Fault Recovery in Swarm Robotics Systems using Learning Algorithms

A thesis presented for the degree of

Doctor of Philosophy

Electronic Engineering

University of York

Oyinlola Oladiran

November, 2019

Abstract

When faults occur in swarm robotic systems they can have a detrimental effect on collective behaviours, to the point that failed individuals may jeopardise the swarm's ability to complete its task. Although fault tolerance is a desirable property of swarm robotic systems, fault recovery mechanisms have not yet been thoroughly explored. Individual robots may suffer a variety of faults, which will affect collective behaviours in different ways, therefore a recovery process is required that can cope with many different failure scenarios. In this thesis, we propose a novel approach for fault recovery in robot swarms that uses Reinforcement Learning and Self-Organising Maps to select the most appropriate recovery strategy for any given scenario. The learning process is evaluated in both centralised and distributed settings. Additionally, we experimentally evaluate the performance of this approach in comparison to random selection of fault recovery strategies, using simulated collective phototaxis, aggregation and foraging tasks as case studies. Our results show that this machine learning approach outperforms random selection, and allows swarm robotic systems to recover from faults that would otherwise prevent the swarm from completing its mission. This work builds upon existing research in fault detection and diagnosis in robot swarms, with the aim of creating a fully fault-tolerant swarm capable of long-term autonomy.

Contents

I Introduction and Literature Review 14

1 Introduction 15
1.1 Hypothesis and Goals . . . 17
1.2 Contribution . . . 18
1.3 Thesis Outline . . . 20

2 Literature Review 21
2.1 Introduction to Swarm Intelligence . . . 21
2.2 Swarm Taxis Algorithms . . . 24
2.2.1 The α algorithm . . . 25
2.2.2 The β algorithm . . . 25
2.2.3 The ω algorithm . . . 26
2.3 Fault Tolerance . . . 26
2.4 Artificial Immune Systems (AIS) . . . 29
2.5 Fault detection in Multi-robot systems . . . 30
2.6 Fault detection in Swarm Robotic Systems . . . 32
2.7 Fault Recovery . . . 35
2.7.1 Single-robot systems . . . 35
2.7.2 Multi-robot systems . . . 35
2.8 Evolutionary Robotics . . . 37
2.8.1 Random Optimisation . . . 37
2.8.2 Online Evolution . . . 39
2.8.3 Offline Evolution . . . 40
2.8.4 Evolutionary Swarm Robotics . . . 41
2.8.5 Issues with Evolutionary Robotics . . . 42
2.8.6 Other methods of evaluating Candidate Solutions . . . 46
2.8.7 Fitness-based Evolution . . . 50

3 Preliminary Results **54**
3.1 Methodology . . . 54
3.1.1 The Collective Phototaxis Algorithm. . . . 54
3.1.2 The Evolutionary Algorithm . . . 58
3.2 Results . . . 60

II Learning Approaches and their Implementations: Fault Recovery Solution (Centralised and Distributed) **65**

4 Learning Techniques **67**
4.1 Reinforcement Learning . . . 67
4.2 Deep Learning . . . 69
4.3 Self-Organising Maps . . . 73

5 Centralised Approach to Fault Recovert **75**
5.1 Empty Environment . . . 75
5.1.1 Learning the Best Recovery Strategy . . . 77
5.2 Experimental Setup . . . 84
5.2.1 Results . . . 85
5.3 Inclusion of Obstacles . . . 94
5.3.1 States . . . 95
5.3.2 Updated Rewards . . . 96
5.3.3 Experimental Setup . . . 97
5.3.4 Results . . . 97

6 Distributed Learning Approach **107**
6.1 Distributed Learning . . . 107
6.1.1 Distributed Deep Learning . . . 110
6.1.2 General Distributed Machine Learning Limitations . . . 111
6.1.3 Multi-Agent Distributed Learning in Robotics . . . 112
6.1.4 Distributed Reinforcement Learning . . . 114
6.2 Learning Approach . . . 115
6.3 Empty Environment . . . 117
6.3.1 States and Actions . . . 120
6.3.2 Rewards . . . 122
6.3.3 Algorithm . . . 123
6.3.4 Experimental Setup . . . 125
6.3.5 Results . . . 129

