control system. For example, set the maximum time to five units, set time delay of the judgment node to one unit and processing node to three units, so the GNP may execute two judgment nodes and one processing node in one step.

Once the GNP starts, it will execute nodes one by one according to the connection, in an infinite loop. The loop can only be stopped externally. In the training phase (introduced later), an execution time is a typical termination condition for GNP, while in the testing phase (introduced later), it can be terminated whenever there is no need to continue the system.

Genotype of the GNP

ID	T	F	D	C_1	C_2	……

Figure 2.4 Structure of the gene of a node

The structure of the gene (Katagiri, et al., 2000) of a node is presented as Fig. 2.3. ID is a unique number of the node and it never changes. T is the type of the node (0: start node, 1: judgment node, 2: processing node). The GNP judgment node and processing node may have multiple functions (Katagiri, et al., 2000), F is the function type of a node. D is the time delay of a node. Finally Ci is the connection to the next node which is the most important part that decides the whole structure of the GNP. A complete GNP includes many nodes, so a complete GNP individual is in a two dimensional structure.

Schematic Diagram of the GNP

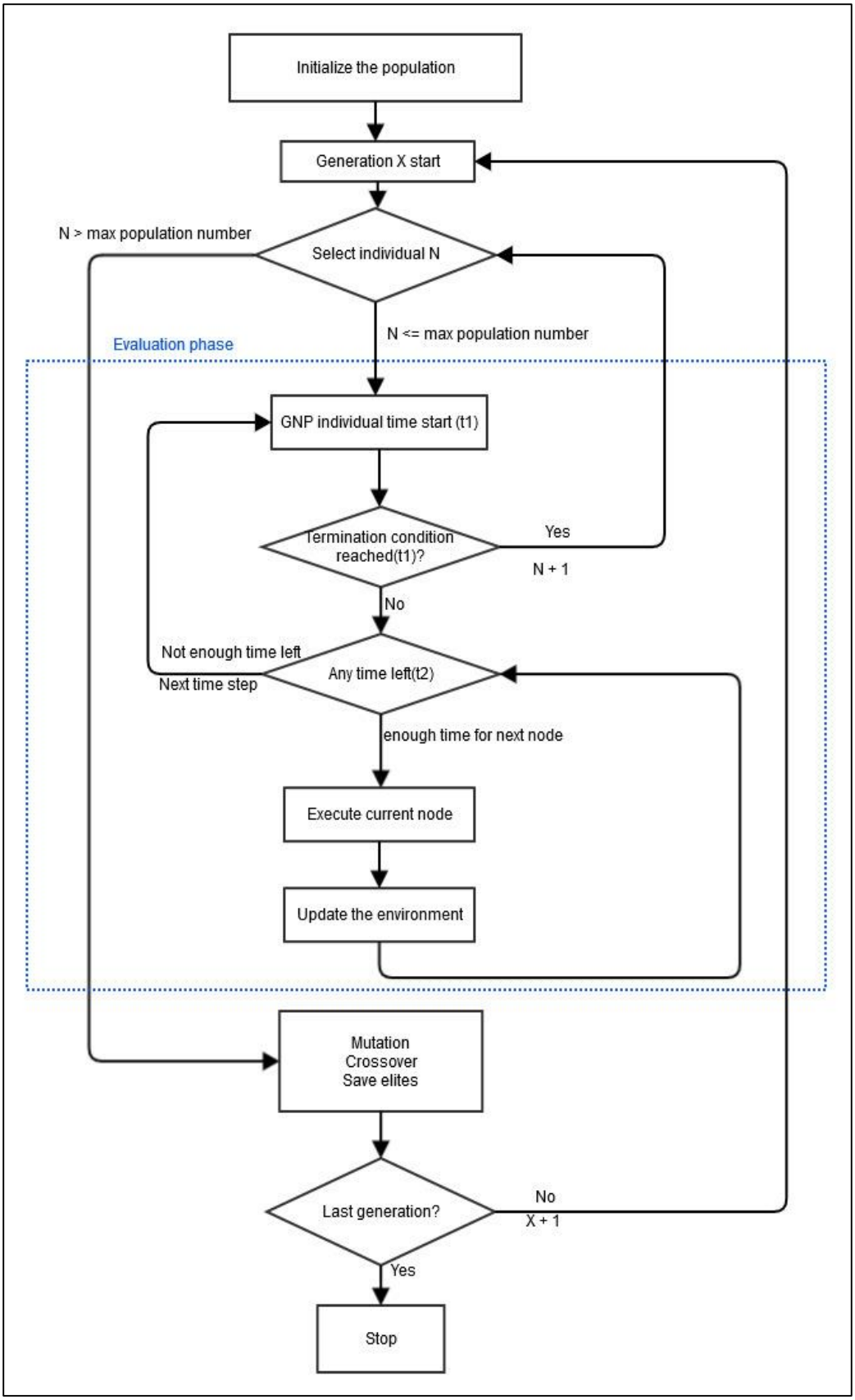


Figure 2.5 Schematic diagram of the genetic network programming algorithm (training phase)

The process can be divided into two phases, training phase and test phase. The Fig. 2.4 shows the schematic diagram of the GNP. It needs a population for evolution, and one graph is an individual in the population.

The population needs to be initialized at first, then starts the evolution. Each individual is evaluated in every generation, elites are kept to next generation and the rest individuals of next generation is generated by gene operation (Mutation and Crossover). At the end of training phase, good individuals can be selected for the test phase (Fig. 2.5).

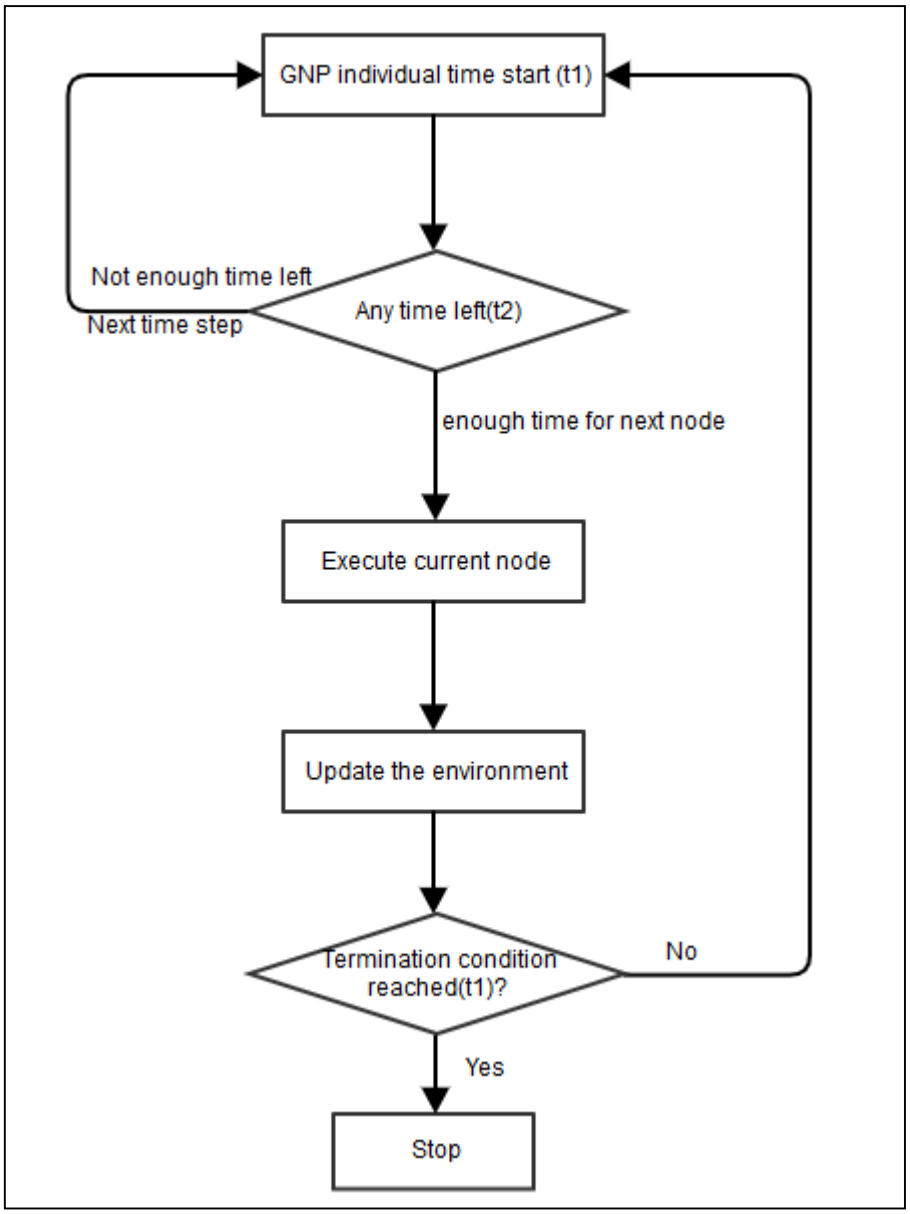


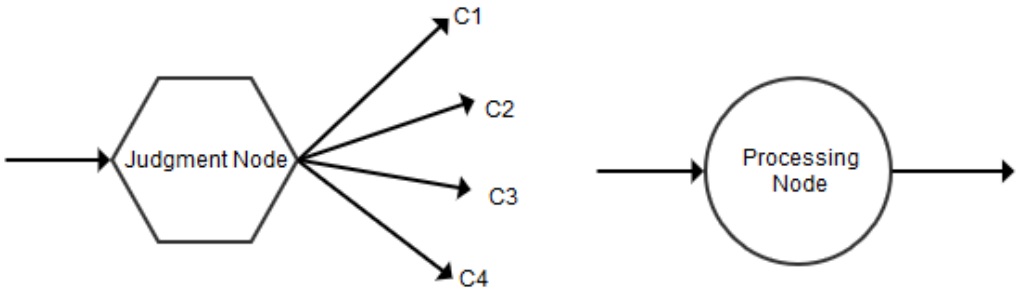
Figure 2.6 Schematic diagram of genetic network programming (testing phase)

2.3.2 Initialization the GNP

Determine the number of each type of node; therefore all individuals in a population have the same number of nodes and the nodes with the same ID have the same function. For each node, the connection is randomly assigned, but never directed to the start node.

2.3.3 Running a GNP Individual

When running a GNP individual, it begins from the start node, and the node transition is based on C_i. For judgment nodes, typically they have multiple connections to the next node (Fig. 2.6 a), for example, C_1, C_2, C_3 etc. The result of the judgment node determines which connection to select. For processing node, there is only one connection typically (Fig. 2.6 b).



(a) (b)

Figure 2.7 Judgment and Processing Node in GNP

Before executing each node, it needs to check the remaining time (t2) as there is a maximum time in one time step. If the remaining time is not enough to execute the next node, the program will wait for the next time step in the simulation or a real world system to continue.

2.3.4 Genetic Operators

In every generation, after running every individual, the program will select one or more good individuals by comparing their fitness. These elite individuals will be kept to the next generation. The rest of the population will mutate or crossover to produce the offspring, so the population keeps steady.

The Mutation process is as follows: (Mabu, et al., 2007)

1. Select one individual and reproduce it as a parent.
2. Each connection of each node is selected with a probability of Pm. The selected Ci will change to other value randomly (Never points to the start node).
3. Keep the mutated individual to the next generation.

The Crossover process is as follows: (Mabu, et al., 2007)

1. Select two individual and reproduce them as a parents.
2. Each node is selected as a crossover node with the probability of Pc.
3. Two parents exchange the genes of the corresponding crossover nodes, i.e., the nodes with the same node number.
4. Generated new individuals become the new ones of the next generation.

When processing the mutation and Crossover, the selection of individuals can use the tournament selection. It needs to define a tournament size N, randomly select N individuals and compare their fitness to choose the best individual.

2.4. GNP with Reinforcement Learning

Reinforcement learning changes the program every time step when running a task as an online learning algorithm. In order to obtain better results in dynamic environments, the GNP is extended by combining with reinforcement learning algorithm (Mabu, et al., 2007). In the GNP algorithm, the graph structure only changes during the evolution and once a good individual has been selected for testing phase the graph is fixed. On the other hand, the combination of GNP and reinforcement learning is able to change the connections of nodes in the testing phase, thereby improve its performance in dynamic environments.

2.4.1. Basic Structure of GNP-RL

In the GNP, each node only has one function, but in the GNP-RL each node may has several functions, and they can be considered as sub-nodes (Fig. 2.7). The major purpose of Reinforcement Learning is mapping a state to an action, so in this case the state is the ID of node and the action is the sub-node. In other words, the node only executes one sub-node a time and it is selected by the Reinforcement Algorithm. The state-action space is set as follows (Table 2.1).

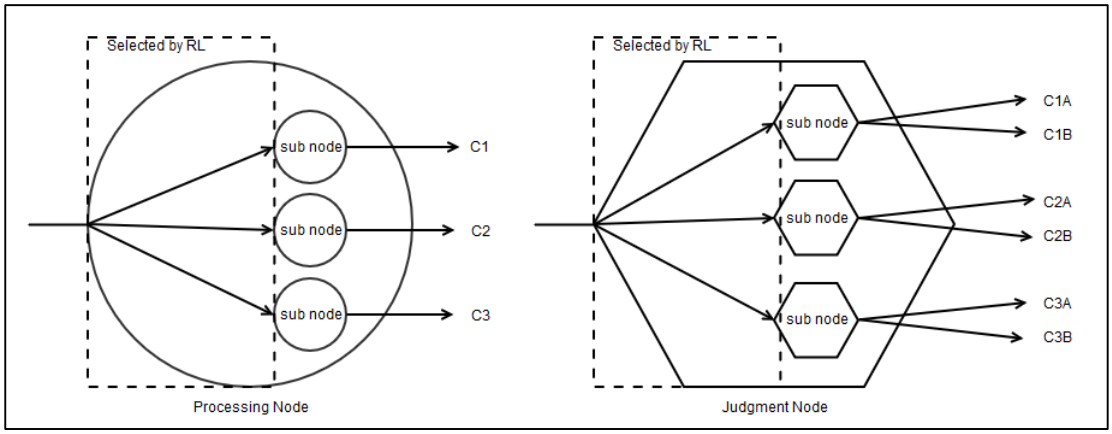


Figure 2.8 Processing node and judgment node with sub-nodes

Table 2.1 State-action space for RL in GNP-RL algorithm

	Sub-node 1	Sub-node 2	Sub-node 3	…	Sub-node M
0	Q-Value	Q-Value	Q-Value	…	Q-Value
1	Q-Value	Q-Value	Q-Value	…	Q-Value
2	Q-Value	Q-Value	Q-Value	…	Q-Value
3	Q-Value	Q-Value	Q-Value	…	Q-Value
…	…	…	…	…	…
N	Q-Value	Q-Value	Q-Value	…	Q-Value

*Q-Value corresponds to each state-action pair

For example, in original GNP algorithm, if there are four actions for the agent, they are considered as four different types of processing node in the network. On the other hand, in GNP-RL algorithm all four of these actions can be considered as sub-nodes of one processing node, and reinforcement learning algorithm is responsible for selecting which one to execute. By combining the reinforcement learning, it makes the structure even more compact, and the code for sub-nodes is also reusable.

An Extension of GNP-RL: Fuzzy Judgment Node (Mabu, et al., 2011)

Typically a judgment node is consists of some if-then statements, and each statement connects to a different node. However, the if-then statement is replaced by the fuzzification part of fuzzy logic system in fuzzy judgment node. Each membership in the fuzzy sets connects to a different node, and when executing the fuzzy judgment node, the degree of each membership function is used as a probability for selecting the connection.

2.4.2. Running a GNP-RL Individual

Basically running a GNP-RL individual proceeds similar to a pure GNP individual, as defined in Section 2.3.3. The only difference is that it needs to update the state-action space in each time step (Fig. 2.8). If the judgment node uses RL, it cannot receive an immediate reward from the environment after execution, because there is no action been taken. Typically the state-action space will be updated after a whole time step, while for processing node, it also can be updated immediately after the execution depends on the training situation. During the training and testing phase, the reinforcement learning always uses the ε-Greedy policy to select the sub-node. It keeps exploring and learning to change the selection of sub-nodes, thereby improve the performance in dynamic environment.

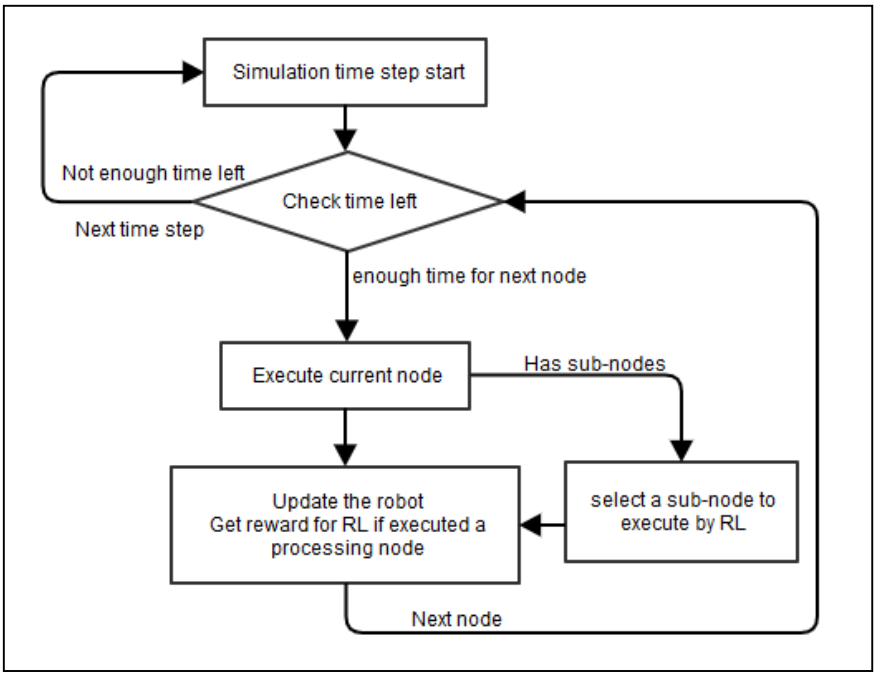


Figure 2.9 Schematic diagram of GNP- RL running in the testing phase

2.5. Summary

The fuzzy logic control system is the last intelligent system among all the algorithms introduced in this chapter. However, it is able to produce an excellent performance, if the fuzzy sets and fuzzy rules all set perfectly. It is necessary to implement fuzzy logic control system first as a benchmark for measuring the performance of other algorithm. Reinforcement learning is simple and efficient, but it is weak dealing with input which is continuous value. For robot control, the obvious choice is to use the combination of fuzzy logic and reinforcement learning algorithm. Genetic network programming can be seen as a substitute of reinforcement learning, they have similar capabilities, except it is not an online learning algorithm. The GNP-RL algorithm (with fuzzy judgment node) combines all the advantages of all these three basic algorithms, the research focuses on implement the GNP-RL algorithm for robot control, until the limitations are found from experiments. By comparing with all implementations of each algorithm, the final decision for achieving multi-objectives is to develop a new architecture of GNP: GNP with Fuzzy-RL nodes.

Chapter 3
Adaptations of the Algorithms for Robot Control: Single Behaviour

3.1. Problem Domain Specifications

This chapter focuses on testing two simple control objectives for the robot: steering angle and speed control for the ball pursuit behaviour. The robot needs to adjust its heading direction towards the ball and the robot speed is slowed down whenever the robot is close to the ball, and sped up whenever the robot is far from the ball.

These control objectives are relatively easy to achieve as they contribute towards achieving the same target pursuit behaviour.

Algorithm 1: Fuzzy logic implementing both steering angle and speed adjustment for ball pursuit
Algorithm 2: Reinforcement learning with fuzzy logic implementing the steering angle controller for ball pursuit.
Algorithm 3: Genetic network programming implementing both steering angle controller and speed adjustment for ball pursuit.

Simulation Environment
The simulation program is based on the FIRA Micro Robot World Cup Soccer Tournament. Since the robot and the ball are all moving on the same plane, it is simulated in a 2D environment.
Dimensions:

- Robot: 7.5 cm. x 7.5 cm.
- Ball: 3 cm. (diameter)
- Pitch: 220 cm. x 180 cm.

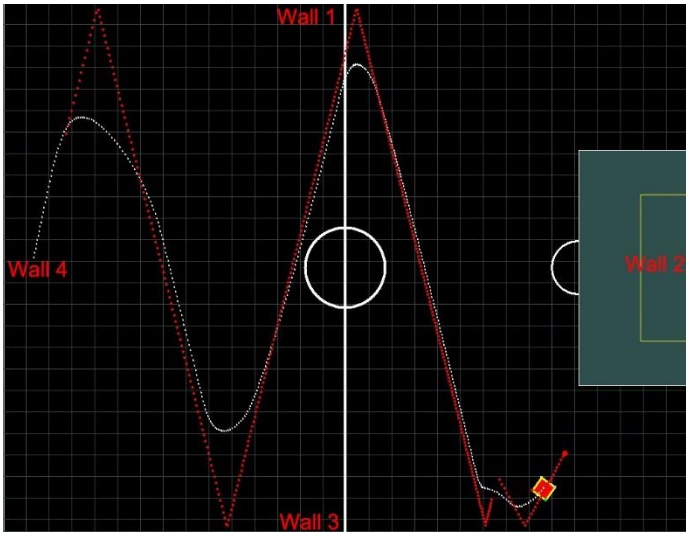


Figure 3.1 2D simulation environment

The simulation program updates the status of ball and robot every time step, and all the simulations in this thesis only considers the position, speed and direction. It is simple to calculate the distance and angle between robot and ball (or wall) by using these attributes, and all experiments in this thesis uses these values as the input of the control system. The control system is only allowed to adjust the speed and direction of the robot. The four boundaries of the pitch is named as Fig. 3.1 shows above. The red dots are the trace of the ball, and the white dots are the trace of robot.

3.2. Algorithm 1: Fuzzy Logic Controller

3.2.1. General Architecture

In this experiment, the fuzzy logic control system receives two input values, one is the heading angle to the ball from robot, and the other is the distance between robot and ball. The schematic diagram of the algorithm is shown in Fig. 3.2.

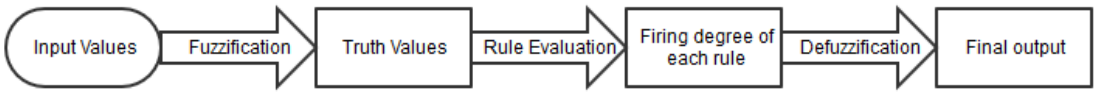


Figure 3.2 Flowchart of fuzzy logic control system

As we know, one fuzzy system only gives one final output value. In order to control the speed and the steering angle simultaneously, this experiment uses two fuzzy logic control systems, and both have the same input values.

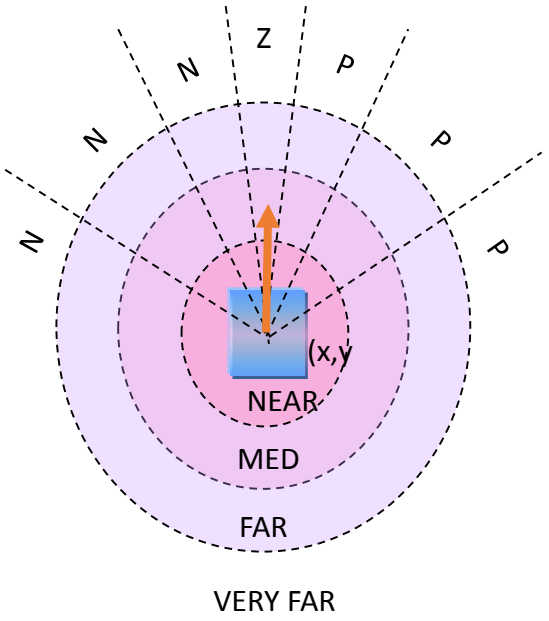


Figure 3.3 Fuzzy logic system design (NL-Negatively Large, NM-Negatively Medium, NS-Negatively Small, ZE-Zero, PS-Positively Small, PM-Positively Medium and PL-Positively Large)

As the Fig. 3.3 shows, the experiment uses a traditional fuzzy logic system design (Reyes, et al., 2013), which divides the angle into seven regions and the distance into four regions. So here are two fuzzy sets corresponding to two input values.