6.3.6 Inclusion of Obstacles . . . 139
6.3.7 States and Updated Rewards . . . 140
6.3.8 Learning Setup . . . 142
6.3.9 Results . . . 144

7 Extension of Experiments 155
7.1 Introduction . . . 155
7.2 Experimental Setup . . . 157
7.3 Experiment Overview . . . 157
7.4 Centralised Approach . . . 162
7.4.1 Collective Phototaxis . . . 162
7.4.2 Aggregation . . . 169
7.4.3 Foraging . . . 173
7.5 Distributed Approach . . . 181
7.5.1 Collective Phototaxis . . . 181
7.5.2 Aggregation . . . 188
7.5.3 Foraging . . . 192

III Conclusion and Future Work 201

8 Conclusion 202

9 Future Work 206

Bibliography 207

IV Appendix 218

10 20 Robots in the Swarm 220

11 40 Robots in the Swarm 226

List of Tables

3.1 Genome Ranges where HARD_TURN angle is a sharp turn, SOFT_TURN is a softer turn and NO_TURN means the robot goes straight. Target Distance, Gain and Exponent are parameters in the Lennard Jones Potential equation that is used to calculate the level of interaction between the robots in the swarm 59
3.2 GA Parameters 60
3.3 Rank Sum and A Test(Between the 0 fault evolved controller and 2 faults evolved controller), Median and InterQuartile Range (IQR) of the number of robots that made it to the beacon 62
3.4 Rank Sum and A Test(Between the 0 fault evolved controller and 5 faults evolved controller), Median and InterQuartile Range (IQR) of the number of robots that made it to the beacon 63
3.5 Rank Sum and A Test (Between the 0 fault evolved controller and 9 faults evolved controller), Median and InterQuartile Range (IQR) of the number of robots that made it to the beacon 63

5.1 The table describes the parameters used for the learning process. All Q-values are set to zero and the weights if the input SOM are initialised randomly in the range [0,1]. The set of parameters described above were empirically derived. The value σ represents the width of the neighbourhood which shrinks over time, σ_o represents the initial width of the neighbourhood, t represents the current iteration loop, λ represents the time constant. 81
5.2 The table describes the percentage of the test input that is selected during the testing phase of the RL + SOM and the recovery type selected. The recovery type column represents what predefined behaviour was chosen for the 50 sample inputs. The percentage column represents what percentage of the test inputs selects the corresponding recovery type. 88

5.3 Rank Sum and A Test for the Collective Phototaxis Experiments . . 89
5.4 Rank Sum and A Test for the Aggregation Experiments 91
5.5 Rank Sum and A Test for the Foraging Experiments 93
5.6 The table describes the parameters used for the learning process. All Q-values are set to zero and the weights if the input SOM are initialised randomly in the range [0,1]. The set of parameters described above were empirically derived. The value σ represents the width of the neighbourhood which shrinks over time, σ_o represents the initial width of the neighbourhood, t represents the current iteration loop, λ represents the space constant (distance measure). 99
5.7 The table describes the percentage of the test input that is selected during the testing phase of the RL + SOM and the recovery type selected. The recovery type column represents what predefined behaviour was chosen for the 50 sample inputs. The percentage column represents what percentage of the test inputs selects the corresponding recovery type. 100

6.1 The table describes the parameters used for the learning process. All Q-values are set to zero and the weights if the input SOM are initialised randomly in the range [0,1]. The set of parameters described above were empirically derived. The value σ represents the width of the neighbourhood which shrinks over time, σ_o represents the initial width of the neighbourhood, t represents the current iteration loop, λ represents the time constant. 125
6.2 This table describes the information on what in transmitted and broadcasted between the robots during the recovery process 127
6.3 The table describes the percentage of the test input that is selected during the testing phase of the RL + SOM and the recovery type selected. The recovery type column represents what predefined behaviour was chosen for the 50 sample inputs. The percentage column represents what percentage of the test inputs selects the corresponding recovery type. 133