3.2.2. Problem-Specific Parameter Settings

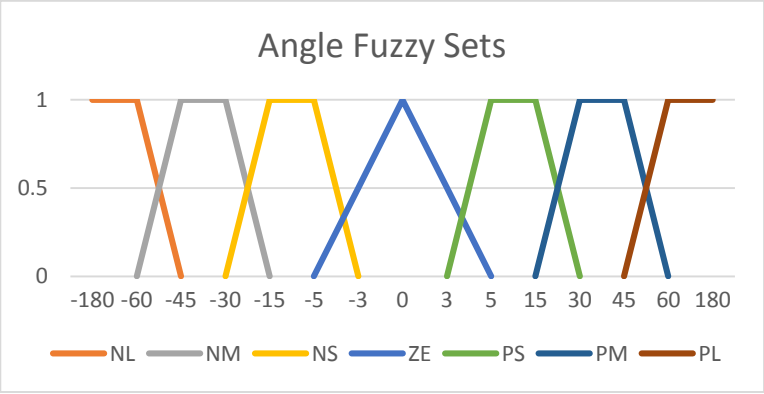


Figure 3.4 Angle fuzzy sets

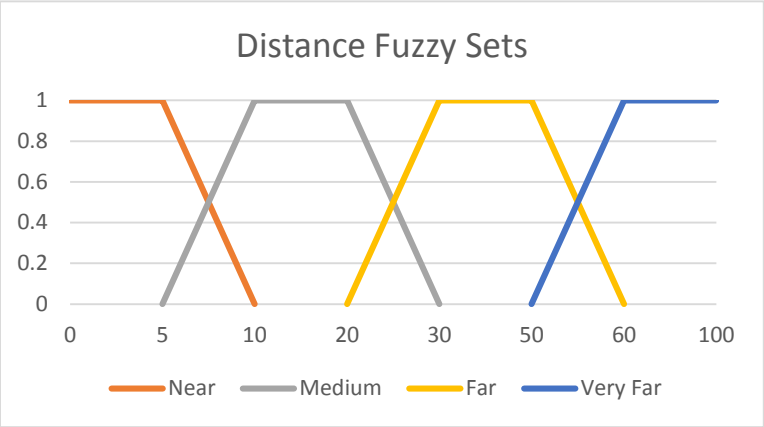


Figure 3.5 Distance Fuzzy Sets

Two fuzzy sets in this experiment are defined as Fig. 3.4 and Fig. 3.5 above. The rules for ball pursuit and speed control can be represented as fuzzy associative memory matrices, and they are shown in Table 3.1 and Table 3.2.

Table 3.1 Fuzzy Associative Memory Matrix for Ball Pursuit: Steering Angle Adjustment

	NEAR	MED	FAR	VERY FAR
NL	Very Sharp Left	Very Sharp Left	Sharp Left	Sharp Left
NM	Sharp Left	Sharp Left	Mild Left	Mild Left
NS	Mild Left	Mild Left	Very Mild Left	Very Mild Left
ZE	Zero	Zero	Zero	Zero
PS	Mild Right	Mild Right	Very Mild Right	Very Mild Right
PM	Sharp Right	Sharp Right	Mild Right	Mild Right
PL	Very Sharp Right	Very Sharp Right	Sharp Right	Sharp Right

Table 3.2 Fuzzy Associative Memory Matrix for Ball Pursuit: Speed Control

	NEAR	MED	FAR	VERY FAR
NL	Very Slow	Very Slow	Slow	Slow
NM	Very Slow	Slow	Medium	Medium
NS	Very Slow	Medium	Fast	Very Fast
ZE	Slow	Medium	Fast	Wicked Fast
PS	Very Slow	Medium	Fast	Very Fast
PM	Very Slow	Slow	Medium	Medium
PL	Very Slow	Very Slow	Slow	Slow

In general these rules are set to achieve two principles.

1. Turn the robot sharply if the angle from the ball is large and the distance is small.
2. Speed down if the angle from the ball is large and the distance is small.

3.2.3. Experiment Results and Analysis

The simulation program records the position of the ball and robot every time step. The Fig. 3.6 is a screen shot of the experiment.

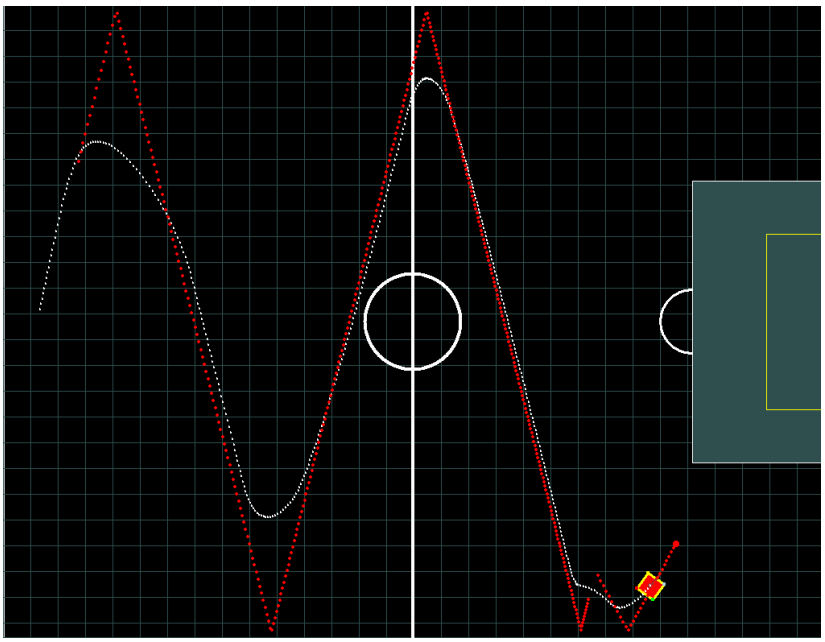


Figure 3.6 Trace of ball and robot (fuzzy logic controller)

The robot moved very smooth in the experiment and achieved ball pursuit and speed control at the same time. The crucial part in fuzzy logic control system is determining

the appropriate fuzzy sets and rules. These settings directly affect the system's performance, and there is no simple method to set these parameters correctly and efficiently. It is a time consuming job to try the different parameters and get a good performance.

3.2.4. Limitations of the Algorithm

There is no doubt that fuzzy logic control is able to present an excellent control system, but the major problem is that it needs an expert's prior knowledge to design and create the fuzzy sets and rules. It's a time consuming job for people to test and improve the performance of a fuzzy logic system. And when dealing with lots of input values, even a very experienced person cannot set proper rules for the system.

3.3. Algorithm 2: Reinforcement Learning with Fuzzy Logic

In general, reinforcement learning maps states into actions, and the number of state and action are finite. However, for the navigation problem domain, the state is the input value and it is a real number, reinforcement learning needs the help of an intermediary algorithm, in this case. So the basic idea of this experiment is to fuzzify the input value to obtain limited states for the reinforcement learning, and it takes responsibility to construct the fuzzy rules. As compared to a pure fuzzy logic control system, this algorithm does not need any prior expert knowledge to create the fuzzy rules; thus, saving a lot of time.

This experiment only tested the steering angle controller for ball pursuit behaviour, as speed control can be simply achieved by adding another reinforcement learning instance. The input value is the angle to the ball and the only output is the steering angle for the robot. The speed of the robot is set to a constant initially. The SARSA(λ) learning algorithm (Sutton, et al., 2012) was used for this experiment.

3.3.1 General Architecture

To combine fuzzy logic and reinforcement learning together, this experiment tried two different architectures for implementation. The first one only combines a fuzzification component with reinforcement learning, and is referred to as RL with fuzzified input. The second approach combines a complete fuzzy logic system with reinforcement learning, and is referred to as Fuzzy-RL.

Algorithm 2a: Reinforcement Learning with a Fuzzified Input

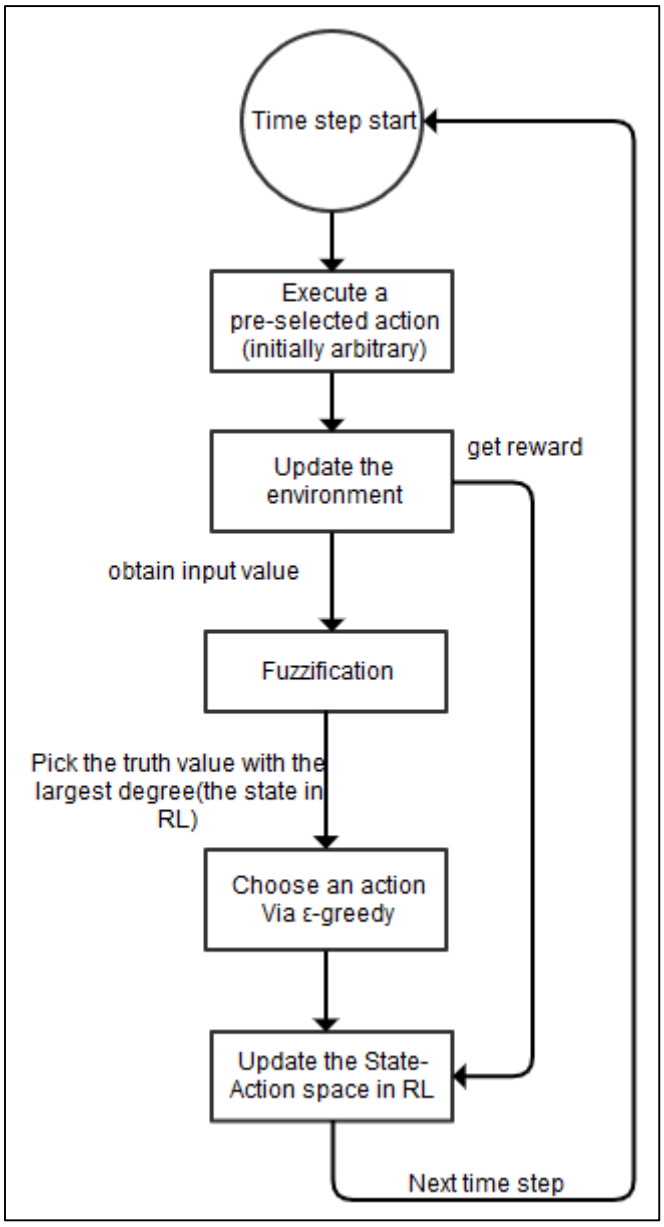


Figure 3.7 Schematic diagram of RL with fuzzified input algorithm

Algorithm 2b: Reinforcement Learning with Fuzzy Logic System (Fuzzy-RL)

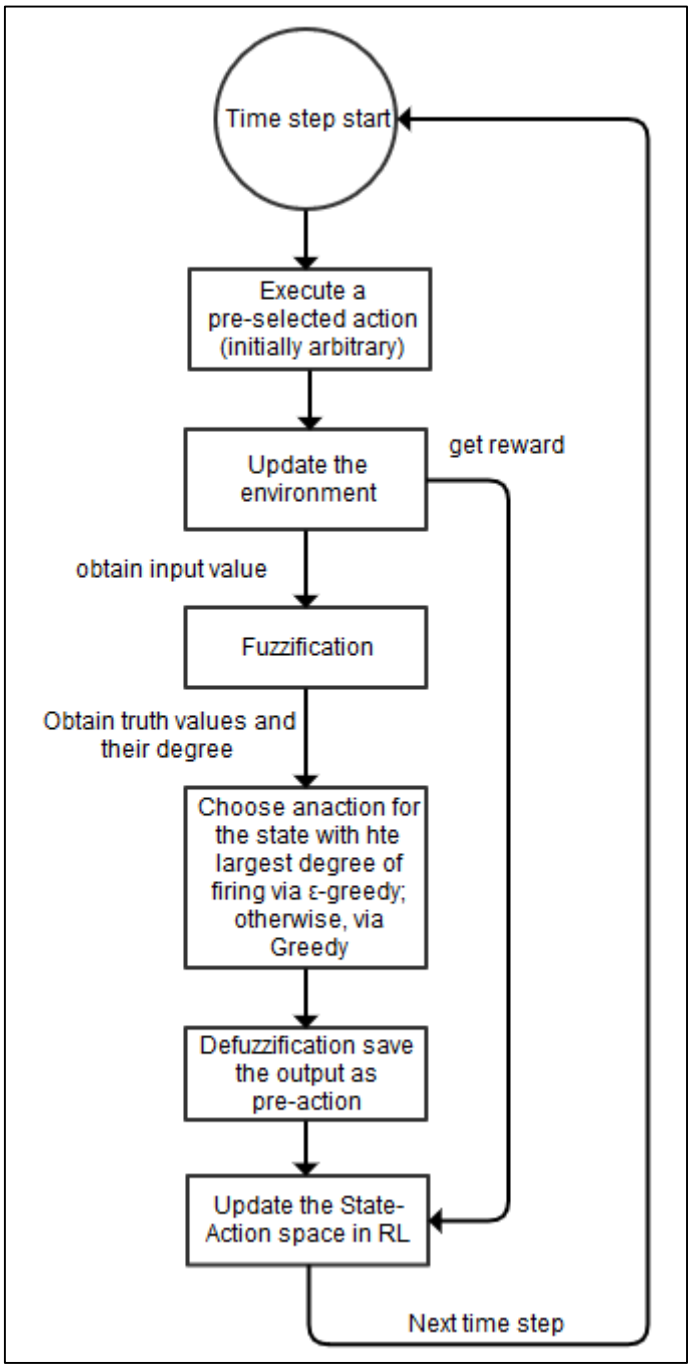


Figure 3.8 Schematic diagram of the Fuzzy-RL algorithm

The two figures (Fig. 3.7, Fig. 3.8) above show the schematic diagram of RL with fuzzified input and Fuzzy-RL algorithm. The system feeds only on one input; that is, the angle from the ball. The SARSA(λ) learning algorithm (Sutton, et al., 2012) is employed in both approaches.

After the fuzzification, as we know there maybe more than one truth value gets a

nonzero degree. In Algorithm 2a, only the truth value with the largest degree of firing is used as the state in RL. In turn, the reinforcement learning algorithm chooses only one action corresponding to this state.

On the other hand, in Algorithm 2b, the Fuzzy-RL algorithm considers all truth values with nonzero degree of firing, then, picks the truth value with the largest degree as the main state for updating and selecting the main action. This is because the reinforcement learning algorithm only reinforces one action-state pair, in one time step. The minor actions are chosen via the greedy policy, instead to limit the exploration phase and make use of knowledge exploitation more. After defuzzification phase, the system calculates the final output, via the centre of mass formula.

3.3.2 Problem-Specific Parameter Settings

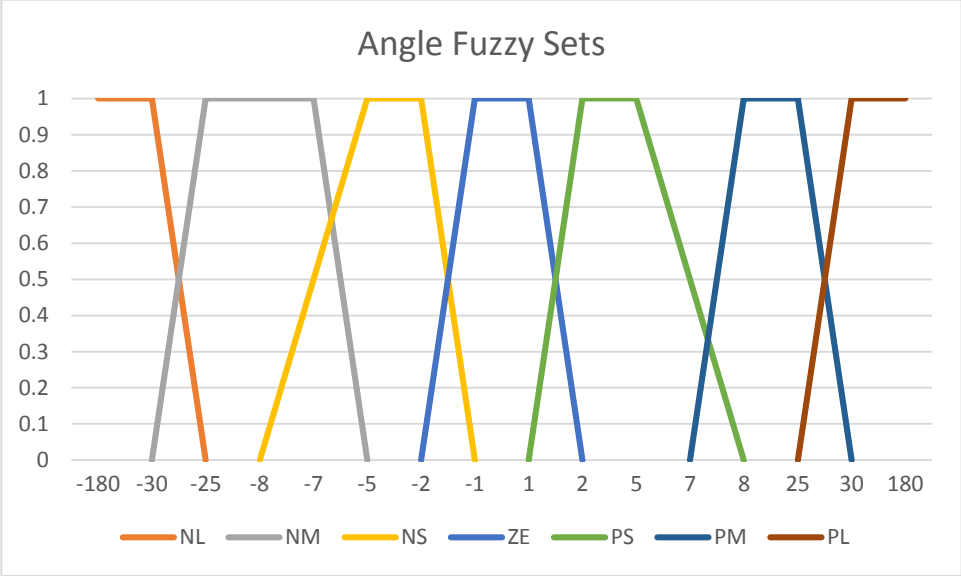


Figure 3.9 Angle Fuzzy Sets

The two algorithms use the same fuzzy sets for the input value (angle from the ball) as Fig. 3.9 shows. This experiment uses seven actions as Table 3.3 shows, the positive degree means turn left and the negative degree means turn right. Once the robot exhibits the appropriate behaviour, it is deemed that the Q-values in the state-action space already converged. At this stage, the reinforcement learning algorithm has completed the construction of the fuzzy rules.

Table 3.3 State-Action space (States are the truth value from Fuzzy system, Actions are steering angles for the robot)

State\Action	0 deg.	1 deg.	-1 deg.	5 deg.	-5 deg.	25 deg.	-25 deg.
NL	6.13061	3.73286	5.5691	7.16532	38.6804	4.48876	1.10006
NM	7.37921	1.95259	7.02415	7.27928	27.4277	3.42272	3.50469
NS	2.82158	6.02192	5.59275	2.19006	3.62111	25.8563	2.38168
ZE	13.9584	11.0166	21.4455	8.45386	14.0966	14.5892	7.41781
PS	17.9597	23.5371	18.2677	22.2188	25.229	14.4371	15.3839
PM	17.5066	16.1582	12.5777	29.9405	24.7555	8.59868	17.2286
PL	12.8531	3.60552	13.6882	14.0344	17.576	44.8491	13.6786

*The value in cells are hypothetical Q-Value, corresponds to each state-action pair

The reward function is the most important part in this experiment. Generally, the system needs to generate a reward when the action accomplishes the goal. In this case, the angle between the robot and the ball is within the range [-1, 1] degrees. The RL is able to reinforce previous actions executed, and by doing this repeatedly, it learns the optimal policy to achieve the goal (i.e. target robot behaviour). However, in the simulation environment, the robot may never get to the goal, but just constantly steers randomly in a small area. The solution to this problem is to evaluate every action and offer a reward that distinguishes a relatively bad action from a relatively good action.

Reward Function

Pseudo code 7: Reward function for RL with FLS

1. **If** newangle is less than oldangle
2. **then** reward = 30 * (1 - newangle / oldangle)
3. An extra reward is given when the robot arrives at the goal.

The parameters in SARSA(λ) learning is set as:

Explore rate = 0.1, λ= 0.5, α= 0.1 γ= 0.7

3.3.3 Results and Analysis

The robot is placed initially at the centre of the field, facing wall 1, with a constant speed of 0.6 units per time step. The ball is set initially to be on the left, 85 units away from the robot with a speed of 0. As reinforcement learning is an online learning algorithm, once the simulation starts, the robot will gradually learn the behaviour of ball pursuit.

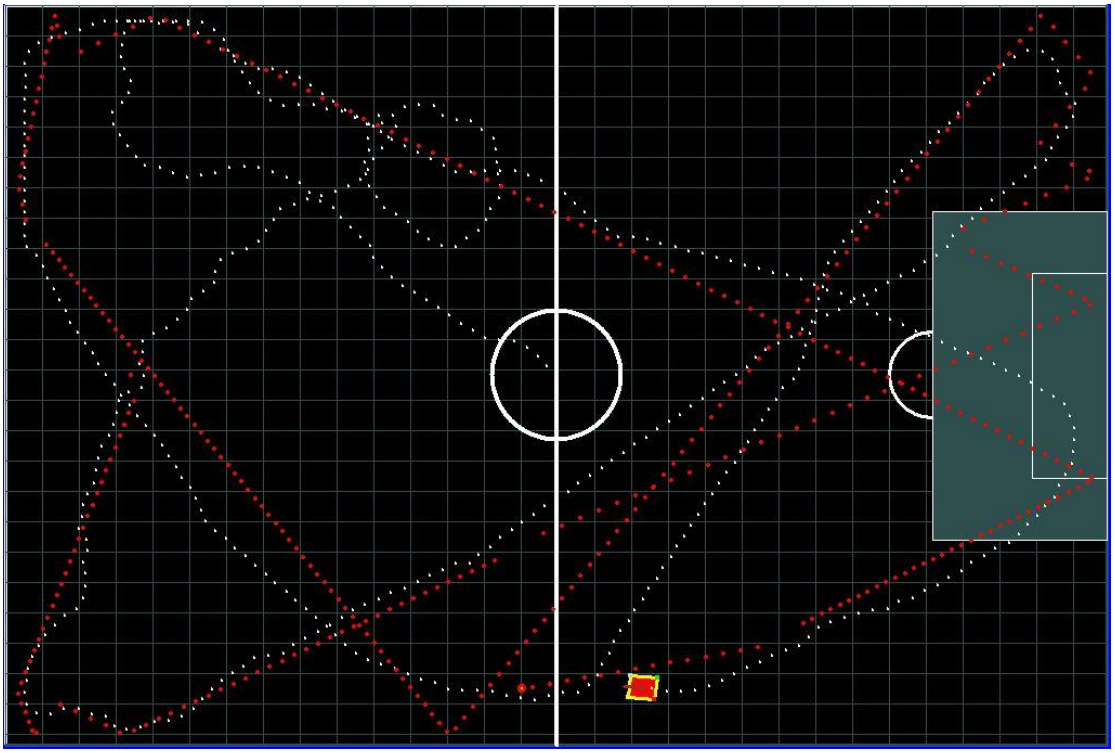


Figure 3.10 Initial part of RL with fuzzified input algorithm

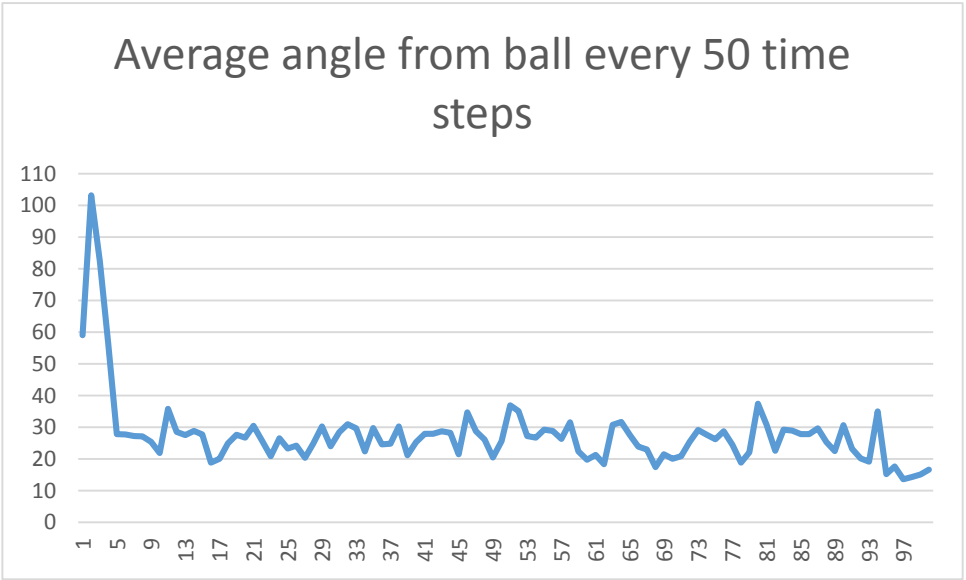


Figure 3.11 Results of Algorithm 2a: Average angle from ball every 50 time steps (y-axis = ave. angle; x-axis: 1 unit = 50 time steps).

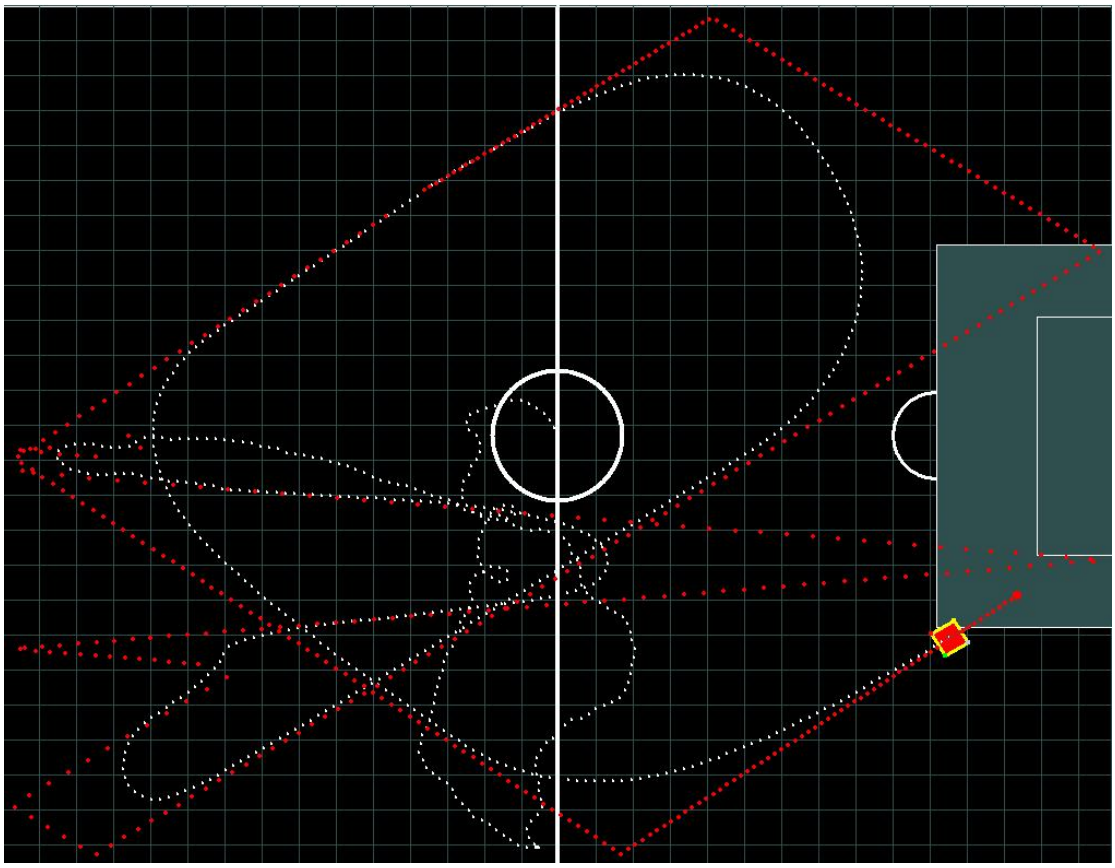


Figure 3.12 Results of Algorithm 2b: Learning phase of the Fuzzy-RL algorithm.