6.4 The table describes the parameters used for the learning process. All Q-values are set to zero and the weights if the input SOM are initialised randomly in the range [0,1]. The set of parameters described above were empirically derived. The value σ represents the width of the neighbourhood which shrinks over time, σ_o represents the initial width of the neighbourhood, t represents the current iteration loop, λ represents the time constant. 142

6.5 The table describes the percentage of the test input that is selected during the testing phase of the RL + SOM and the recovery type selected. The recovery type column represents what predefined behaviour was chosen for the 50 sample inputs. The percentage column represents what percentage of the test inputs selects the corresponding recovery type. 148

7.1 This table gives an overview of all the results that are all the experiments that are in this chapter. 162

List of Figures

4.1 Playing Atari Games using Deep Q-Learning [1] 70
4.2 Two-dimensional Representation of Self Organising Maps [1] 74

5.1 This figure describes the three-dimensional representation of the Q-table for this specific reinforcement learning technique. As can be seen, it is different from the traditional representation of the Q-table which is represented as a two dimensional table. 76
5.2 This figure represents one of the tasks, collective phototaxis. This is the ability for a robot swarm to sense a light source in the environment and collectively move towards it. 77
5.3 This figure represents the second task that is tested, aggregation. This is the ability for the robots in the swarm to cluster together no matter how spread out in the environment that they are. 78
5.4 This figure represents the second of the task, foraging. This is the ability for a robot swarm to 'forage' or to search for items around an environment and bring them back to a 'base' or 'nest'. 79
5.5 Representation of how the self organising map works with reinforcement learning in our approach 84
5.6 Rewards Convergence 87
5.7 Results for collective phototaxis: Motor failures 88
5.8 Results for collective phototaxis: Communication sensor failures . . . 89
5.9 Results for aggregation when the range and bearing sensors (Communication sensors) on the robots are faulty. 91
5.10 Results for foraging when light sensor failures are injected: Number of robots that reached the light source. 92
5.11 Results for foraging when light sensor failures are injected: Number of items collected. 93
5.12 Rewards Convergence for New Experimental Setup. 99
5.13 Results for collective phototaxis: Communication sensor failures . . . 101

5.14 Results for collective phototaxis: Motor failure 102
5.15 Results for aggregation when the range and bearing sensors (Communication sensors) on the robots are faulty. 103
5.16 Results for foraging when light sensor failures are injected: Number of robots that reached the light source. 104
5.17 Results for foraging when light sensor failures are injected: Number of items collected. 105

6.1 This figure represents the properties and features of a foot-bot. Taken from [2] 119
6.2 This figure represents the properties and features of an e-puck. Taken from [3] 120
6.3 Rewards Convergence for each Robot 133
6.4 Collective Phototaxis: Motor failures 134
6.5 Collective Phototaxis: Communication sensor failures 135
6.6 Results for aggregation when the range and bearing sensors (Communication sensors) on the robots are faulty. 136
6.7 Results for foraging when light sensor failures are injected: Number of robots that reach the nest at the end of the task. 137
6.8 Results for foraging when light sensor failures are injected: Number of items collected in the environment 138
6.9 Rewards Convergence for each Robot 148
6.10 Collective Phototaxis: Motor failures 149
6.11 Collective Phototaxis: Communication sensor failures 150
6.12 Results for aggregation when the range and bearing sensors (Communication sensors) on the robots are faulty. 151
6.13 Results for foraging when light sensor failures are injected: Number of robots that reach the nest at the end of the task. 152
6.14 Results for foraging when light sensor failures are injected: Number of items collected in the environment 153