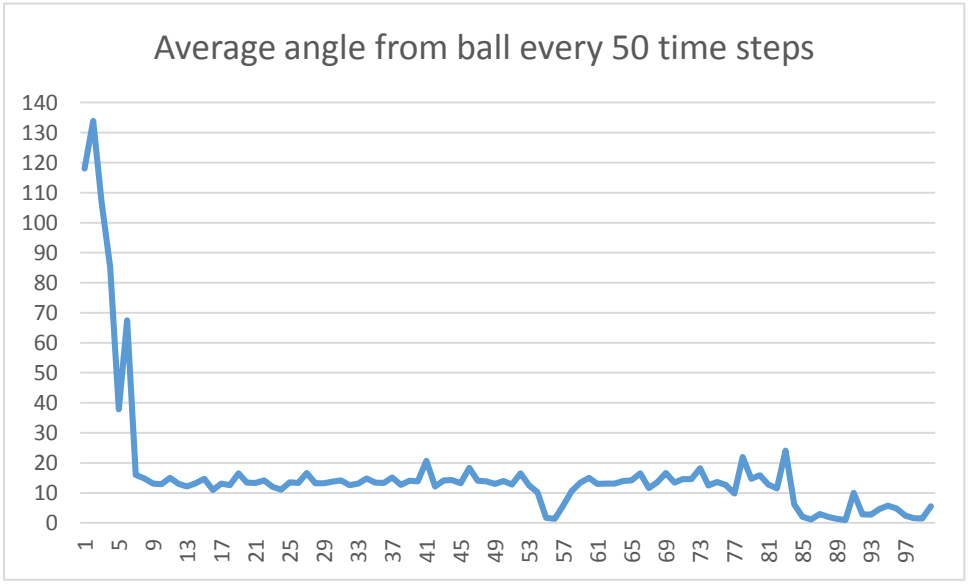


Figure 3.13 Results of Algorithm 2b: Average angle from ball every 50 time steps (y-axis = ave. angle; x-axis: 1 unit = 50 time steps).

As Fig. 3.10 and Fig. 3.12 show, the robot just moves randomly at first, and then learns how to pursue the ball. The average angle from ball decreases dramatically in the first 10 units (500 time steps) time steps, and then keeps steady (Fig. 3.11, Fig. 3.13). There is a slight drop in Fig. 3.13 at the 85th unit (4250 time steps) and then the value fluctuates at around 5 degree. In contrast, the angle in Fig. 3.11 always fluctuates at around 15 degrees after 10 units (500 time steps).

The figures in 3.14 and 3.15 below show the path traced after running the simulation. The speed of the ball is initialized faster than the speed of the robot, and it is slowing down because of the friction in the simulation environment. The Q-values in the state-action space already converged in both algorithms. Comparing the average angle from ball (Fuzzy-RL: around 5 degrees; RL with fuzzified input: around 15 degrees) and the observable trace of robot, we can clearly see that Fuzzy-RL algorithm is much better than RL with only the fuzzified input algorithm. The robot using Fuzzy-RL algorithm moves very smoothly, just like using a hand-calibrated pure fuzzy logic control system.

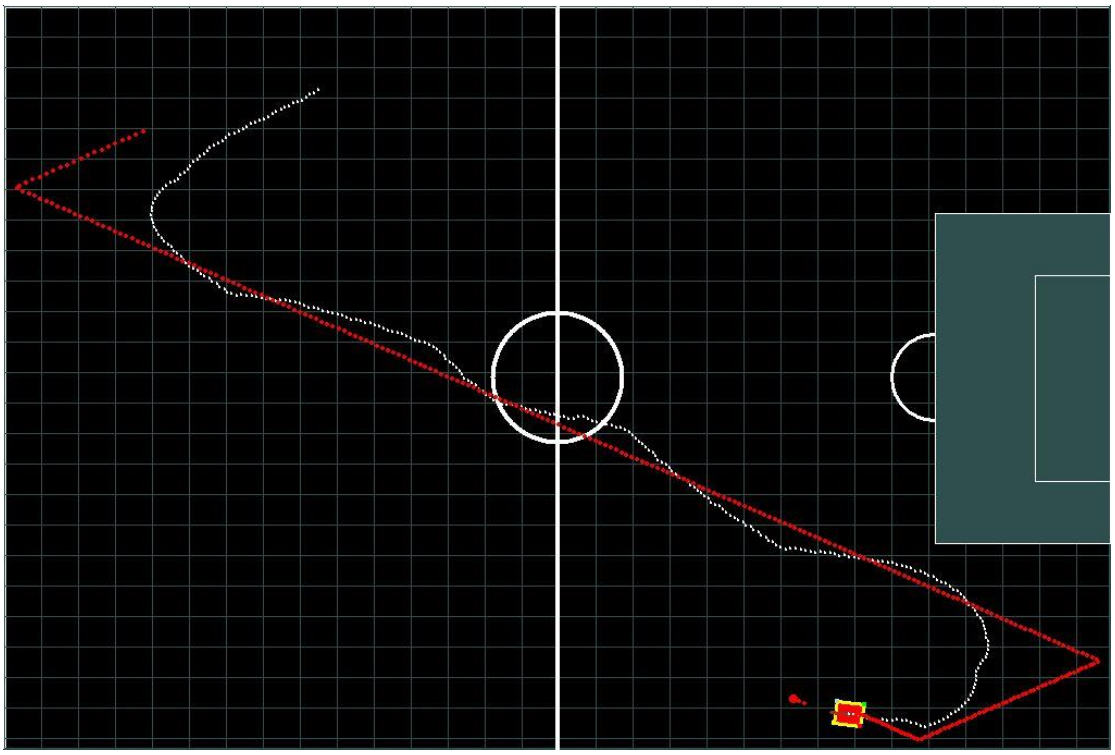


Figure 3.14 The performance after running for a while (RL with fuzzified input algorithm)

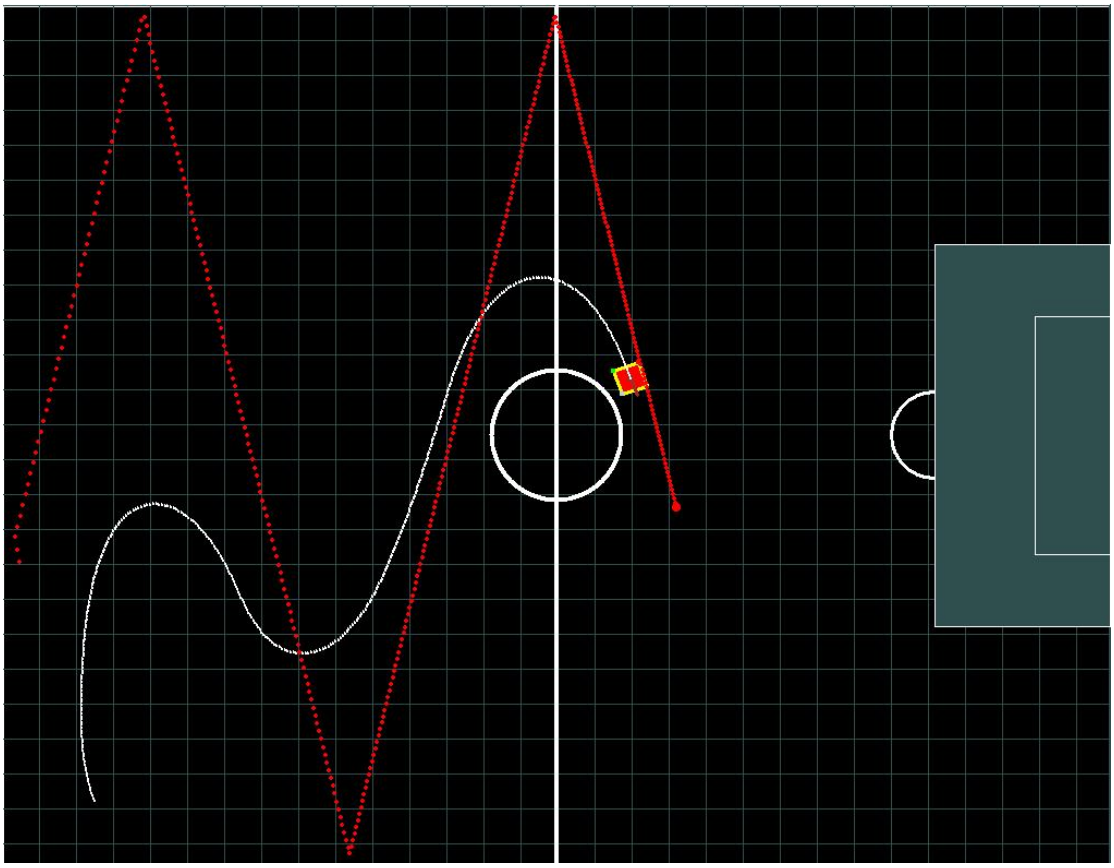


Figure 3.15 The performance after running for a while (Fuzzy-RL algorithm)

3.3.4 Limitations of the Algorithm

As we can see from the experiments, the RL with fuzzified input algorithm doesn't perform well enough. On the other hand, the Fuzzy-RL algorithm gives a wonderful performance just like the hand-calibrated fuzzy logic control system, and all the rules were generated automatically. The crucial part in this algorithm is the reward function, a well-designed reward function can accelerate the convergence of the Q-values in the state-action space, and produce an excellent result. Otherwise, it may not be very efficient. The other limitation of Fuzzy-RL will be introduced in chapter 4.

3.4. Algorithm 3: Genetic Network Programming with Reinforcement Learning

3.4.1. General Architecture

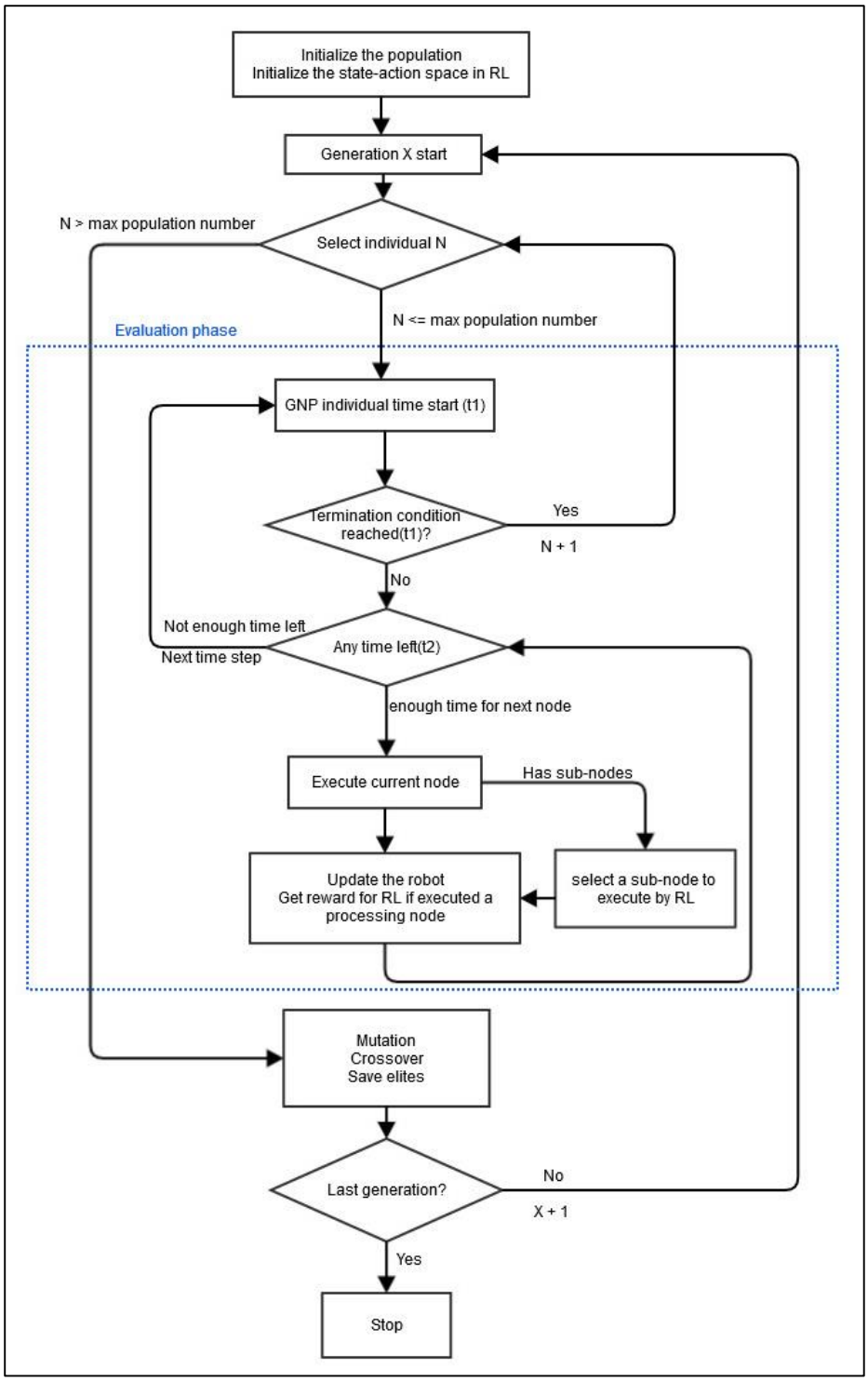


Figure 3.16 Schematic diagram of GNP with RL for training phase

The Fig. 3.16 shows the schematic diagram of GNP with RL for training phase. When training an individual in the environment, it needs to set some conditions for calling a failure, in order not to waste too much time (t1) on a very bad individual. For ball pursuit, if the robot is far away from the ball, or uses too much time steps (t1), it considers it a failure, then jumps to the next individual. Once the next individual enters the simulation environment, the environment is reset, so that every individual is trained using the same environmental conditions.

Each GNP individual uses one reinforcement learning algorithm instance to select a (function) sub-node in a processing node. The state here is the ID of each node, and the actions are the functions (sub-nodes) in each node. The State-Action space is shown as Table 3.4.

Table 3.4 The state-action space of RL

State\Function	0	1	2	3	4	……	M=number of functions
0	Q-Value	Q-Value	Q-Value	Q-Value	Q-Value	……	Q-Value
1	Q-Value	Q-Value	Q-Value	Q-Value	Q-Value	……	Q-Value
2	Q-Value	Q-Value	Q-Value	Q-Value	Q-Value	……	Q-Value
……	……	……	……	……	……	……	……
N = number of nodes	Q-Value	Q-Value	Q-Value	Q-Value	Q-Value	……	Q-Value

*Q-Value corresponds to each state-action pair

Fitness is calculated for each GNP individual after it is trained in the simulation environment. The individual with the highest fitness is considered as the most elite and will be kept to the next generation. Once it reaches the last generation, the top five individuals are selected for the testing phase.

The Fig. 3.17 shows the schematic diagram of GNP with RL for testing phase, in this stage the performance of the best individual is displayed by the graph engine, so it can be manually judged whether it is good or not.

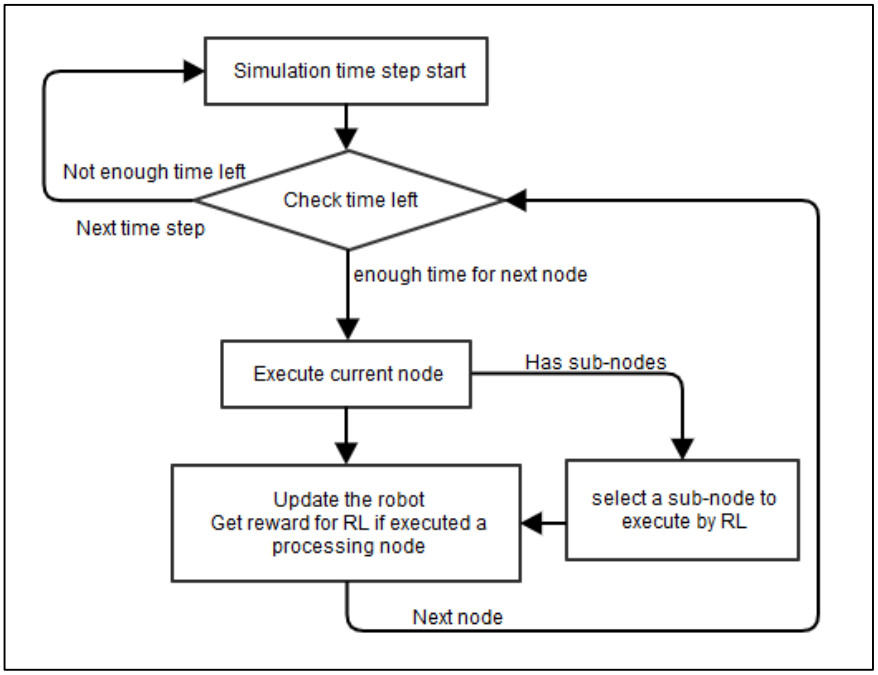


Figure 3.17 Schematic diagram of GNP with RL for testing phase

3.4.2. Problem-Specific Parameter Settings

Judgment Nodes

There are two different types of judgment node, one is for judging the angle from ball, and one is for judging the distance from ball. The judgment node settings are shown in the Fig. 3.18.

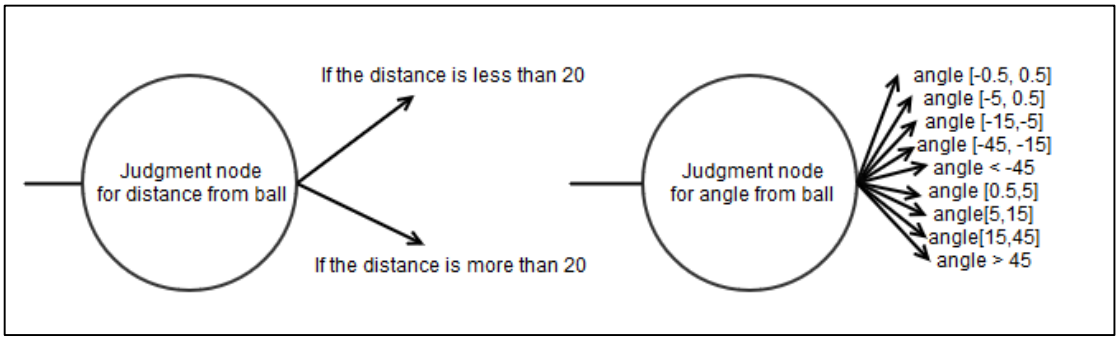


Figure 3.18 Judgment node settings

Processing Nodes

Four different types of processing nodes are defined in this implementation, one is to turn left, one is to turn right, one is for high speed and one is for low speed, they are shown as Fig. 3.19.

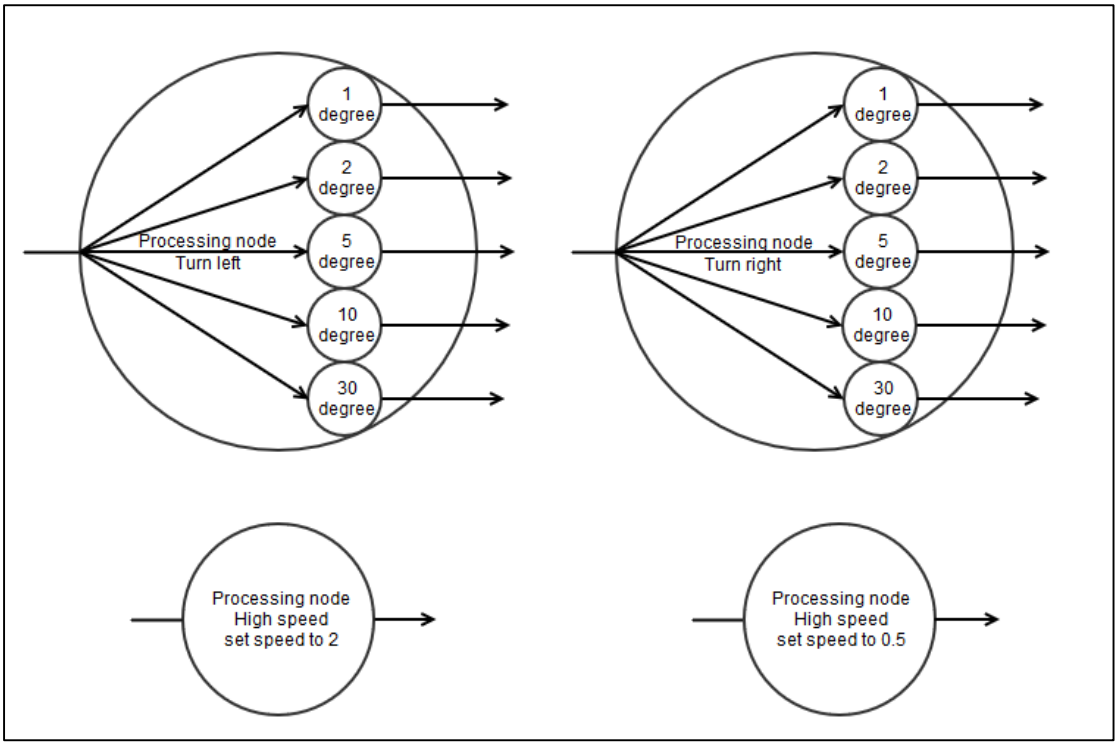


Figure 3.19 Processing node settings

The time delay of each judgment node is set to 1, the time delay for each processing node is set to 3, and the max time in a simulation time step is set to 8. The probability of mutation and crossover is not very sensitive to the performance as only the connection changes in this implementation. They are all set to a small value; less than 0.5 is fine. The other parameters are set as follows:

- Population: 100
- Mutation: 63
- Crossover: 32
- Elites: 5
- Tournament size: 6

Reward Function

Pseudo code 8: Reward function for GNP-RL

1. **If** newangle is less than oldangle
2. **then** reward = 30 * (1 - newangle / oldangle)
3. An extra reward is given when the robot arrives at the goal.

There are three conditions to terminate one individual:

1. if the time (t1) steps is over 200;
2. if the robot moves too far away from the ball, and
3. if the robot reaches the ball successfully.

The fitness function is a bit complex, it can be generally described as follows:

1. In each time (t1) step, if the robot gets the correct speed, then the fitness increases.
2. At the end of the simulation, the robot that gets closer to the ball gets the higher fitness.
3. If the robot reaches the ball successfully, the lesser time steps it took, gives it a higher fitness.

3.4.3. Results and Analysis

In the training phase, the ball and the robot is set to two fixed positions when an individual begins its training simulation. Figure 3.20 shows the fitness of the best individual in the training phase, the x-axis is the generation count.

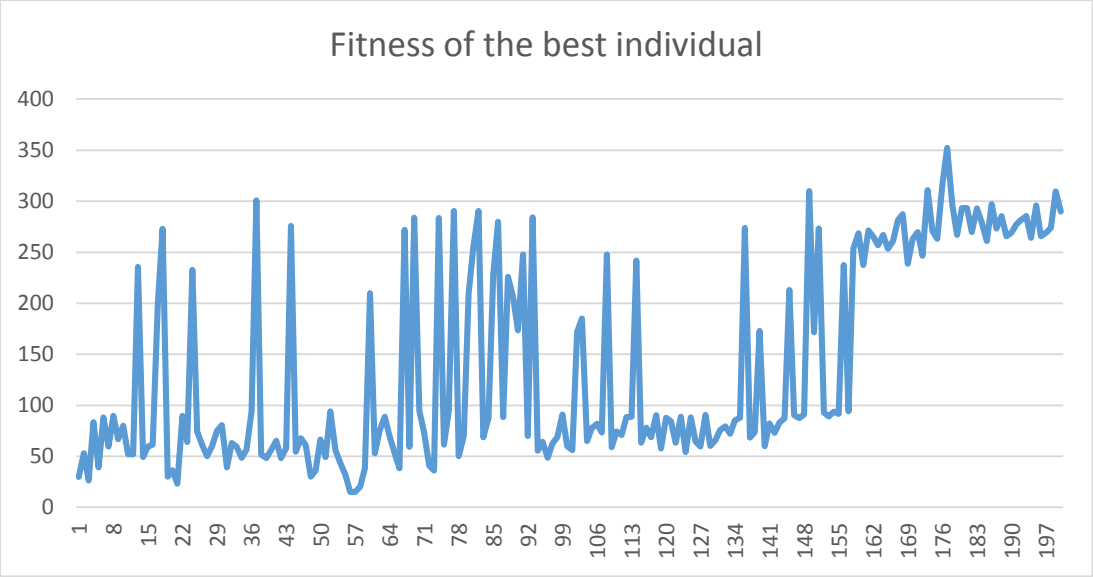


Figure 3.20 Fitness of the best individual (y-axis = fitness; x-axis = generation count)

The fitness fluctuates dramatically before 155 generations, and mainly keeps a low value at around 100. After 155 generations, the fitness goes up relatively steady at around 300. This kind of figure is very different from genetic algorithm or even pure genetic network programming. All these fluctuations are caused by the reinforcement learning component, because it is also exploring in the training phase, and the performance is not steady yet. So in this algorithm, the fitness is not a

strict indicator of the actual performance of an individual in the real-world, but merely an estimate. This is why I keep five elite individuals at the end of training. The highest fitness does not accurately correspond to the best individual when testing them manually. Nevertheless, the fitness value is still able to indicate when should the training stop. It is clearly shown in the figure that the fitness stabilizes after 155 generations, this is the signal to stop the training phase.

Table 3.5 Performance data of top five individuals

	Fitness	Average angle from ball*	Percentage of time with the desired speed*
1	282.189	80.0269	0.1%
2	275.659	49.7347	19.6%
3	271.19	29.8886	38%
4	267.41	38.8341	31.7%
5	260.921	19.4059	46.7%

*Calculated by running the testing phase for 2000 time steps

The first column in Table 3.5 shows the fitness value obtained during the training phase, the second column shows the average angle from ball - calculated in the testing phase (set X is the total angle from ball in 2000 time steps, Average angle from ball = X/2000), and the last column is the percentage of time of the robot with the desired speed in the testing phase (set an accumulator Y, in each time step if the robot gets the target speed corresponding to the distance from ball, Y plus 1. The percentage of time = Y/2000). The highest fitness does not correspond to the best performance though. The best had to be selected manually from the top five.

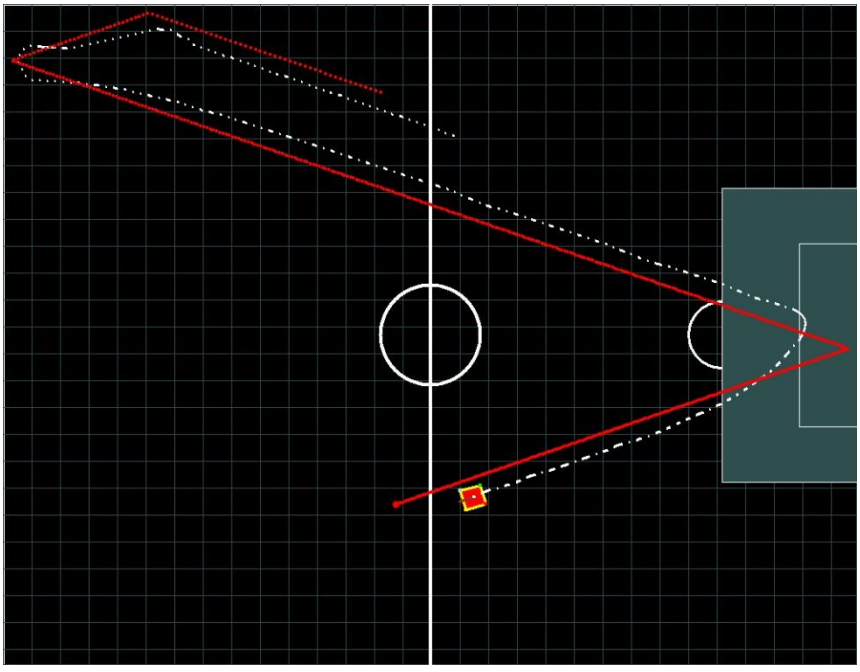


Figure 3.21 Performance of the GNP with RL

Figure 3.21 depicts a well-trained individual running under testing phase. At first, the robot was far away from the ball, then it moved relatively faster towards the ball. As soon as it is close enough to the ball, it slowed down to catch it. This is the reason that the trace of the robot is like a dashed line towards the end. The speed control perfectly meets the expectations. The performance on ball pursuit is also good, as it achieved smooth turns when the ball changed its direction.

3.4.4. Limitations of the Algorithm

In general, this algorithm accomplishes the expected objectives: speed control and steering angle control for ball pursuit, simultaneously. However, there are some limitations that we can be observed from the experiments. The most obvious problem is that the fitness value always fluctuates. It cannot reflect the quality of an individual accurately, so the best individual still needs to be picked manually from the five top individuals at the end of the training. The reinforcement learning part is exploring during the training phase, and therefore can cause some confusion. It is hard to tell whether there is enough time (t1) steps allotted for learning in one generation, for one individual. The fitness relies on the performance of the reinforcement learning, as well. In this implementation it still works fine, but the fitness value can be expected to fluctuate even more, if we are to add more sub-nodes (functions) into the processing nodes.

3.5. Summary

From the results of all algorithms that tested in this chapter, there is no doubt that fuzzy logic control algorithm and Fuzzy-RL algorithm (Algorithm 2b) archives the best performance. However, fuzzy logic control algorithm is not continued for multi-behaviour robot test, because it requires too many manual settings which is not original intention of this research. Fuzzy-RL becomes the best candidate for testing multi-behaviour robot control, and the details of implementation and experiments are discussed in chapter 4. GNP with RL algorithm is also not continued due to the limitations mentioned before. Nevertheless, the flexible structure of GNP algorithm promised there are still lots of possibilities that can to be explored. The novel architecture introduced in chapter 5 is developed based on all the results of experiments in chapter 3 and 4.

Chapter 4
Fuzzy-Reinforcement Learning for Robot Multi-behaviour

4.1 Introduction

This chapter focuses on combining a new behaviour, wall avoidance with ball pursuit and speed control. The general idea behind speed control for the ball pursuit behaviour is to slow down whenever the robot is close to the ball, and to speed up whenever it is away from the ball. Two different algorithms were tested in this chapter, namely: Fuzzy-Reinforcement learning and Genetic network programming with trained Fuzzy-RL nodes. The Fuzzy-Reinforcement learning algorithm has been tested earlier for ball pursuit in chapter 3. However, it is modified in this chapter to will deal with more input values (such as the distance from the wall and the angle from the wall, as shown as Fig. 4.1) to form a more complex behaviour.

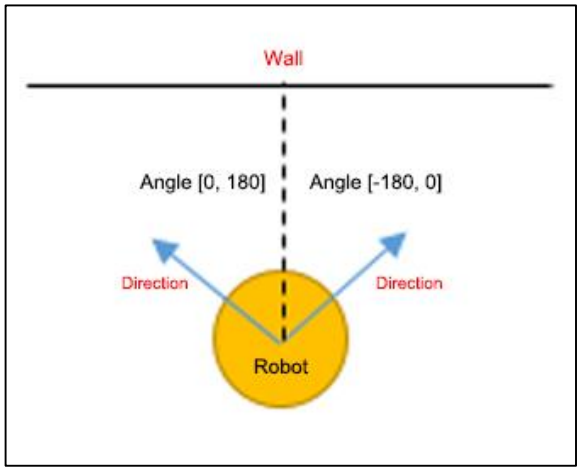


Figure 4.1 Calculation of difference between the heading angle of the ball, and the nearest wall

The Fuzzy-RL algorithm only considers the nearest wall to avoid. As introduced in chapter 3, Figure 3.1, there are four walls. The GNP with Fuzzy-RL nodes algorithm is designed to distinguish four walls and take different actions to avoid the different walls. The simulation environment is the same as in chapter 3.

4.2 General Architecture

The general architecture is the same as in chapter 3, but with a few modifications concerning the reward function and the state-action space and the inputs to the system. The schematic diagram of the Fuzzy-RL algorithm is shown in Fig. 4.2.

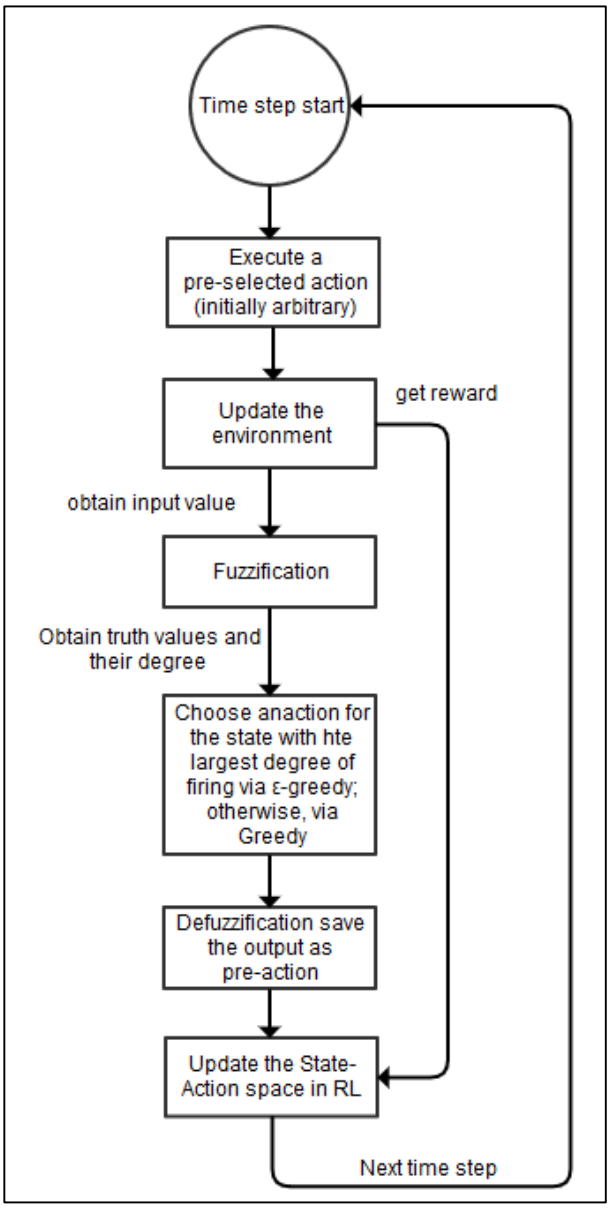


Figure 4.2 Schematic diagram of the Fuzzy-RL algorithm

4.3 Problem-Specific Parameter Settings

4.3.1 Fuzzy Logic Parameters

Three input values are used in this implementation: distance from the nearest wall, angle from the nearest wall, angle from the ball. The fuzzy sets for these input values are shown below (Fig. 4.3, Fig. 4.4 and Fig. 4.5). The negative values correspond to wall or ball positions that are located to the left of the robot's heading angle. These were hand-calibrated, and use exactly the same parameters as those derived for Algorithm 2a & 2b.

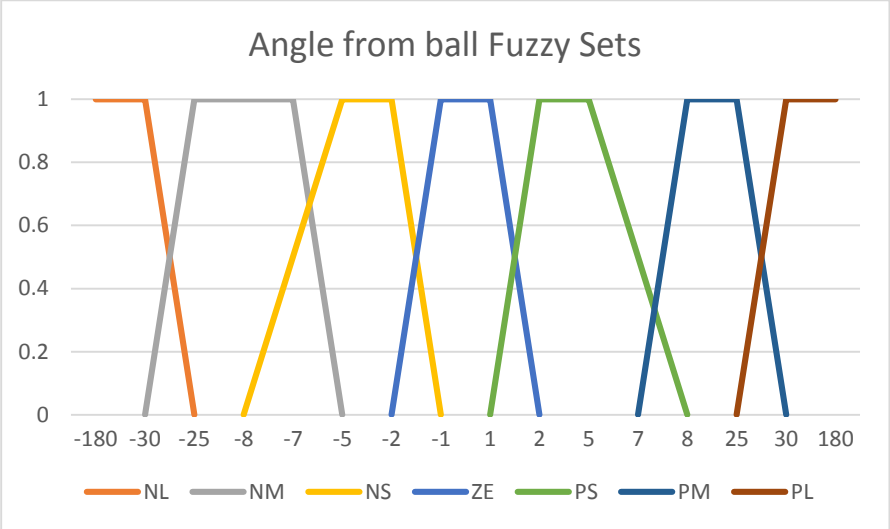


Figure 4.3 Angle from ball Fuzzy Sets

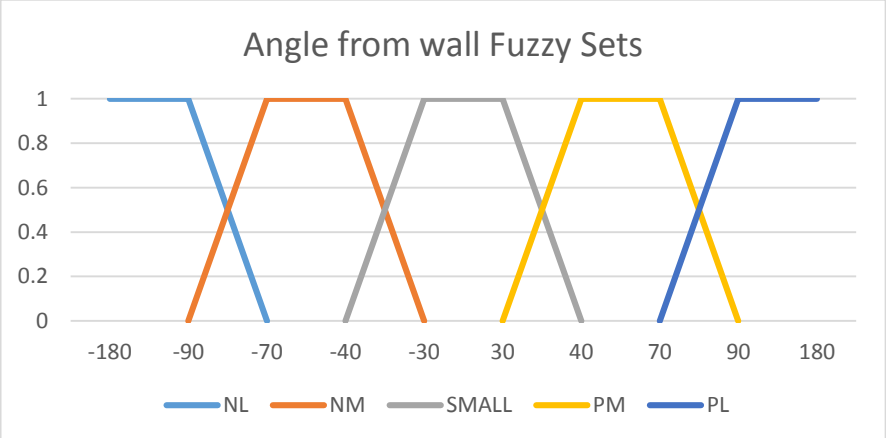


Figure 4.4 Angle from wall Fuzzy Sets

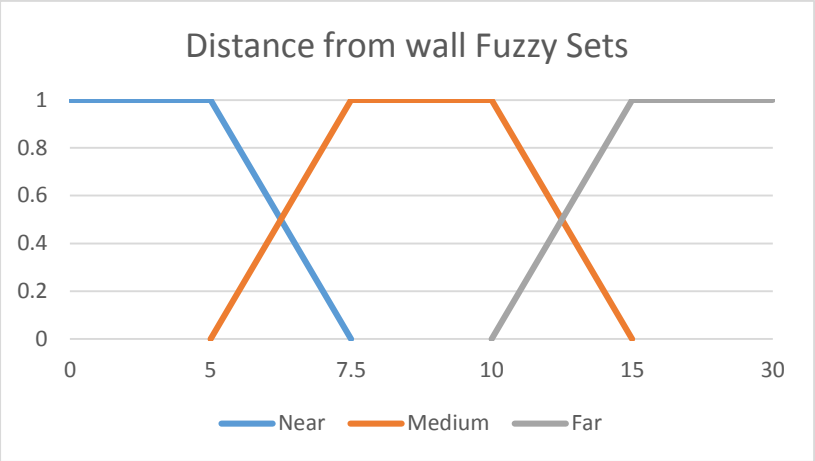


Figure 4.5 Distance from wall Fuzzy Sets

Now, the combination of the three input values corresponds to the states, and the actions correspond to the different steering angles.

4.3.2 Reinforcement Learning Parameters

States used by the Reinforcement Learning

Table 4.1 The ID of states for corresponding input combination (Distance from wall is near)

Angle from ball-> Angle from wall	PL	PM	PS	ZE	NS	NM	NL
PL	0	1	2	3	4	5	6
PM	7	8	9	10	11	12	13
SMALL	14	15	16	17	18	19	20
NM	21	22	23	24	25	26	27
NL	28	29	30	31	32	33	34

Table 4.1 shows the ID of the reinforcement learning states corresponding to the three input combination (Distance from the nearest wall is NEAR, angle from nearest wall, angle from ball). As there are two other regions left for the distance from wall input (i.e. Medium and Far), the ID numbers of the reinforcement learning states has a total of 104 (not shown here anymore). So there is a total of 105 states (ID numbers: 0 … 104) in this implementation, and the State-Action space is defined below.

States-Action Space used by the Reinforcement Learning

Table 4.2 State-Action space (y-axis: ID of RL States, x-axis: actions)

	0 degree	1 degree	-1 degree	5 degree	-5 degree	25 degree	-25degree
0	Q-Value	Q-Value	Q-Value	Q-Value	Q-Value	Q-Value	Q-Value
1	Q-Value	Q-Value	Q-Value	Q-Value	Q-Value	Q-Value	Q-Value
2	Q-Value	Q-Value	Q-Value	Q-Value	Q-Value	Q-Value	Q-Value
3	Q-Value	Q-Value	Q-Value	Q-Value	Q-Value	Q-Value	Q-Value
4	Q-Value	Q-Value	Q-Value	Q-Value	Q-Value	Q-Value	Q-Value
……	……	……	……	……	……	……	……
104	Q-Value	Q-Value	Q-Value	Q-Value	Q-Value	Q-Value	Q-Value

*Q-Value corresponds to each state-action pair

The number in the first column of Table 4.2 is the ID of the state. There are seven actions for the robot (i.e. 0, 1,-1, 5, -5, 25, -25 degrees of steering). The positive values mean turn the robot to the left, while the negative values mean turn to the right. It is the job of the reinforcement learning to formulate a policy that maps the actions to their appropriate states.

The SARSA(λ) algorithm was used in this research, and was set-up using the following parameters:

- Explore rate = 0.1, Trace decay rate λ= 0.5, Learning rate α= 0.1, and the discount factor γ= 0.7

Reward Function for the Integrated Ball Pursuit and Wall Avoidance

In general, during ball pursuit, the reward awarded is increased whenever the angle from the ball decreases. On the other hand, during the wall avoidance phase, whenever the robot gets too close to the wall, the reward given is increased if the robot steers away from the wall (i.e. angle from wall is bigger). An extra reward is added if the angle from the wall is larger than 90 degrees. When that happens, the robot will chase the ball again. Overall, the expected result of using this reward function is that robot is able to chase the ball safely. Detailed definition is shown below (Pseudo code 9).

Pseudo code 9: Reward function for ball pursuit and wall avoidance

1. **If the AngleFromWall is greater than or equal to 90 degrees and the robot is far from the wall, then:**
2. **Reward function for ball pursuit:**
    - If newAngleFromBall is less than oldAngleFromBall, then reward = 30 * (1 - newAngleFromBall / oldAngleFromBall).
    - An extra reward of 10 is awarded when the angle from the ball is within the range of [-1, 1] degrees.
3. **Else Special case (wall avoidance): If the ball is too close to the wall (the Distance is less than 15)**
4. **If the angle from the wall is less than 90 degrees:**
    - If newAngleFromWall is bigger than oldAngleFromWall, then reward = 30 * (1 - oldAngleFromWall / newAngleFromWall).
5. **Else if the angle from wall is bigger than 90 degrees:**
    - An extra 10 extra points is awarded.
6. **End**

4.4. Results and Analysis

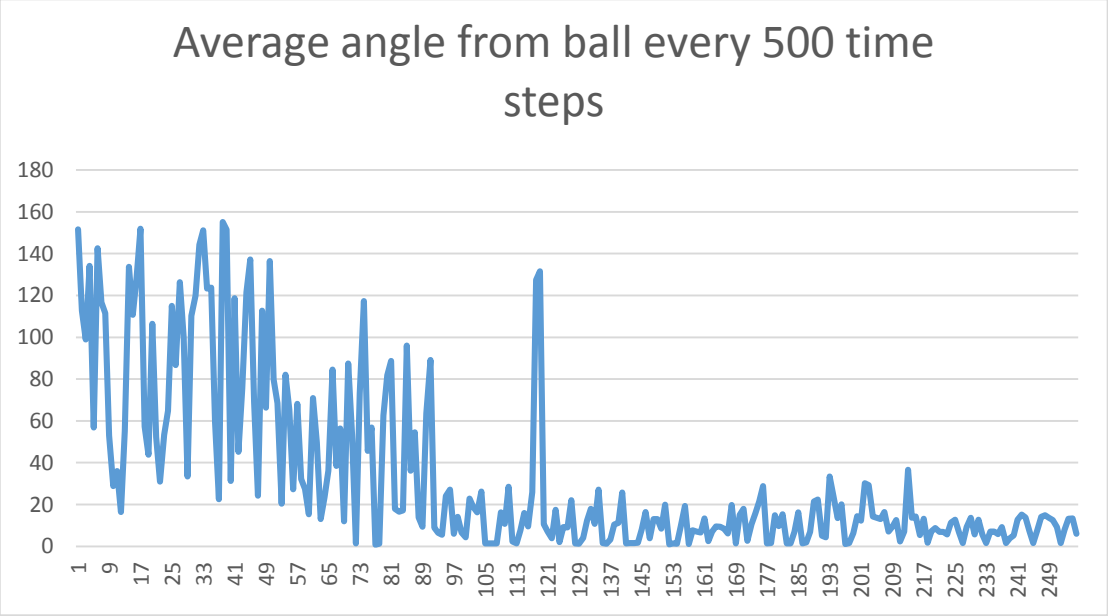


Figure 4.6 Average angle from ball (measured every 500 time steps) during robot training.

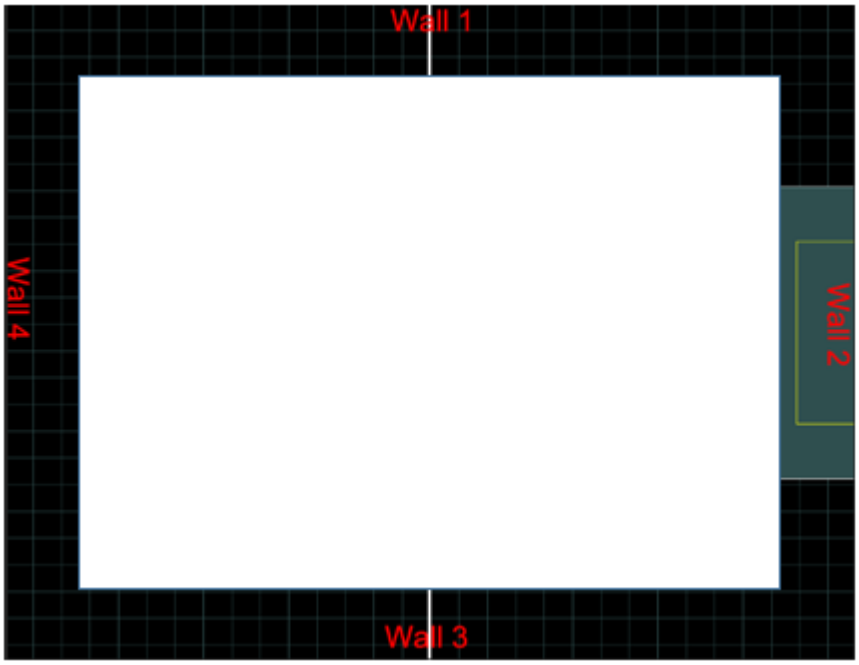


Figure 4.7 Pre-defined restricted area used in the experiments. The ball is initially placed within the black region depicted in the figure. The white region is the prohibited area.

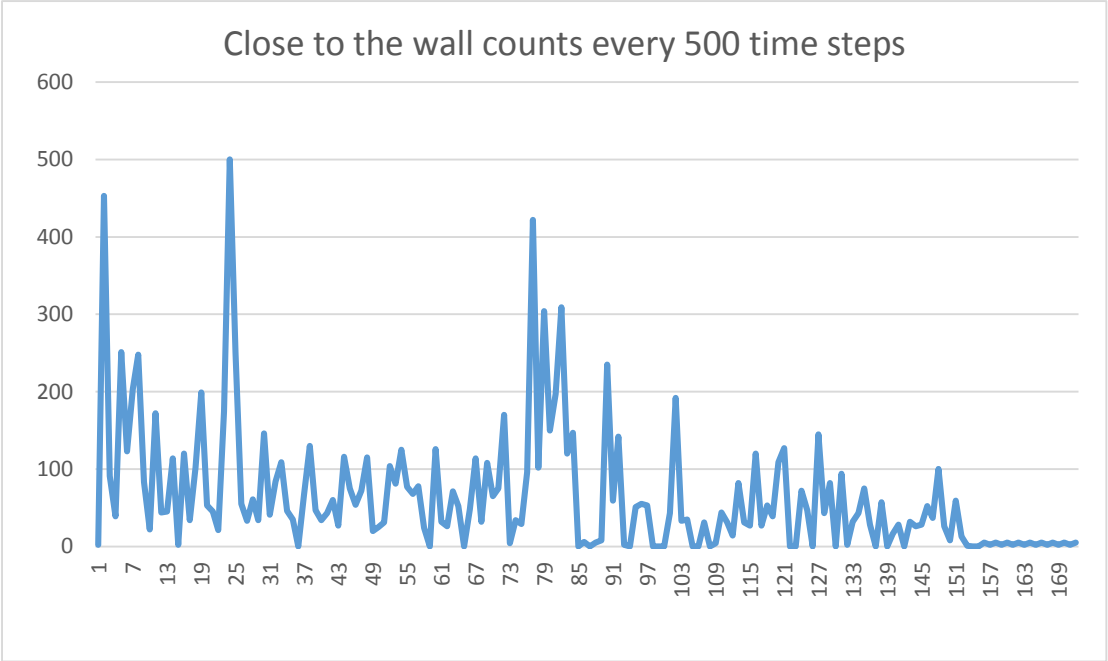


Figure 4.8 Close to the wall counts every 500 time steps

The data of Fig. 4.6 is collected under normal circumstances in the simulation. There are no interruptions or limitations in this experiment. As can be observed in the graph, the Fuzzy-RL is able to make the robot chase the ball, as the average angle from the ball decreases over time.