7.1 Collective Phototaxis: Motor failures 163
7.2 Collective Phototaxis: Complete Sensor Failures 164
7.3 Collective Phototaxis: Multiple Failures 165
7.4 The plots below describe Collective Phototaxis for Multiple Failures when injected with 4,6 and 8 faults 166
7.5 Aggregation: Complete Sensor Failures 169
7.6 The plots below describe Aggregation for Multiple Failures when injected with 4,6 and 8 faults 170

7.7 Foraging: Light Sensor Failure (Number of Robots at Base) 173
7.8 Foraging: Light Sensor Failure (Food Items Collected) 174
7.9 The plots below describe Foraging for Light Sensor Failure (Number of Robots at Base) when injected with 4,6 and 8 faults 175
7.10 The plots below describe Foraging for Light Sensor Failure (Food Items Collected) when injected with 4,6 and 8 faults 178
7.11 Collective Phototaxis: Motor failures 182
7.12 Collective Phototaxis: Complete Sensor Failures 183
7.13 Collective Phototaxis: Multiple Failures 184
7.14 The Plots below describe Collective Phototaxis for Multiple Failures when injected with 4,6 and 8 faults 185
7.15 Aggregation: Complete Sensor Failures 188
7.16 The Plots below describe Aggregation for Multiple Failures when injected with 4,6 and 8 faults 189
7.17 Foraging: Light Sensor Failure (Number of Robots at Base) 192
7.18 Foraging: Light Sensor Failure (Food Items Collected) 193
7.19 The Plots below describe Foraging for Light Sensor Failure (Number of Robots at Base) when injected with 4,6 and 8 faults 194
7.20 The plots below describe Foraging for Light Sensor Failure (Food Items Collected) when injected with 4,6 and 8 faults 197

10.1 Collective Phototaxis: Motor failures 221
10.2 Collective Phototaxis: Communication sensor failures 222
10.3 Aggregation: Communication sensor failures 223
10.4 Foraging: Light Sensor (Number of Food Collected) 224
10.5 Foraging: Light Sensor (Number of Robots at Nest) 225

11.1 Collective Phototaxis: Motor failures 227
11.2 Collective Phototaxis: Communication sensor failures 228
11.3 Aggregation: Communication sensor failures 229
11.4 Foraging: Light Sensor (Number of Food Collected) 230
11.5 Foraging: Light Sensor (Number of Robots at Nest) 231

Acknowledgements

I would like to thank my amazing supervisors, Jon Timmis, Alan Millard, Martin Trefzer and Danesh Tarapore for their support and understanding throughout my PhD process especially towards the end of my PhD when things became more difficult. I appreciate all your help, your feedback, your ideas, your criticisms; all of these contributed to my thesis and also as a person. I would like to thank my fellow colleagues, for those days where i needed help with some programming or with the simulator, and understanding how it works better. I would also like to thank my parents and my sisters for their support both monetary and otherwise; without them i would not have been able to start and end this PhD journey and i would forever be extremely grateful to them. Finally, i would like to thank my friends for their support, the calls, the check-up sessions to make sure i am okay; i appreciate all the love and support. Thank you all for everything.

Declaration

I declare that this thesis is a presentation of original work and I am the sole author. This work has not previously been presented for an award at this, or any other, University. All sources are acknowledged as References.

Part I

Introduction and Literature Review

Chapter 1

Introduction

This thesis analyses fault tolerance in robotic systems, especially swarm robotic systems, discusses possible evolutionary techniques that can be used in achieving a fault tolerant system while also introducing learning techniques used in conjunction with predefined behaviour to achieve a fault tolerant system that is able to recover autonomously from faults and continue whatever task it has been assigned.

Some approaches to swarm robotics are inspired by the self-organised behaviours of social insects, exploiting simple rules and local interactions to achieve robust, scalable, and flexible collective behaviours for the coordination of large numbers of robots [4]. Robustness signifies a swarm's ability to recover from faults that could result from the individual robots in the swarm [5] while flexibility signifies that the individual robot in the swarm should be able to coordinate their behaviours to adjust to a difference in the nature of the task [6]. A swarm is said to scalable when the system functions with varying group sizes without adversely affecting the behaviour of the swarm [6].