On the other hand, the data of Fig. 4.8 was collected in a restricted simulation, where

the ball is only allowed to move in a restricted region (black region that is close to the four walls in Fig. 4.7). During wall avoidance training, if the ball moves out of this restricted area, the ball will be reset to a random position within the black region. This is because the pitch is a very big place, if the ball is allowed to go everywhere, the robot will also follow and they may spend most of the time in the central area of the pitch - in this case the data is meaningless for training the robot to avoid the walls.

As the two figures (Fig.4.6, Fig. 4.8) show, it takes much longer time steps to train the robot to learn both ball pursuit and wall avoidance, as compared to learning only the pursuit behaviour. In real time, it may not be very long, approximately 5-10 min.

Figures 4.9 – 4.16 show the performance of the robot when it gets too close to each of the walls. In performance 1, the ball is very close to the wall and the robot just patrols around it, and does not get too close to the wall as the experiment expected. In performance 2, the ball is manually put to the central area of the pitch, and the robot turns to pursue the ball immediately. And as above figures show, the robot behaviours whenever it gets close to each wall are similar. This is because the algorithm only considers the nearest wall. It cannot distinguish between the different walls to act differently.

The core code (written in C++) of this implementation is attached in Appendix A, and the video test result can be viewed at http://youtu.be/Ibs-sDoU5VM

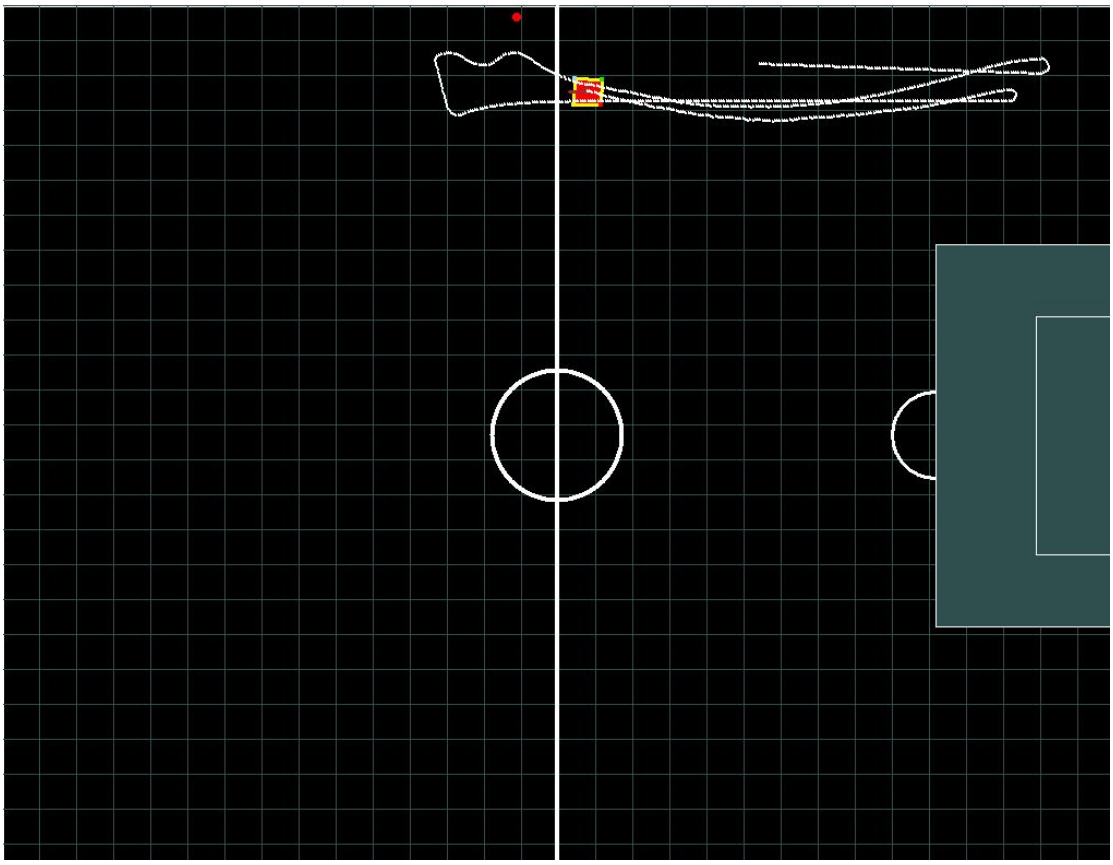


Figure 4.9 Trained sample close to wall 1 performance 1

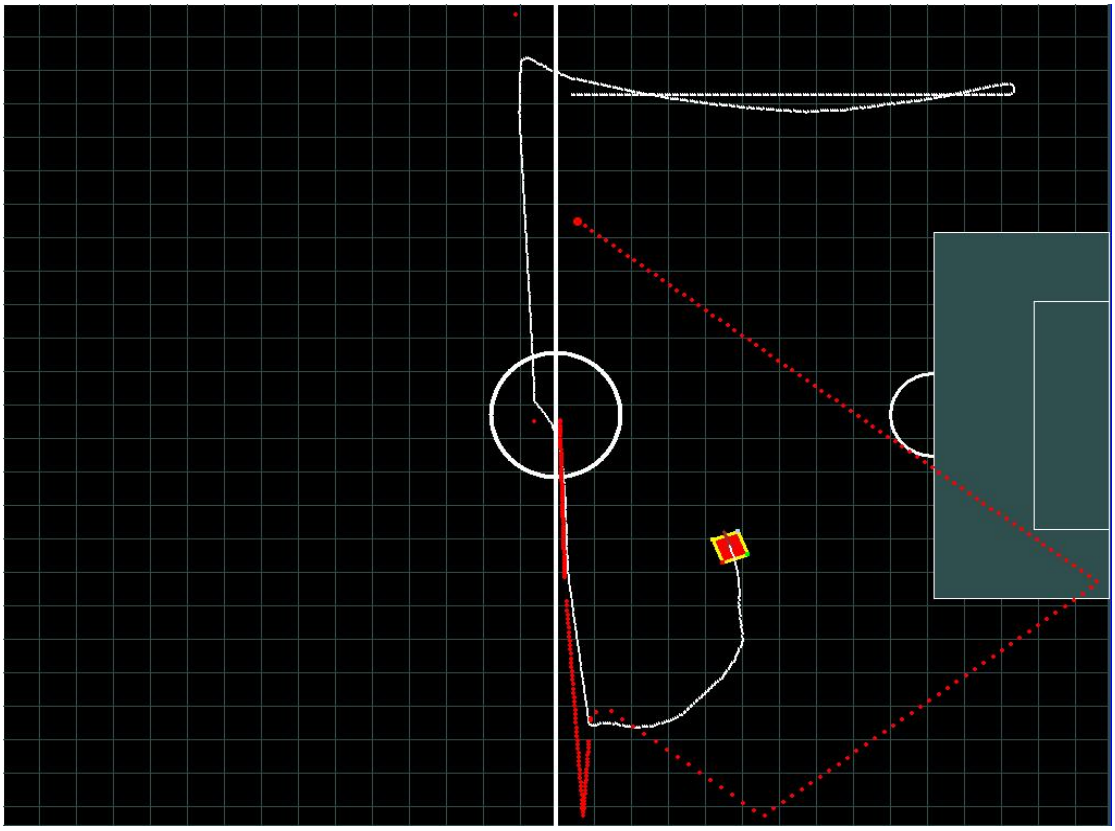


Figure 4.10 Trained sample close to wall 1 performance 2

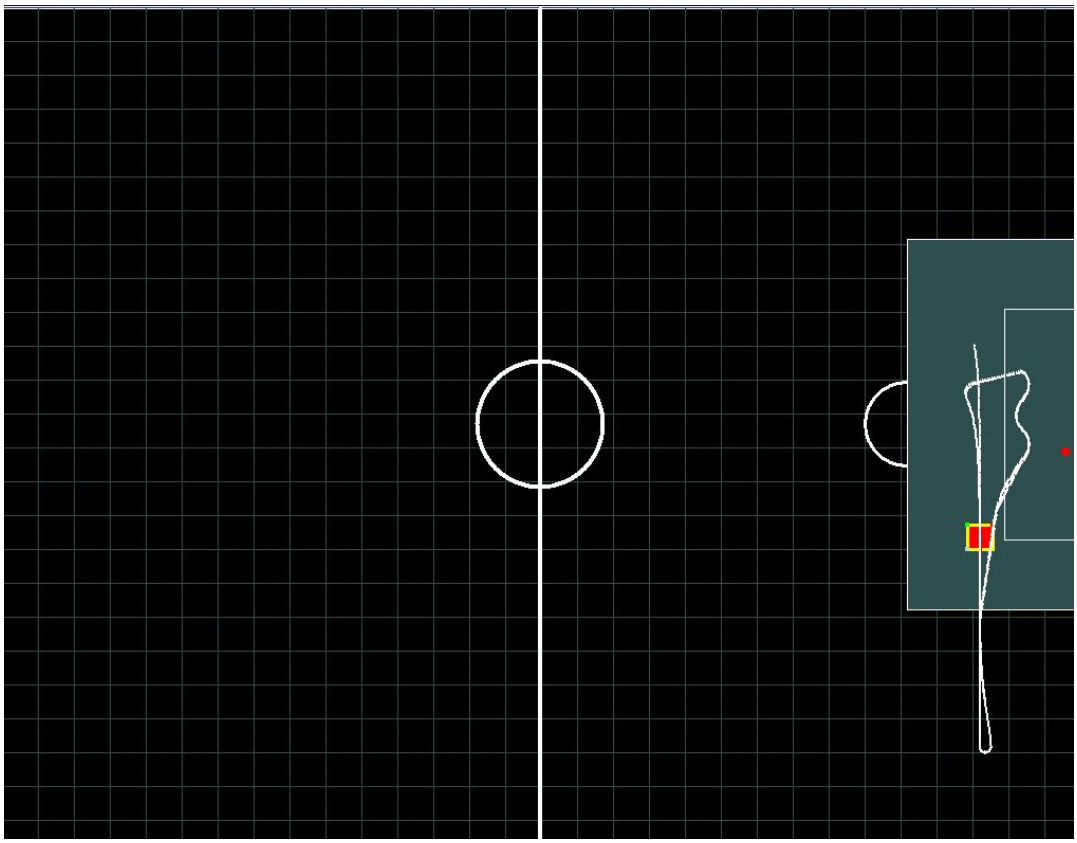


Figure 4.11 Trained sample close to wall 2 performance 1

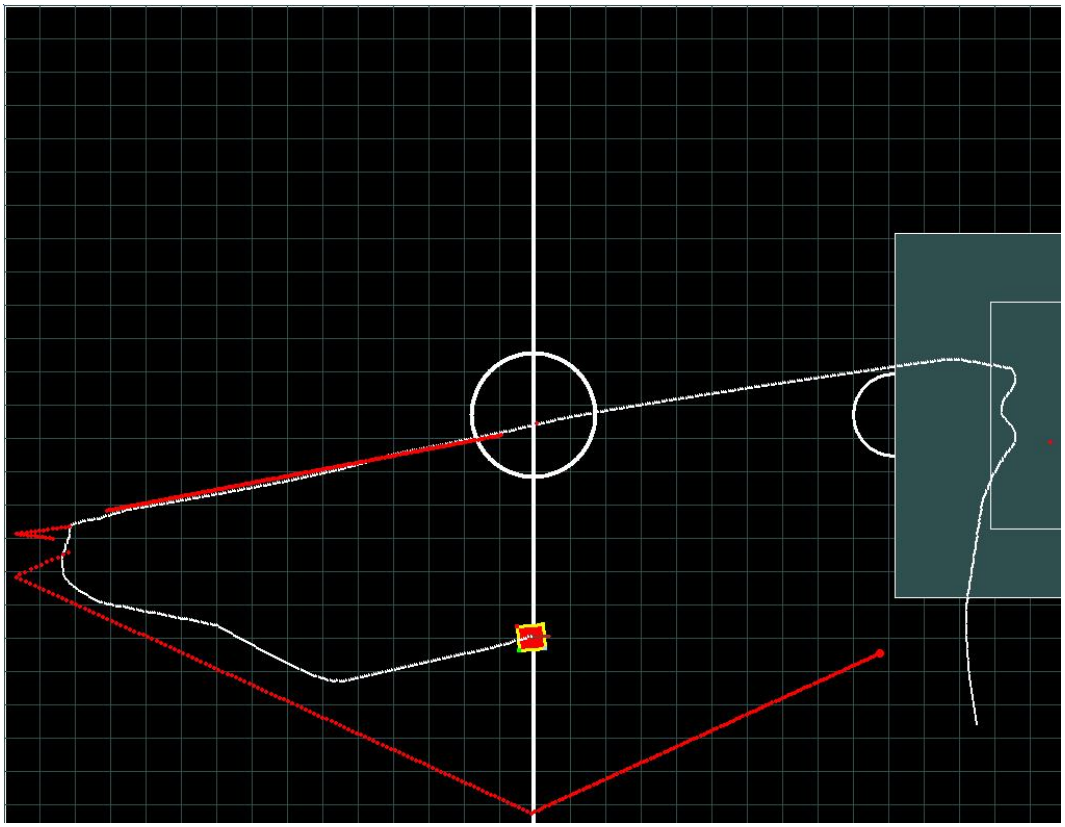


Figure 4.12 Trained sample close to wall 2 performance 2

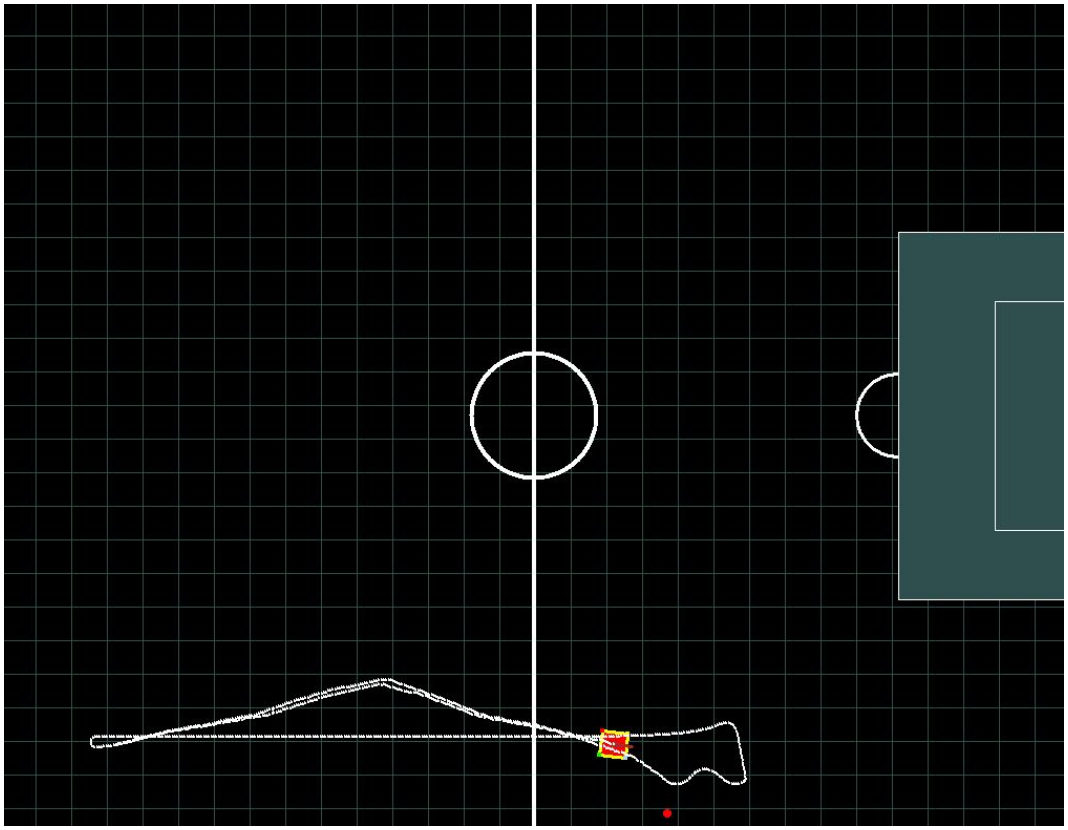


Figure 4.13 Trained sample close to wall 3 performance 1

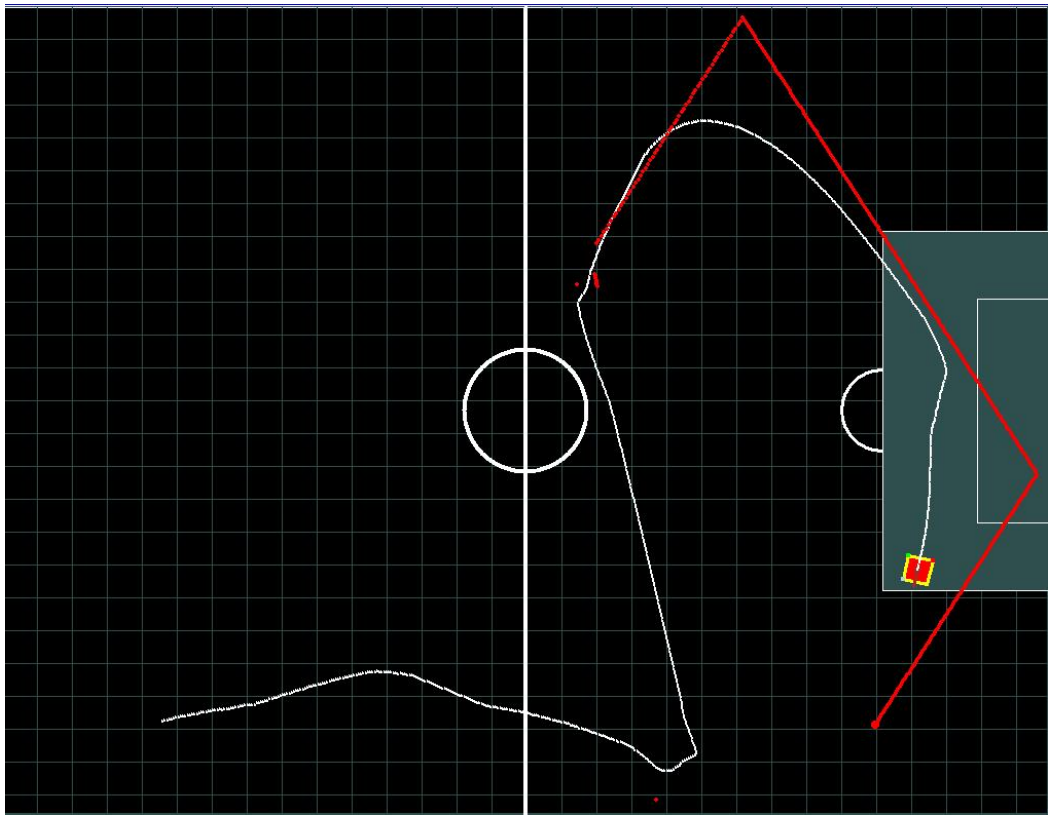


Figure 4.14 Trained sample close to wall 3 performance 2

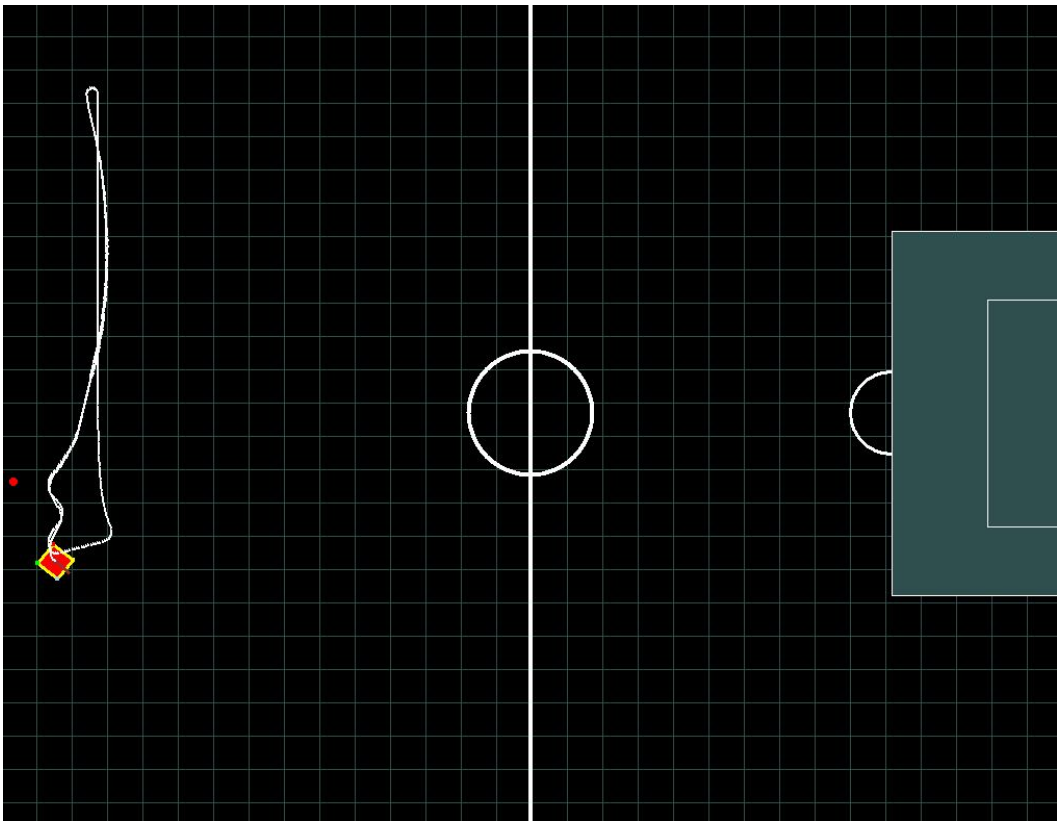


Figure 4.15 Trained sample close to wall 4 performance 1

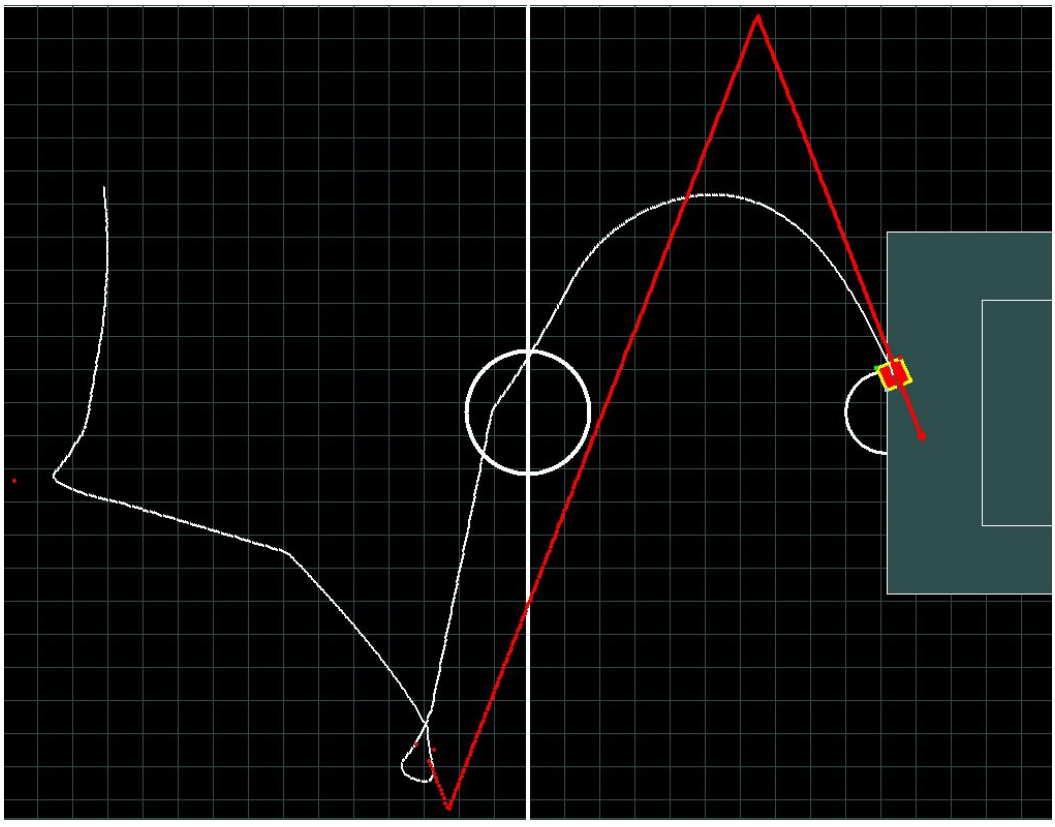


Figure 4.16 Trained sample close to wall 4 performance 2

4.4 Limitations of the Algorithm

As chapter 3 already introduced, the reward function is very important to the performance and from the experiment results in this chapter, we can see that although the robot achieves the wall avoidance behaviour, it moves a bit randomly whenever it gets close to the wall. It is hard to change this behaviour because it needs expert knowledge to adjust the reward function.

The other problem is, even though it just looks for the nearest wall, it already necessitates 105 reinforcement learning states. If there is a need to add some more features, more states will be required. Also, it is hard to train the system because there is no guarantee that every state can be visited enough during a simple and relatively short training phase. Theoretically, all states can be visited enough if given enough time for running the simulation program, it is hard to determine the length of the training time, because everything is running automatically in the simulation environment and based on the initial status of the ball and the robot some situation (states) may never happen. The only possible solution is manually set all kinds of initial status for robot and ball to meet all the situation (states), and this is hard to do when the number of states becomes very large.

Chapter 5
GNP with Trained Fuzzy-RL Nodes for Learning Multi-Behaviours

5.1 General Architecture

Two major changes were made to the original GNP, in order to allow the algorithm to incorporate trained control systems; therefore, making it a more powerful learning algorithm.

Firstly, a new Fuzzy-RL processing node is introduced in the GNP composition. The new processing node now runs a complete Fuzzy-RL system, trained for the ball pursuit behaviour, with steering angle computation only (defined in Algorithm 2b). Consequently, the new processing node requires some input value from the environment (i.e. angle from the ball), and outputs a continuous-valued action (steering angle). In general, each type of processing nodes can supply a complex behaviour for the robot. Previously, this is not possible in the original design of GNP.

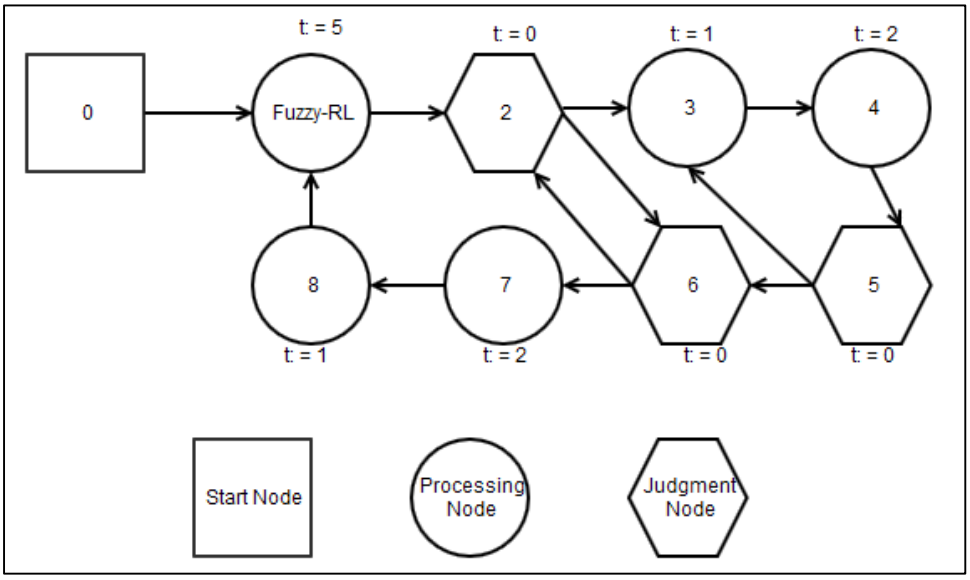


Figure 5.1 Modified GNP Individual used in the new algorithm

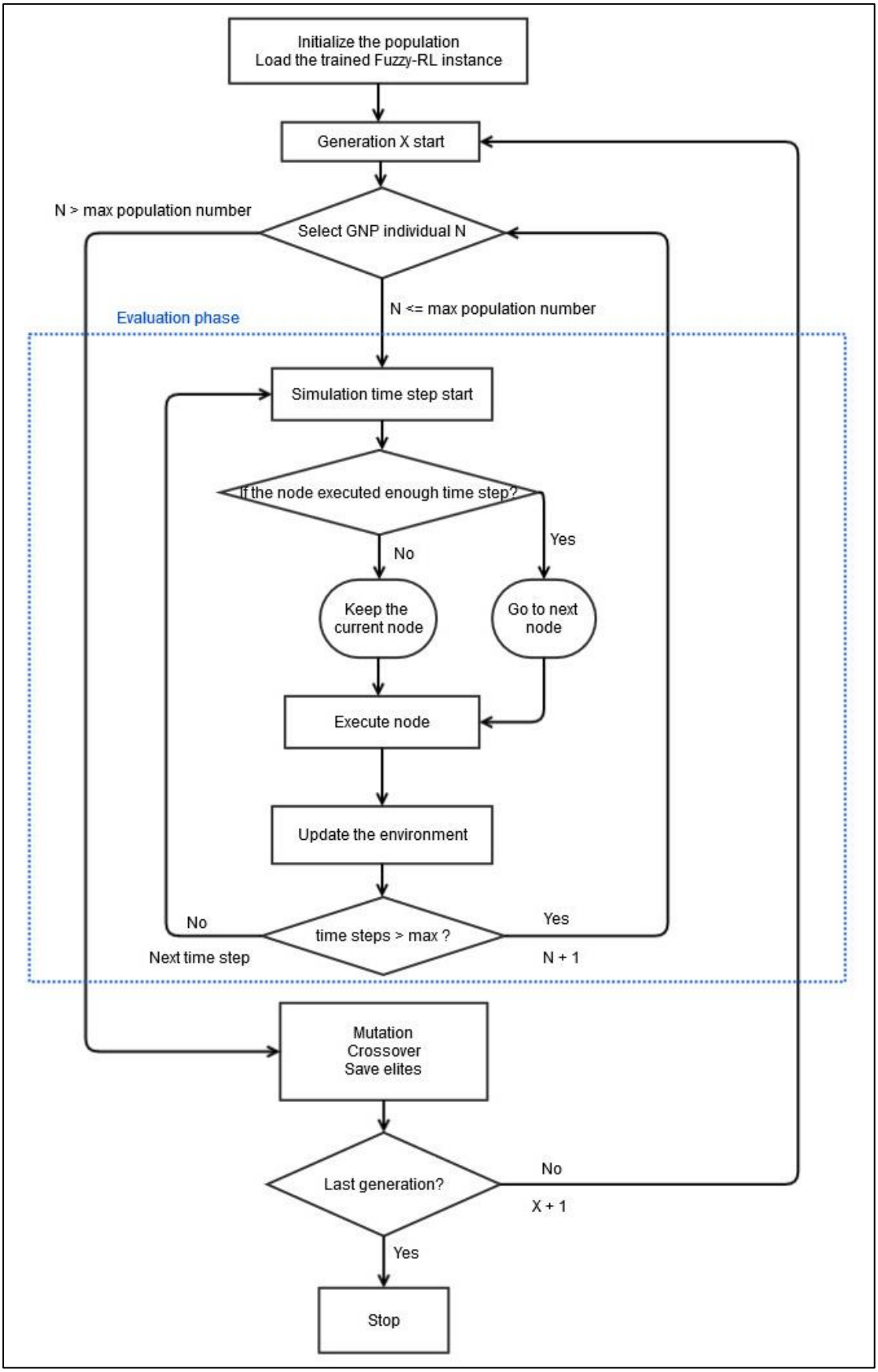


Figure 5.2 Schematic diagram of GNP with trained Fuzzy-RL nodes algorithm

Secondly, the other change is the new implementation of the execution time (this is referred to as “time delay” in the original GNP, (Katagiri, et al., 2000)) of each node, comprising the GNP individual. Referring to Fig. 5.1, a GNP individual is defined with 5 processing nodes, 3 judgment nodes, and a start node. As illustrated, there is a Fuzzy-RL processing node (for ball pursuit behaviour) with an execution time of 5. This means that the Fuzzy-RL will execute for 5 complete execution of the ball pursuit behaviour, before transitioning to the next node, which is a simple judgment node. The nodes executed within an individual solely depend on the environment conditions that the robot is experiencing while training. This is due to the decisions made by the judgment nodes. Also, during training, a fixed maximum training time is set, to evaluate the performance of the individual. The schematic diagram of GNP with trained Fuzzy-RL nodes algorithm is shown as Fig. 5.2.

Altogether, the changes made to the architecture allow the processing node to form a more complex behaviour more easily. For example, if the robot needs to move forward for a certain angle and distance, this behaviour can be achieved by a processing node that turns the angle to some value, and setting the execution time to a certain value to control the distance travelled.

In the example used in this thesis, only simple action processing nodes and Fuzzy-RL processing nodes were used, but there are no restrictions for the types of processing nodes in the new algorithm. Others, such as fuzzy logic control nodes or some machine learning algorithm-controlled nodes would be perfectly suitable, too. However, an important limitation is that the complex nodes have to be trained first before integrating it into the GNP architecture. A processing node should be able to present a stable behaviour, in order not to affect the fitness of an individual.

During the training phase of the GNP algorithm, the trained Fuzzy-RL node used is only running the greedy policy, for stability reasons. It is the job of the GNP to evolve the GNP individuals by changing the connections between the nodes, as well as allowing for mutations. On the other hand, during the testing phase, the Fuzzy-RL node uses the ε-greedy policy, in order to adapt to changes in the environment.

Lastly, the hill climbing algorithm, which is a greedy search algorithm that is often used to optimize evolutionary algorithms, is also used here. It is expected to help the GNP produce better individuals and also to accelerate the evolution process.

5.2 GNP with Trained Fuzzy-RL Pseudo Code: Training Phase

Pseudo code 10: GNP with Trained Fuzzy-RL

```
1.  Load trained Fuzzy-RL instance into the GNP individual.
2.  Initialize the population.
3.  Repeat (for each generation)
4.      Repeat (for each individual)
5.          Repeat (for each time step)
6.              Execute current node [*]
7.              Update the environment
8.              Update fitness of individual
9.              If the current node has been executed with enough time steps, go
                to next node
10.         Until time steps exceeded the maximum value for one individual
11.         Calculate final fitness of individual.
12.     Until all individual have been evaluated
13.     Apply hill-climbing algorithm for elites (e.g. top 3 individuals)
14.     Keep elites, and select more individuals using tournament selection
15.     Apply Mutation Operation
16.     Apply Crossover Operation
17. Until maximum generation is reached
18. End
```

[*] If the current node is a Fuzzy-RL node, take the input values and run the trained Fuzzy-RL instance

5.3 Problem-Specific Settings

5.3.1. Judgment Nodes

There are two different types of judgment nodes (Fig. 5.3), one is for judging the angle from the ball, and the other is for distinguishing which wall the robot is close to. The input value for the angle judgment node is the angle from the ball and the input values of the wall judgment node are the coordinates of the robot. The execution

time for judgment node is set to 0, meaning in one time step, the next node can be executed after the judgment node.

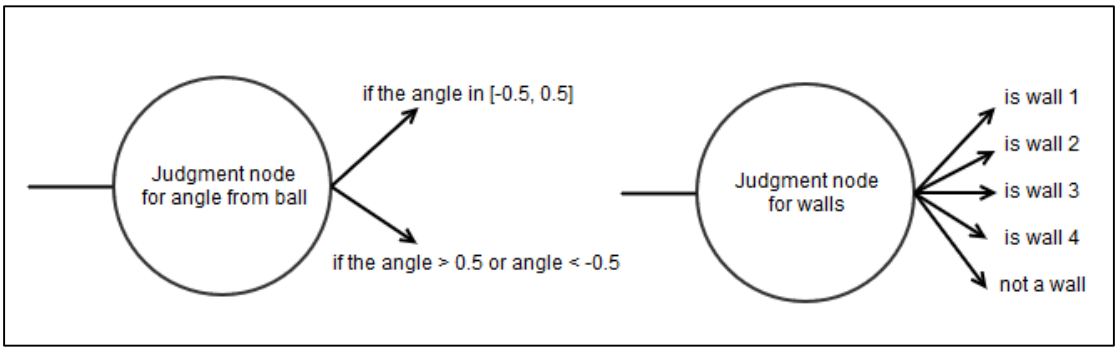


Figure 5.3 Judgment node settings

5.3.2. Processing Nodes

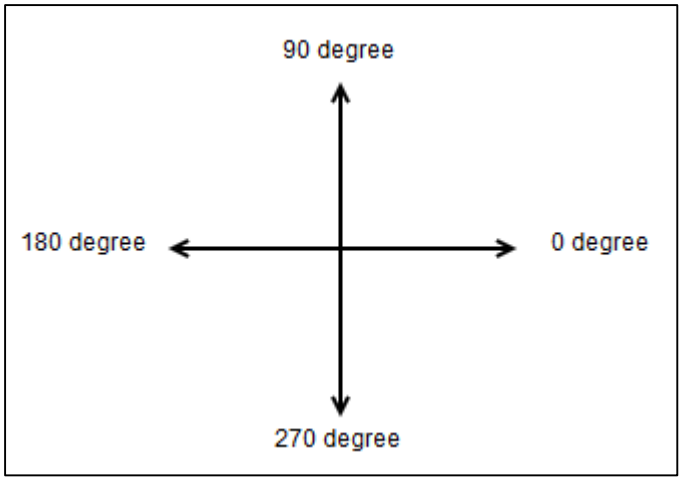


Figure 5.4 Absolute angle of the robot relative to the field

There are six different processing nodes, for six different behaviours in this implementation. The angle mentioned in actions below are the absolute angle of the robot relative to the field (Fig. 5.4).

Processing Node 1:

- Input: angle from Wall 1
- Description: Wall 1 avoidance
- Action: turn to 225 or 315 degrees corresponding to the angle from wall
- Execution time: 5

Processing Node 2:

- Input: angle from Wall2
- Description: Wall 2 avoidance
- Action: turn to 135 or 225 degrees corresponding to the angle from wall
- Execution time: 5

Processing Node 3:

- Input: angle from Wall 3
- Description: Wall 3 avoidance
- Action: turn to 45 or 135 degrees corresponding to the angle from wall
- Execution time: 5

Processing Node 4:

- Input: angle from Wall 4
- Description: Wall 4 avoidance
- Action: turn to 45 or 315 degrees corresponding to the angle from wall
- Execution time: 5

Processing Node 5:

- Input: distance from ball
- Description: Speed Control Action
- Action: set speed to one of the following values: 0.5, 1, 1.5, 2
- Execution time: 1

Processing Node 6:

- Input: angle from Wall
- Description: runs a trained Fuzzy-RL algorithm instance for ball pursuit
- Action: steering angle to turn the robot
- Execution time: 1

The Fuzzy-RL algorithm for ball pursuit as defined in chapter 3, is used here as a processing node. All the parameters are the same as those defined in Algorithm 2b, and a trained state-action space is used. The ε-greedy policy is temporarily changed to greedy policy during the training phase of GNP, but the ε-greedy policy is used during the testing phase in order to allow the robot to adapt to any possible changes in the environment.

5.3.3. GNP Fitness Function for Integrated Target Pursuit and Wall

Avoidance Behaviours

The fitness function and the training environment is the most important part of the GNP with Trained Fuzzy-RL nodes. In order to achieve the different robot behaviours for avoiding the different walls, the GNP individual is subjected into four different environment settings during the training phase. The difference between these four environments is the position of the ball; it is set to a fixed point closed to each of the walls. The initial positions of the robot are all in the central area of the pitch.

The fitness function consists of three parts: one for ball pursuit behaviour, one for speed control behaviour, and one for the wall avoidance behaviour.

Speed Control

The fitness function for speed control behaviour (Pseudo code 11) is the simplest one, it increases every time step if the robot moves with the right speed.

Pseudo code 11: Fitness function for speed control behaviour

1. **If** (DistanceFromBall < 10 and speed = 0.5) **or** (DistanceFromBall∈ [10, 20) and speed = 1.0) **or** (DistanceFromBall∈ [20, 50) and speed = 1.5) **or** (DistanceFromBall>=50 and speed = 2.0)
2. **Then** fitness += 6.

Ball Pursuit

The fitness function for ball pursuit behaviour can be the same as the reward function used in the Fuzzy-RL algorithm (Algorithm 2b), but that one is too specific and may cause some slight fluctuations of the fitness value. In order to make the fitness function perfectly stable and accurate enough to estimate the quality of an individual, a complete fuzzy logic system is used here. The parameters for the fuzzy sets and the fuzzy rules are shown in Fig. 5.5 and Table 5.1. The final fitness value is calculated via the centre of mass formula.

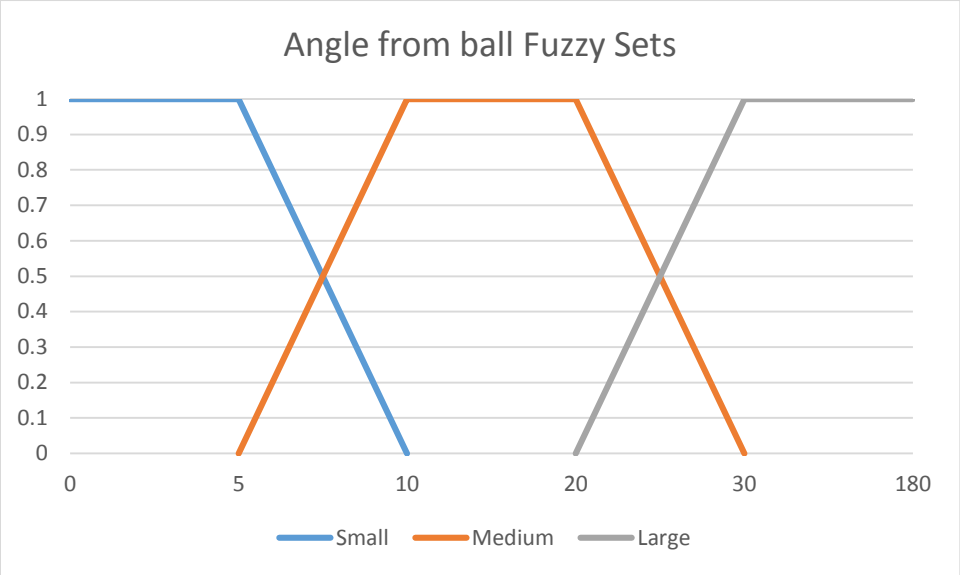


Figure 5.5 Angle from ball fuzzy sets

Table 5.1 Fuzzy rules for calculating the ball pursuit behaviour fitness.

Angle	Fitness (Rules outputs)
Small	8
Medium	4
Large	0

Wall Avoidance

The fitness function for wall avoidance employs a scheme based only on punishment, show as pseudo code 12.

Pseudo code 12: Fitness function for wall avoidance

1. **If** the distance from the wall is less than 15 units
2. **then** decrease the fitness value by 50

There is no reward given for taking an action for wall avoidance. Otherwise, the robot may move close to the wall, and respectively avoid the wall to obtain high fitness, and this is not desired. This was observed through the experiments performed but is not shown here.

Fitness Function Summary

The fitness function now is still not good enough for the algorithm. When actually running the system under testing phase, some bad individuals still receive very high fitness. Four wall avoidance processing nodes are simple actions turn to a fixed angle, if one individual only executes these processing node and combine with ball pursuit processing node or speed control processing node, they may still get very high fitness. When testing these kinds of individuals, they just make the robot turn around (as observed in the experiments). In order to eliminate these kinds of situations, two more rules are added to the fitness function:

If the robot moves away from the ball the fitness value decreases a bit, and at the end of each individual's training simulation, the distance of the robot from the ball is measured and used as a component in the fitness calculation. Consequently, when the robot moves closer to the ball it gets a higher fitness.

The final fitness function is shown in Pseudo code 13.

Pseudo code 13: Final fitness function

1. Individual starts
2. Fitness = 0
3. **Repeat** (each time step)
4. **If** DistanceFromWall < 20
5. **If** the AngleFromWall <= -90 or AngleFromWall > 90 (AngleFromWall in figure 4.1)
6. **Then** Fitness += 20 + Ball Pursuit Reward [*]
7. **Else**
8. Fitness += Ball Pursuit Reward [*]
9. **If** (DistanceFromBall < 10 and speed = 0.5) **or** (DistanceFromBall∈[10, 20) and speed = 1.0) **or** (DistanceFromBall∈[20, 50) and speed = 1.5) **or** (DistanceFromBall>=50 and speed = 2.0)
10. **Then** Fitness += 6
11. **If** DistanceFromWall < 15 then Fitness -= 50
12. **If** the robot moves away from the ball then Fitness -= 15
13. **Until** time steps exceeded the maximum training time for one individual
14. Fitness += 500 – 10*DistanceFromBall
15. Individual training ends.

[*] Ball Pursuit Reward is generated by the Fuzzy logic system introduced

5.3.4. Parameters for GNP

Number of population: 200
Number of mutation: 77
Number of crossover: 120
Tournament size: 5
Probability of mutation: 0.1
Probability of crossover: 0.5

5.3.5. Hill-climbing Algorithm

This algorithm tries to improve the fitness of a GNP individual by examining the connections one node at a time by way of brute force greedy search. Pseudo code 14 is the pseudo code of Hill-climbing algorithm.

Pseudo code 14: Hill-climbing algorithm

```
1.  Hill-climbing algorithm for an individual
2.  Start
3.  Set the MaxFitness = individual's fitness acquired through GNP evolution
4.  Repeat (for each node Nn)
5.      Repeat (for each connection Cm)
6.          Set the BestConnection = Cm
7.          Repeat (for each ID of node exclude the start node)
8.              Set the Cm = ID
9.              Evaluate the individual and get newFitness
10.             If newFitness > MaxFitness
11.             Then MaxFitness = newFitness and BestConnection = ID
12.         Until ID exceeded the maximum value
13.         Set Cm = BestConnection
14.         Set individual's fitness = MaxFitness
15.     Until all connections in a node been visited
16. Until all nodes in an individual been visited
17. End
```

As hill-climbing algorithm tests every possible connections for each node, it takes long time for running on one individual, and the time is increasing dramatically with the increasing number of nodes. From the experience of experiments without

hill-climbing algorithm, the best fitness value is always stay the same for a long time in some period. The hill-climbing algorithm is expected to remit this situation, in order to accelerate the evolution. As it is a time-consuming job, it is only for the top three individuals and there is no need to run it in every generation. The hill-climbing algorithm is applied every five generations in the experiments.

5.4 Results and Analysis

Figure 5.6 shows one possible GNP individual that is able to accomplish the objectives defined, with the minimum number of nodes. At first, the number of each GNP nodes is set to one instance. However, in the actual experiments, using the settings shown in Fig. 5.6, the algorithm just can't find a good individual within 200 generations. The best individual found in one experiment is able to pursue the ball, control the speed and only avoid two walls.

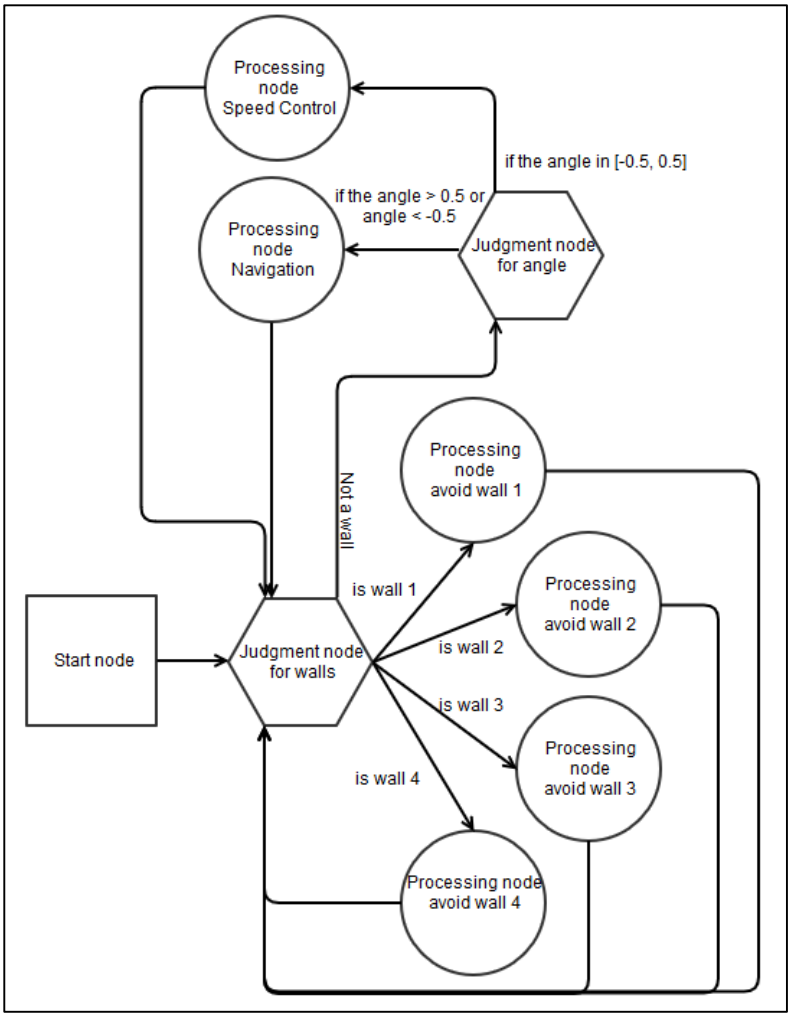


Figure 5.6 Sample GNP individual with the minimum number of nodes. Note that the algorithm may generate a variety of individuals with different nodes and connections.

The solution is to add more nodes to a GNP individual, except there is always only one start node. From the experiments, it was observed that increasing the number of each type of nodes to two, helped in obtaining and a good individual (a good solution). The experiments results are shown below.

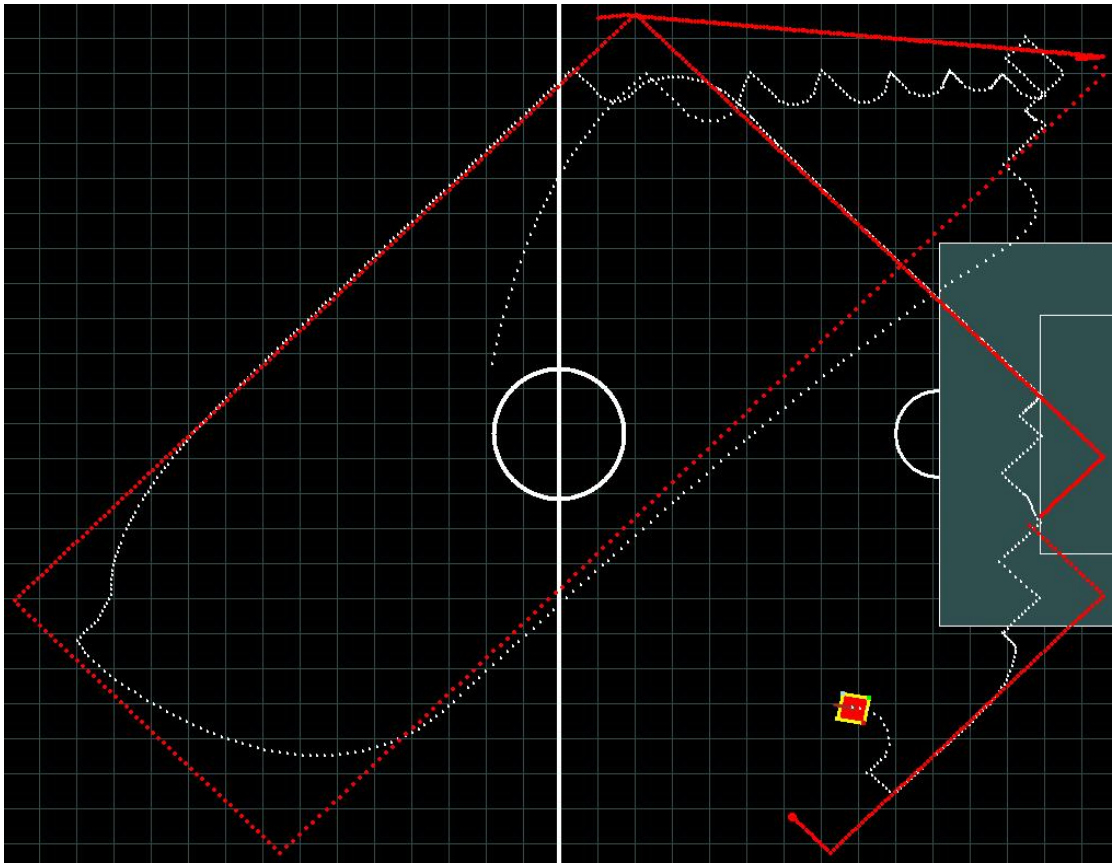


Figure 5.7 General performance of a good individual

As the Fig. 5.7 shows, this good individual is able to steer smoothly, control the speed corresponding to the distance from ball, and most importantly it is able to distinguish four walls and take different actions. More figures are attached below to show the behaviour when the robot is close to the wall.

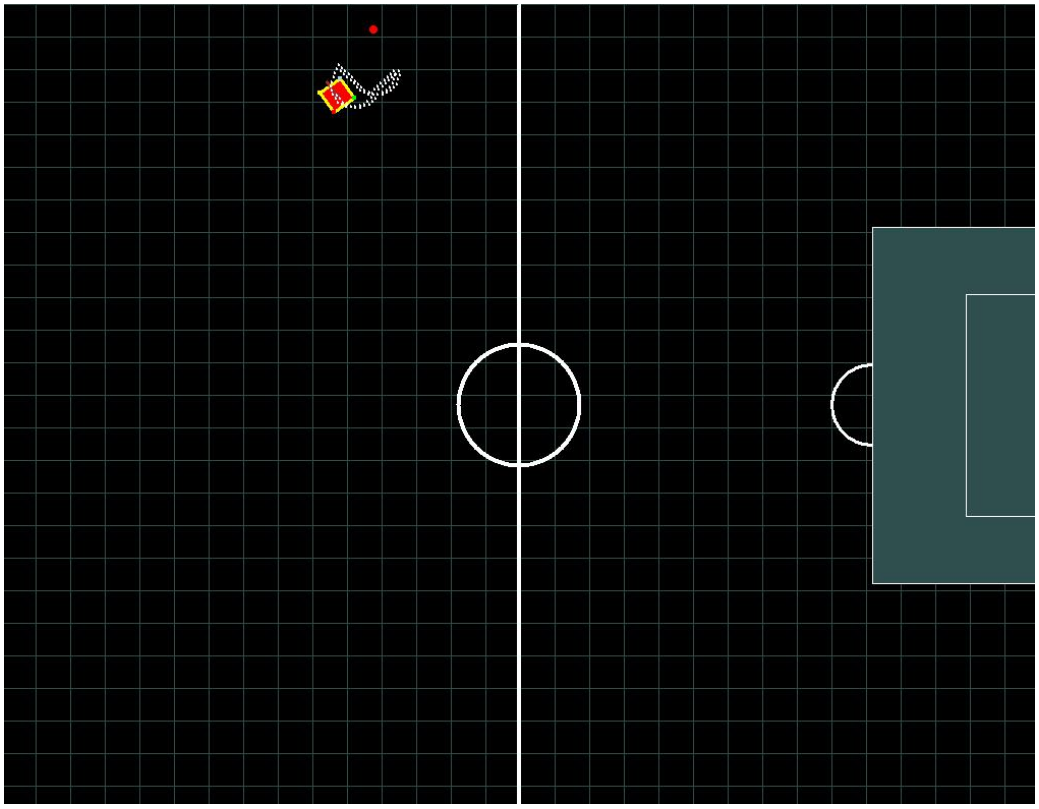


Figure 5.8 The performance close to wall 1

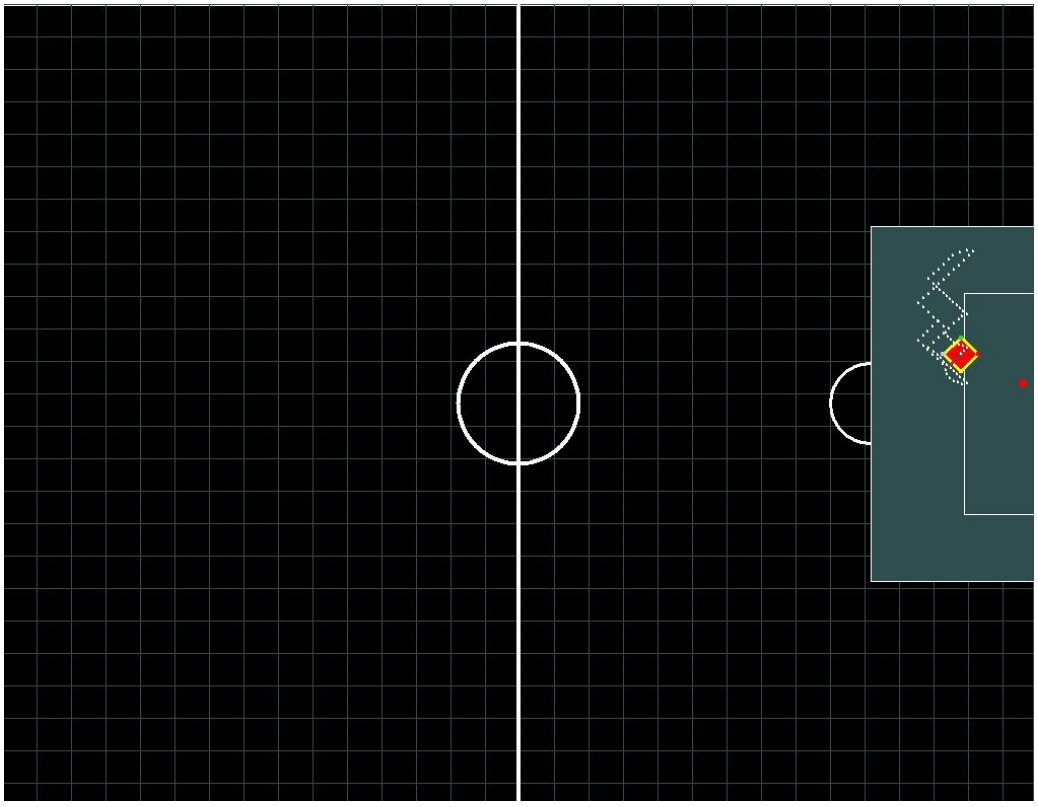


Figure 5.9 The performance close to wall 2

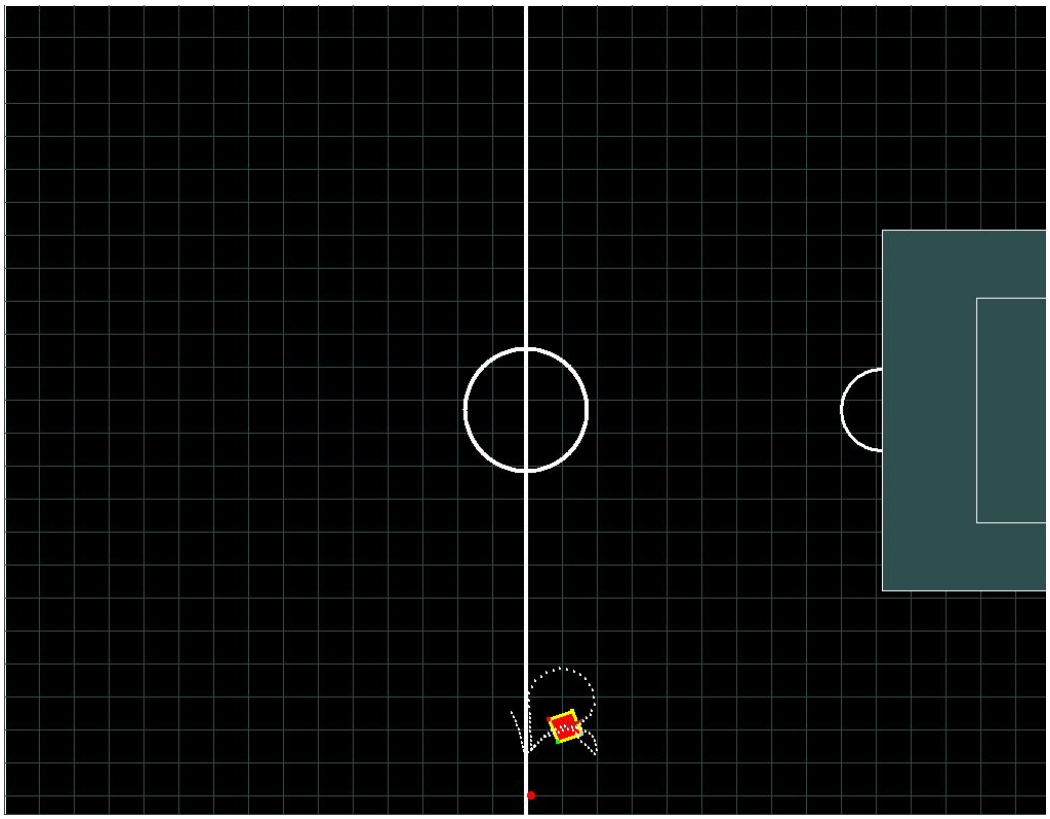


Figure 5.10 The performance close to wall 3

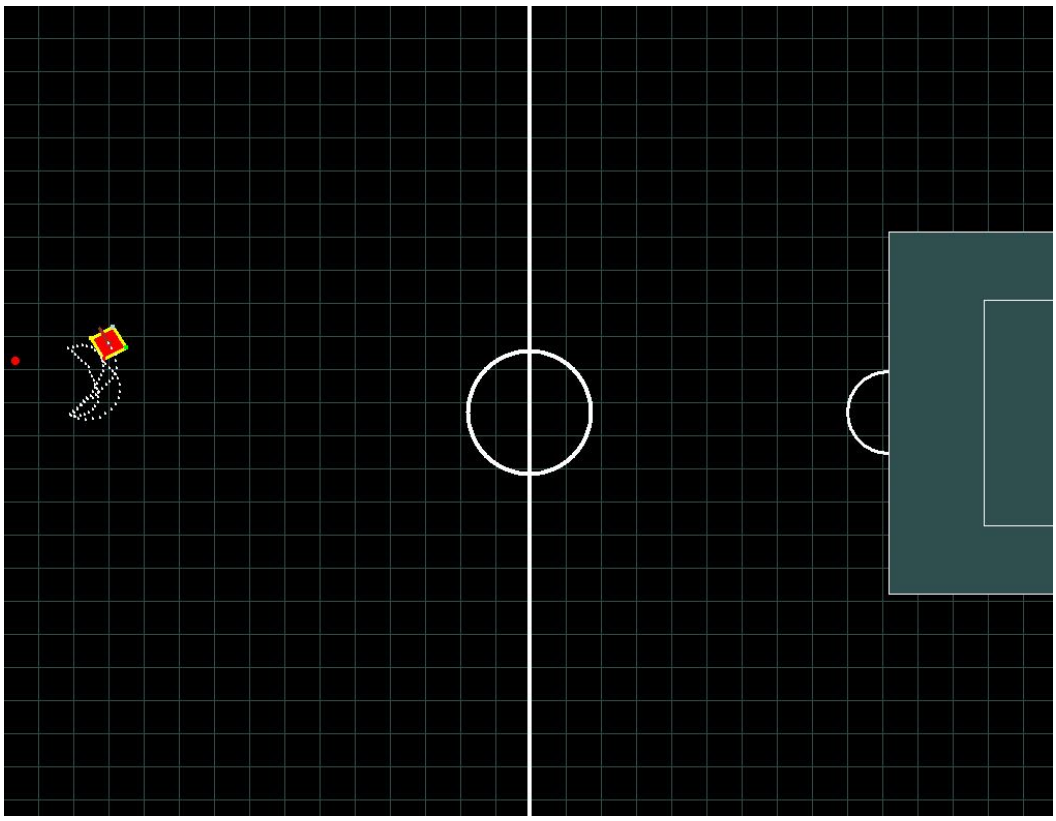


Figure 5.11 The performance close to wall 4

The four figures above (Fig. 5.8-5.11) show the robot behaves as desired. It distinguishes the four different walls and chooses the right action. In contrast to the Fuzzy-RL algorithm defined in chapter 4, the wall avoidance behaviour here is fully controlled by those four processing nodes.

The hill-climbing algorithm was also used in the experiments. The two figures below (Fig. 5.12, Fig. 5.13) show the fitness values of the top 3 individuals during the evolution phase, with and without the utilisation of the hill-climbing algorithm.

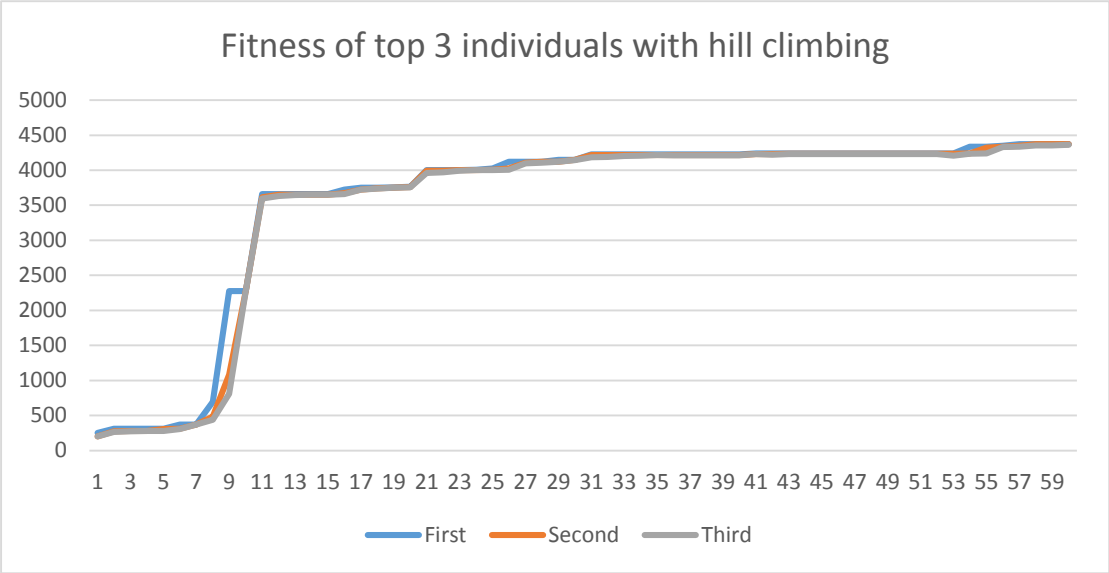


Figure 5.12 Fitness of top 3 individuals with hill climbing

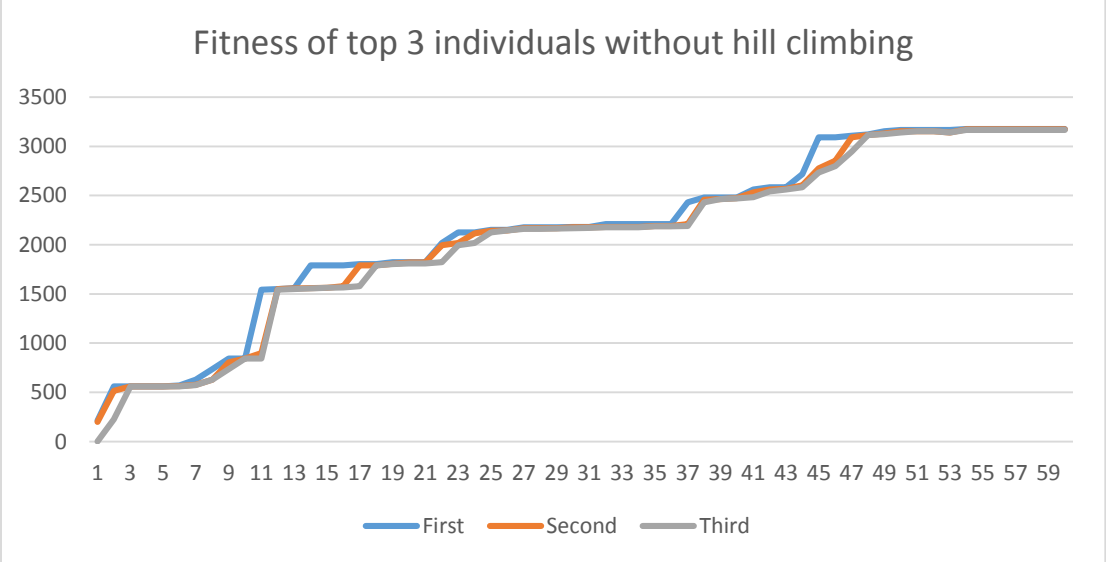


Figure 5.13 Fitness of top 3 individuals without hill climbing

The data in these two figures (Fig. 5.12, Fig. 5.13) were collected from two experiments, with all other settings kept the same. As we know, the performance of an evolutionary algorithm is not stable, sometimes the fitness goes very high in a few generations and sometimes it may keep a low value for a long time. However, in the

long run, these two figures are still representative of the general outcome. The hill-climbing algorithm was applied after every ten generations. As can be observed from the graph, there is a huge increase in fitness value at the 10^{th} generation. Consequently, it accelerated the evolution process towards finding a good individual.

The core code (written in C++) of this implementation is attached in Appendix B, and the video test result can be viewed at http://youtu.be/woqMnbO-CKg

5.5 Limitations of the Algorithm

There are two major difficulties in this algorithm. One is the setting for the fitness function and the training environment and procedure. The fitness function is closely linked to the training environment, any improper settings among them can lead to failure in learning. Furthermore, if more behaviours are required, the fitness function will become more complex, and it may take some time to test and improve. Another problem is the same with other evolutionary algorithms, it does not guarantee to produce the optimal results, and it may need to run the training phase for a long period of time to get a good individual. Lastly, there is still room for improvement in the algorithms training implementation, as it takes about 30 minutes to train the system for 100 generations, although the program is already parallelized. In general, this algorithm needs to be implemented carefully to achieve a good performance in its training phase. Nevertheless, once a robot is trained, it can run very fast.

Chapter 6
Summary and Future Work

Table 6.1 Comparison of different algorithms

	Fuzzy Logic System	Fuzzy-RL	GNP (Katagiri, et al., 2000)	GNP with RL (Mabu, et al., 2007)	Fuzzy-GNP with RL (Mabu, et al., 2011)	GNP with Trained Fuzzy-RL Nodes
Capable of Multi-behaviour	Yes(multiple FLS)	Yes Chap.4	Yes *	Yes *	Yes *	Yes Chap.5
Calibration requirements	Hand-calibrated	Reward function	Fitness function	Fitness function	Fitness function	Fitness function Chap.5.3.3
Difficulty of setting the fitness/reward function for multi-behaviour	N/A	High Chap. 4.3.2	Theoretically High **	Theoretically High **	Theoretically High **	Medium Chap.5.3.3
Allows integration of multiple algorithms	N/A	N/A	No	No (only RL)	No (only Fuzzy&RL)	Yes Chap.5.1
Allows integration of other trained complex nodes using different AI algorithms	N/A	N/A	No	No	No	Yes Chap.5.1
Allows integration of simple processing nodes into GNP	N/A	N/A	Yes	Yes	Yes	Yes Chap.5.3
Allows complete FLS integration into a GNP individual	N/A	Yes	N/A	N/A	No	Yes Chap.5.3
Uses an additional learning mechanism that aids the evolutionary approach	N/A	N/A	No	No	No	Yes, Hill-climbing
Capable of on-line learning	No	Yes	No	Yes	Yes	Yes

[*] No previous attempt in the literature.

[**] The difficulty is estimated on a single behaviour problem tested in chapter 3.4

Three major algorithms, and their hybrid variants were introduced and tested in this thesis (i.e. Fuzzy logic control, Reinforcement learning and Genetic network programming, Fuzzy-RL, GNP with RL, GNP with trained Fuzzy-RL). Using a simulation of the robot soccer platform, ball pursuit and ball pursuit with wall avoidance behaviours were used as benchmarking test beds. As observed in the experiments, the algorithms have different advantages and disadvantages. The table above shows some of features tested for evaluating the performances of these algorithms.

Pure Fuzzy Logic (Steering angle and speed control)

From the results of experiments, it was able to produce an almost perfect performance (by visual inspection), provided that there is an expert prior knowledge to set all the fuzzy sets and the fuzzy rules, even though it is very hard and time consuming to calibrate, when multiple objectives are required.

Reinforcement Learning

Reinforcement learning is simple and efficient, but it is difficult to apply to control problems with continuous-valued outputs. The combination of reinforcement learning and fuzzy logic control was able to overcome their own weaknesses. Two different integration architectures of reinforcement learning and fuzzy logic were implemented for ball pursuit in this work:

- Algorithm 2a – RL with fuzzified inputs
- Algorithm 2b - RL integrate with a complete fuzzy logic system

Algorithm 2a integrated only a fuzzification component to the reinforcement learning, which is equivalent to having nested if-then decision-control statements. The results of experiments clearly show that the performance of Algorithm 2a is worse than Algorithm 2b, which comprises the whole fuzzy logic system. The Fuzzy-RL algorithm (Algorithm 2b) surmounts the shortcomings of reinforcement learning and fuzzy logic control algorithm while keeping their strengths. However, when facing multiple objectives, the training phase becomes more and more complex to manage; it is very hard to define a reward function for a specific multi-behaviour.

Genetic Network Programming

The genetic network programming is based on genetic algorithm and provides a flexible graph structure for storing extra computational information (i.e. rules, processing nodes, judgment nodes, allotted node execution time). It was first proposed by H. Katagiri, K. Hirasawa and J. Hu in 2000, and a Tile World game was used to demonstrate the algorithm. The execution of the GNP algorithm can be

divided into a training phase and testing phase, just like other evolutionary algorithms. The most outstanding advantage of GNP is that once a good individual is obtained from training phase, it can be used in a whole new dynamic environment for the same objective, because the graph structure is able to provide sufficient rules for operating in a dynamic environment. As a result, it is extremely fast and non-computationally intensive when running a GNP individual.

Genetic Network Programming with RL

In order to obtain better results in dynamic environments, the GNP was extended by combining it with reinforcement learning (Mabu, et al., 2007). It considers different types of processing nodes as sub-nodes, and uses reinforcement learning to select the best sub-node when executing a parent processing node of the GNP. Now, the reinforcement learning takes most of the responsibility for the performance of an individual both in training or testing phases. The RL helps make the structure of a GNP individual more compact, and even more flexible.

From the experiments performed in this thesis, some difficulty during training was experienced. In the GNP-RL algorithm, the reinforcement learning component explores a variety of possible actions during the training phase; unfortunately, this approach does not guarantee finding the best solution based solely on the fitness value of the individual. This eventually lead to some confusion for picking the best trained robot for a single behaviour (Chapter 3.4); probably even worse in multi-behaviour problem. To alleviate the problem, the top five GNP individuals were further tested to pick the best one by human visual inspection.

In another related work, a variant called the Fuzzy Genetic Network Programming with Reinforcement Learning algorithm was tested for wall following behaviour for a bi-wheel robot (Mabu, et al., 2011). A fuzzification component was added to the GNP with RL algorithm to deal with inputs which are continuous-valued.

Genetic Network Programming with Trained Fuzzy-RL

A new novel architecture was introduced in this research. It uses both trained complex processing nodes and of simple action processing nodes in the GNP architecture. The complex processing node is able to output different actions corresponding to changes in the environment. As an example of the proposed architecture, a complex processing node was defined to be a trained Fuzzy-RL node. In contrast with the original GNP with RL algorithm (Mabu, et al., 2007), the new algorithm derived the different connections between the judgment nodes, complex

trained processing nodes and simple processing nodes to generate the desired multi-behaviour.

During the testing phase, the Fuzzy-RL node has the responsibility of adapting to changes in the dynamic environment. The GNP with trained Fuzzy-RL node algorithm achieved ball pursuit and wall avoidance in the experiment while keeping its flexible and compact structure. This is a solid evidence that proves that the new algorithm is highly extendable and possesses a strong ability to learn multiple objectives. The graph structure and complex nodes have advantages in dealing with the dynamic environment, in other words, the algorithm is excellent for any control system (robot control, traffic light control, elevator control, etc.). Nevertheless, the core ability of the algorithm is to make a complex judgment and give the desired output. It is also applicable to other problem domains that require determining the suitable judgment and processing nodes. For example, when using the algorithm for data mining purposes, it is simple to build a powerful classifier by integrating different kinds of judgment nodes.

Development of the new GNP with Trained Fuzzy-RL Nodes

In the original Fuzzy GNP with RL paper (Mabu, et al., 2007), the RL algorithm used simple processing nodes that take an action based on the state of the world. The RL algorithm was used to learn a policy that maps the actions to the nodes, with the aim of maximising the rewards. The simple actions are represented as sub-nodes, enclosed within a parent node in the GNP architecture. On the other hand, in this research, in particular in the Fuzzy-RL integration part, the state of the RL was based on a fuzzified information that feeds on the angle from the ball.

From the initial single behaviour learning test bed, the Fuzzy-RL integrated architecture succeeded in learning the ball pursuit behaviour. Consequently, it was then hypothesized that any of the processing nodes within a GNP individual can be transformed into a complex trained intelligent system; thereby, allowing any AI algorithm to be used as a complex node within a GNP individual. As a general rule, given a complex multi-behaviour learning problem, the idea in this thesis is to subdivide the problem into multiple more manageable sub-problems. As a result, this research proceeded with the integration of the trained Fuzzy-RL and GNP to test the novel idea. In the course of algorithm development, the training component of the original algorithm (Mabu, et al., 2011) was modified to improve its learning stability (the details of the changes in the algorithm can be found in Chapter 5.1), and a multi-behaviour learning problem (ball pursuit with wall avoidance) served as a

benchmark for the new algorithm presented.

Future Work

There are four objectives that can be identified for future work:

1. Test the compatibility of other algorithms, if they can work as processing nodes in the genetic network programming architecture.
2. Find a solution for optimizing the number of nodes within one GNP individual.
3. Depending on the application domain, more attributes (time allotted for node execution, mutation of processing nodes, mutation of judgment nodes, etc.) of a node can be changed during gene evolution.
4. Create a better simulation environment with a more accurate physics engine implementation and more optimized implementation of the training phase of the algorithm.
5. The fitness function could be modified to enhance the smoothness of navigation, while avoiding the walls. In this research, wall avoidance did not take into account the “smoothness” factor, but merely safety of the robot.

Appendix A.
Codes for the implementation of Fuzzy-RL

```
void runGame()
{
#region environment
        //initialize settings
        float ballX = 25;
        robot.setSpeed(0.6);
        robot.setX(110);
        robot.setY(90);
        robot.setAngle(90);
        ball.setY(90);
        ball.setX(ballX);
        ball.setSpeed(0);
        int steps = 0;
        angleFromTarget = calcAngleFromTarget(robot, worldGx, worldGy,
targetPosition);
        double wallDistance;
        preCalculateTextPositions();
        double wallAngle = getWallAngle(robot, wallDistance);
        //init s,a
        vector<double> preStateWeight;  //store all weights for each state
        //toState() return the state with the max weight
        myRL.preState = toState(angleFromTarget, wallAngle, wallDistance,
preStateWeight);
        myRL.preAction = 5;
        // keep running the program until the ESC key is pressed
        int collisionCount = 0;
        ofstream outf1;
        outf1.open("angle.txt");
        double anglesum = 0;
        while ((GetAsyncKeyState(VK_ESCAPE)) == 0)
        {
            steps++;
            setactivepage(page);
            cleardevice();
            drawPanel();
```

```
            drawGrid(robot);
            getKey(robot);  //get key strokes

            float oldangle = calcAngleFromTarget(robot, worldGx, worldGy,
targetPosition);
            preCalculateTextPositions();
            wallAngle = getWallAngle(robot, wallDistance);

            //calculate the turning angle
            double myAngle = 0;//turning angle
            double weightSUM = 0;
            for (int i = 0; i < preStateWeight.size(); i++)
            {
                //if the weight for the state is larger than zero and it's not the
state with the max weight
                //using greedy search to find a action.
                //Note: if the sum of Q value for this state is zero, it will choose
action 0, which is turn 0 degree, so no bad effect.
                if (i != myRL.preState && preStateWeight[i]>0)
                    myAngle +=
toAngle( myRL.chooseAction(i,1) )*preStateWeight[i];
                //if find the state with the max weight, the action is the previous
choosed action
                else if (i == myRL.preState)
                    myAngle += toAngle(myRL.preAction)*preStateWeight[i];

                weightSUM += preStateWeight[i];
            }
            myAngle = myAngle / weightSUM;


            float changedAngle = robot.getAngle() + myAngle;
            if (changedAngle<0)
                changedAngle = 360 + changedAngle;
            else if (changedAngle>359)
                changedAngle = changedAngle - 360;
            robot.setAngle(changedAngle);

            robot.move();   //Update Object's (x, y) position and (angle)
orientation
            robot.draw(page);     //Display Object
            if (robot.getX() <= 3.75 || robot.getX() >= 216.25 || robot.getY() <=
```

```
3.75 || robot.getY() >= 176.25)
            {
                collisionCount++;
                cout << "Wall Collision: " << collisionCount<<endl;
            }


            //get reward and update RL
            float newangle = calcAngleFromTarget(robot, worldGx, worldGy,
targetPosition);
            float newdistance = calcDistanceFromTarget(robot, worldGx, worldGy);
            double newwallDistance;
            preCalculateTextPositions();
            float newwallAngle = getWallAngle(robot,newwallDistance);
            int state = toState(newangle, newwallAngle, newwallDistance,
preStateWeight);


            double reward = 0;
            //Reward for pursuing the ball
            if (oldangle<0 && newangle<0 && newangle>oldangle)
                reward +=  30 * (1 - newangle / oldangle);
            else if (oldangle>0 && newangle > 0 && newangle < oldangle)
                reward += 30 * (1 - newangle / oldangle);
            else if (newangle < 1 && newangle >-1)
                reward += 10;

            //The RL only rewards the state which got the highest weight
            myRL.process(state, reward,0); //update the state-action space

#region DrawGraph
        }
        outf1.close();

}



//update state-action space and other process in RL
void RL::process(int state_ID, double reward,bool greedy)
{
    int newAction = chooseAction(state_ID,greedy);
```

```
    //int maxAction = findMaxQ(state_ID); //Q learning
    //if (stateSpace[state_ID].Q[newAction] == stateSpace[state_ID].Q[maxAction])
    //  maxAction = newAction;
    if (greedy == 0)
    {
        double Q = stateSpace[state_ID].Q[newAction]; //SARSA
        //double Q = stateSpace[state_ID].Q[maxAction]; //Q learning

        DELTA = reward + GAMMA*Q - stateSpace[preState].Q[preAction];
        e[preState].Q[preAction] += 1;

        for (int i = 0; i < stateSpace.size(); i++)
        {
            for (int j = 0; j < stateSpace[i].Q.size(); j++)
            {
                stateSpace[i].Q[j] = stateSpace[i].Q[j] + ALPHA*DELTA*e[i].Q[j];
                e[i].Q[j] = GAMMA*LAMDA*e[i].Q[j];
            }
            updateSum(i);
        }
    }


    preAction = newAction; //set preAction
    preState = state_ID; //set preState

}
```

Appendix B.
Codes for the implementation of GNP with trained Fuzzy-RL nodes

```
//trainning phase
void train(GNP &myGNP, int max_gen)
{
    Fuzzy FuzzyAngle; //fuzzy logic system for fitness
    FuzzyAngle.init(angle_regions);
    FuzzyAngle.initMemFunc_Angle();
    double elitefitness = -99999;
    int num_elite = myGNP.num_population - myGNP.num_crossover -
myGNP.num_mutation;
    myGNP.generation = 0;
    ifstream ifs("FuzzyRL"); //Load Fuzzy-RL instance from file
    boost::archive::text_iarchive ia(ifs);
    RL FuzzyRL_Nav;
    ia >> FuzzyRL_Nav;
    ofstream outf;
    outf.open("fitness.txt");
    while (true) //the Loop for evoluting generations
    {
        //each individual in the population do the process
        //using open mp to accelerate the FOR loop
        //each core has their own FuzzyRL and Fuzzy instance
       #pragma omp parallel for //openMP
        for (int i = num_elite; i<myGNP.num_population; i++)
        {
            trainIndiv(myGNP.population[i], FuzzyRL_Nav,FuzzyAngle);
        }//-----------------each individual in the population do the process END

        myGNP.generation++;
        myGNP.population_temp.clear();
        myGNP.sortPopulation();
        cout << myGNP.generation << "// ";
        for (int i = 0; i<num_elite; i++)
        {
            cout << myGNP.population[i].fitness << " - ";
            outf << myGNP.population[i].fitness << "\t";
```

```
        }
        outf << "\n";
        cout << endl;
        //if reach the max generation break the loop
        if (myGNP.generation>max_gen - 1)
            break;
        //Run hill-climbing algorithm every 5 generation
        if (myGNP.generation == 1 || myGNP.generation % 5 == 0)
        {
            for (int i = 0; i < num_elite; i++)
            {
                cout << "hillclimbing for " << i << " :";
                outf << "hillclimbing for " << i << "\t";
                hillclimbing(myGNP.population[i]);
                cout << myGNP.population[i].fitness << endl;
                outf << myGNP.population[i].fitness << "\n";
                //keep top n individual
                myGNP.population_temp.push_back(myGNP.population[i]);
    }
        }
        else
        for (int i = 0; i < num_elite; i++)
            //keep top n individual
            myGNP.population_temp.push_back(myGNP.population[i]);
        //Do gene operations
        myGNP.mutation();
        myGNP.crossover();
        myGNP.population = myGNP.population_temp;

    }
}




//Training for each individual which contains 4 environment setting
void trainIndiv(GNP_INDIV& indiv, RL FuzzyRL_Nav, Fuzzy FuzzyAngle)
{
    //positions of ball and robot for 4 training environment
    double RobotSetting[4][4] = { { 110, 130, 1.5, 180 }, { 170, 90, 1.5, 270 },
{ 110, 50, 1.5, 180 }, { 50, 90, 1.5, 90 } };
    double BallSetting[4][4] = { { 110, 175, 0, 0 }, { 215, 90, 0, 0 }, { 110, 5,
```

```
0, 0 }, { 5, 90, 0, 0 } };

    double fitness[4] = { 0, 0, 0, 0 };
    #pragma omp parallel for// openMP
    for (int x = 0; x < 4; x++)
    {
        double robotP[4] = { RobotSetting[x][0], RobotSetting[x][1],
RobotSetting[x][2], RobotSetting[x][3] };
        double ballP[4] = { BallSetting[x][0], BallSetting[x][1],
BallSetting[x][2], BallSetting[x][3] };
        //training for each environment setting
       fitness[x] = IndivsPosition(indiv, FuzzyRL_Nav, FuzzyAngle, robotP, ballP);

    }

    double totalfitness = 0;
    for (int i = 0; i < 4; i++)
        totalfitness += fitness[i];


    //update fitness
    indiv.cal_fitness(totalfitness);
}

//training for a certain environment setting
double IndivsPosition(GNP_INDIV indiv, RL FuzzyRL_Nav, Fuzzy FuzzyAngle, double
robotP[], double ballP[])
{
    float minDistance = 0.0;
    float tempDistance = 0.0;
    float x, y;
    int gX = 0, gY = 0; //Target in device coordinates
    float worldGx, worldGy; //Target in world coordinates
    float angleFromObstacle;
    float angleFromTarget;
    float distanceFromObstacle;
    float distanceFromTarget;
    char msg[100];

    //initial  robot and ball
    Robot robot(robotP[0], robotP[1], robotP[2], robotP[3]);
    Ball ball(ballP[0], ballP[1], ballP[2], ballP[3]);
```

```
    Position targetPosition;
    initWorld();
    //initial goal position = center of screen
    gX = getmaxx() / 2;
    gY = getmaxy() / 2;
    worldGx = float(((gX - fieldX1) * 220.0f) / abs(fieldX2 - fieldX1));
    worldGy = worldBoundary.y1 - (float(((gY - fieldY1) * 180.0f) / abs(fieldY2 -
fieldY1)));
    worldGx = ball.getX();
    worldGy = ball.getY();
    calcAngleFromTarget(robot, worldGx, worldGy, targetPosition);
    preCalculateTextPositions();
    double fitness = 0;

    //init for simulation ----------------------------------------------
    int time_step = 0; //accumulate time steps for one individual in the simulation
environment
    FuzzyRL_Nav.preAction = 0;
    FuzzyRL_Nav.preState = toState(calcAngleFromTarget(robot, worldGx, worldGy,
targetPosition), 0, 0, FuzzyRL_Nav.preStateWeight, FuzzyAngle);
    indiv.Time_left = 0;
    int current_node = 0;

    //The simulation loop for one individual starts-----------------------------
    //set some conditions to break the loop
    while (time_step<200)
    {
        time_step++;
        if (indiv.Time_left == 0 && indiv.Nodes[current_node].TD != 0)
        {
            indiv.Time_left = indiv.Nodes[indiv.next_node].TD;
            current_node = indiv.next_node;
        }
        //current node type
        int node_type = indiv.Nodes[current_node].type;

        //input values-------------------------------------------
        angleFromTarget = calcAngleFromTarget(robot, worldGx, worldGy,
targetPosition);
        distanceFromTarget = calcDistanceFromTarget(robot, worldGx, worldGy);
        float oldangle = angleFromTarget;
        float olddistance = distanceFromTarget;
```

```
        //input selector----------------------------------------
        double input1 = 0, input2 = 0, input3 = 0;
        switch (node_type)
        {
        case 1:
            input1 = angleFromTarget;
            break;
        case 2:
            input1 = distanceFromTarget; input2 = 0; input3 = 0;
            break;
        case 3:
            input1 = angleFromTarget;
            break;
        case 4:
            input1 = distanceFromTarget; input2 = 0; input3 = 0;
            break;

        }
        int preNode = current_node;
        //execute node and nextnode becomes current excuted node
        indiv.toNextNode(ExeNode(robot, FuzzyRL_Nav, node_type, input1, input2,
input3), current_node);
        indiv.Time_left--;
        if (indiv.Time_left == -1)
        {
            indiv.Time_left = indiv.Nodes[indiv.next_node].TD;
            current_node = indiv.next_node;
            continue;
        }

        //Update
Environment-------------------------------------------------------------------
        robot.move();   //Update Object's (x, y) position and (angle) orientation

        if (robotTrails.size() > 250){
            robotTrails.clear();
        }

        if (ballTrails.size() > 350){
            ballTrails.clear();
        }
        ball.move(robot.getX(), robot.getY(), robot.getAngle(),
```

```
robot.getSpeed());
        worldGx = ball.getX();
        worldGy = ball.getY();

        //Calculate Reward-------------------------------
        float newangle = calcAngleFromTarget(robot, worldGx, worldGy,
targetPosition);
        float newdistance = calcDistanceFromTarget(robot, worldGx, worldGy);
        double newwallDistance;
        float newwallAngle = getWallAngle(robot, newwallDistance);
        double reward = 0;//the Fuzzry-RL is using greedy search when training, so
no need for reward

        //PostProcess--------------------
        PostProcess(FuzzyRL_Nav, node_type, newangle, reward, 0, FuzzyAngle);//no
need reward now,only greedy search

        //fitness
        if (indiv.Nodes[preNode].TD != 0)
        {
            //Calculate fitness for wall avoidence and ball pursuing
            if (newwallDistance<20)
            {
                //only if the robot is heading away from the ball gives extra reward
                and reward for ball pursuing
                if (newwallAngle <= -90 || newwallAngle >= 90)
                {
                    fitness += 20;
                    fitness += ballReward.output(newangle, newdistance);
                }
            }
            else
                fitness += ballReward.output(newangle, newdistance);

            //Calculate fitness for speed control
            if (newdistance < 10 && robot.getSpeed() == 0.5)
                fitness += 6;
            else if (newdistance >= 10 && newdistance < 20 && robot.getSpeed() ==
            1.0)
                fitness += 6;
            else if (newdistance >= 20 && newdistance < 50 && robot.getSpeed() ==
            1.5)
```

```
                fitness += 6;
            else if (newdistance >= 50 && robot.getSpeed() == 2.0)
                fitness += 6;

            //Some punishment for bad situations
            if (newwallDistance < 15)
                fitness -= 50;
            if (distanceFromTarget - newdistance < 0)
                fitness -= 15;
        }


    }//-------The simulation loop END
    //after 200 time-steps, closer to the ball gets higher fitness
    fitness += 500 - calcDistanceFromTarget(robot, worldGx, worldGy) * 10;

    return fitness;
}


//hill-clibing algorithm for one individual
void hillclimbing(GNP_INDIV& myBest)
{
    Fuzzy FuzzyAngle;
    FuzzyAngle.init(angle_regions);
    FuzzyAngle.initMemFunc_Angle();
    RL FuzzyRL_Nav;
    ifstream ifs("FuzzyRL");
    boost::archive::text_iarchive ia(ifs);
    ia >> FuzzyRL_Nav;
    double RobotSetting[4][4] = { { 110, 130, 1.5, 180 }, { 170, 90, 1.5, 270 },
{ 110, 50, 1.5, 180 }, { 50, 90, 1.5, 90 } };
    double BallSetting[4][4] = { { 110, 175, 0, 0 }, { 215, 90, 0, 0 }, { 110, 5,
0, 0 }, { 5, 90, 0, 0 } };

    //change 1 connection each iteration
    double maxfitness = myBest.fitness;
    int i = myrand() % myBest.num_nodes;
    for (int ii = 0; ii < myBest.num_nodes;ii++)
    {
        for (int j = 0; j < myBest.Nodes[i].C.size(); j++)
        {
```

```
            int bestC = myBest.Nodes[i].C[j];
            for (int n = 1; n < myBest.num_nodes; n++)
            {
                myBest.Nodes[i].C[j] = n;
                double fitness[4] = { 0, 0, 0, 0 };
                #pragma omp parallel for
                for (int x = 0; x < 4; x++)
                {
                    double robotP[4] = { RobotSetting[x][0], RobotSetting[x][1],
                    RobotSetting[x][2], RobotSetting[x][3] };
                    double ballP[4] = { BallSetting[x][0], BallSetting[x][1],
                    BallSetting[x][2], BallSetting[x][3] };

                    fitness[x] = IndivsPosition(myBest, FuzzyRL_Nav, FuzzyAngle,
                    robotP, ballP);
                }
                double totalfitness = 0;
                for (int i = 0; i < 4; i++)
                    totalfitness += fitness[i];
                if (totalfitness >= maxfitness)
                {
                    maxfitness = totalfitness;
                    bestC = n;
                }
            }
            myBest.Nodes[i].C[j] = bestC;
            myBest.cal_fitness(maxfitness);

        }

        i++;
        if (i == myBest.Nodes.size())
            i = 0;
    }
}
```

Reference

Barto, A.G., Sutton, R.S. and Anderson, C.W. 1983. Neuronlike adaptive elements that can solve difficult learning control problems. *Man and Cybernetics.* Sept.-Oct. 1983, Vols. SMC-13, 5, pp. 834-846.

Bellman, R. 1957. A Markovian Decision Process. 1957, Vol. 6.

Berenji, H.R. and Khedkar, P. 1992. Learning and tuning fuzzy logic controllers through reinforcements. *Neural Networks, IEEE Transactions.* Sep. 1992, Vol. 3, 5, pp. 724-740.

Berenji, Hamid R. 1992. A reinforcement learning—based architecture for fuzzy logic control. *International Journal of Approximate Reasoning.* February 1992, Vol. 6, 2, pp. 267-292.

Bonarini, Andrea, et al. 2009. Reinforcement distribution in fuzzy Q-learning, Fuzzy Sets and Systems. 10, 16 May 2009, Vol. 160, pp. 1420-1443.

Faria, Gedson and Romero, R. 2000. Incorporating fuzzy logic to reinforcement learning [mobile robot navigation]. *Fuzzy Systems, 2000. FUZZ IEEE 2000. The Ninth IEEE International Conference.* 2000, Vol. 2, pp. 847-852.

Katagiri, H., Hirasawa, K. and J. Hu. 2000. Genetic Network Programming Application to Intelligent Agents. 2000.

Kormushev, P., Calinon, S. and Caldwell, D.G. 2013. Reinforcement Learning in Robotics: Applications and Real-World Challenges. *Robotics.* 2013, Vol. 2, pp. 122-148.

Li, Xianneng, Mabu, S. and Hirasawa, K. 2014. A Novel Graph-Based Estimation of the Distribution Algorithm and its Extension Using Reinforcement Learning. 2014, Vol. 18, 1, pp. 98-113.

Mabu, Shingo and Kotaro Hirasawa. 2011. Fuzzy Genetic Network Programming with Reinforcement Learning for Mobile Robot Navigation. 2011.

Mabu, Shingo, Kotaro Hirasawa and Jinglu Hu. 2007. A Graph-Based Evolutionary Algorithm: Genetic Network Programming (GNP) and Its Extension Using Reinforcement Learning. 2007, Vol. 15, 3, pp. 369-398.

Metropolis, Nicholas and S. Ulam. 1949. The Monte Carlo Method. *Journal of the American Statistical Association.* 247, Sep. 1949, Vol. 44.

Mukherjee, S., et al. 2011. Reinforcement Learning Approach to AIBO Robot's Decision Making Process in Robosoccer's Goal Keeper Problem. *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD).* 6-8 July 2011, pp. 24-30.

Pitoyo, Hartono and Kakita, Sachiko. 2009. Fast reinforcement learning for simple

physical robots. *Memetic Comp.* 1, 2009, pp. 305–313.

Poole, David and Alan Mackworth. 2010. *Artificial Intelligence: Foundations of Computational Agents.* s.l. : Cambridge University Press, 2010.

Reyes, N. H., et al. 2013. Real-Time Fuzzy Logic-based Hybrid Robot Path-Planning Strategies for a Dynamic Environment. [book auth.] & J. Zurada B. Igelnik. *Efficiency and Scalability Methods for Computational Intellect.* s.l. : Hershey, 2013, pp. 115-141.

Ross, Timothy J., Jane M. Booker and W. Jerry Parkinson. 2002. *Fuzzy Logic and Probability Applications: Bridging the Gap.* 2002. pp. 29-31.

Rummery and Niranjan. 1994. *Online Q-Learning using Connectionist Systems.* 1994.

Sutton, Richard S. and Andrew G. Barto. 2012. *Reinforcement Learning: An Introduction.* s.l. : The MIT Press, 2012.

Watkins, C. and P. Dayan. 1992. Q-learning. *Machine Learning.* 1992, Vol. 8, pp. 279–292.

Zadeh, L.A. 1965. Fuzzy sets. *Information and Control.* 3, 1965, Vol. 8, pp. 338–353.