Swarm robotics can be described as the study of how a large number of physically simple robots can be designed in a way whereby a collective behaviour emerges from local interaction and communication amongst the robots and also between the robots and their environment [7]. There are a number of qualities that are specific to swarm robotics and are used to differentiate it from the multi-robots research. Although these qualities are not rigid, they are useful in describing aspects of swarm robotics [7]:

1. The individuals in the swarm are described as being autonomous. They must have a physical embodiment and physically interact with their environment and be autonomous. There are some systems that the individuals can attach to themselves and detach from each other as well. They can still be considered

as swarm systems only if there is decentralised control present.

2. Swarm robots are classified as consisting of a large number of individuals. Small number of robots that are not scalable do not fall within the classification of swarm robots. Swarm sizes typically start from about 10 individuals.

3. Regardless of how large a robotic system is, the number of individual homogeneous robots should be large. Robotic systems that are more heterogeneous are classified as being less 'swarm robotic'. The reason for this that if the robot-robot interaction is varying, this has a significant effect on the behaviour of the robots. Also, if the behaviours are random, it is not possible to achieve similar results when the experiment is run under the same conditions.

4. The individual robot in a swarm can be described as being inefficient, so that they cannot accomplish the task assigned on its own. The reason for this is to enable the individuals to cooperate to accomplish the task or by utilising a group of these robots, the robustness is improved when accomplishing the task.

5. The robots used in a swarm should only have local communication and sensing capabilities. The allows the robots to be have scalable coordination mechanisms.

In the field of swarm robotics, it was initially believed that swarm robotic systems were robust to failure. However, analysis done by Bjerknes et al. [8] demonstrated that partially failed individuals can result in detrimental behaviour in swarm robotic systems. It is desired that swarm robotic systems should be fault tolerant. Fault tolerance is the ability of a system to continue operating even when in the presence of failures [7]. It can be divided into a three step process: fault detection, diagnosis and fault recovery. When a fault is detected, the location of the fault is isolated (diagnosis) to allow for a form recovery process to remove the fault or significantly reduce the effect of the fault so as to allow the robot to continue in its task [9]. This thesis focuses on fault recovery in swarm robotic systems while assuming that the system is capable of detecting and diagnosing faults.

To achieve we propose utilising learning approaches with predefined behaviours to design a partially 'adaptive' fault recovery swarm robotic system. Learning techniques can be used to generalise solutions to fit problems that are introduced to the swarm.

In preliminary experiments, we employ genetic algorithms for the evolution of fault tolerance in the presence of complete motor failure. Specifically, we explore the anchoring effect in collective phototaxis in the swarm and evolve controllers that

are tolerant to certain failures, thus reducing the overall effect of anchoring of the swarm. We focus on the evolution of specific parameters of the controller, using a simple fitness function of minimising the distance between the robots and the beacon. These experiments are discussed and expanded in the next chapter and are done to test a possible approach towards solving the fault recovery problem presented in this thesis. Initially, genetic algorithms were considered as a possible approach for developing fault recovery solutions due to its ability to develop new solutions that might not have been considered by the developer; if certain parameters of the robots state or function can be evolved, can we develop some sort of fault tolerant system initially before moving to a more active fault recovery approach. This is described more in detail in chapter 2.10.

Collective phototaxis is a behaviour involving the swarm orienting and moving towards a source of light, such as a beacon. The ω-algorithm, proposed by Bjerknes et al. [10], provides aggregation of robotic swarms and emergent swarm taxis, through the use of simple attraction and repulsion mechanisms, taxis was towards an infra-red beacon. However, the authors observed that partial failure types had a detrimental effect on the ability of the swarm to achieve taxis. Under certain conditions, an "anchoring effect" was observed, whereby fully operational robots would be hindered by failed robots preventing them from reaching the light source [10]. This scenario forms a case study and initial work that is done for this PhD. This problem mentioned signifies one, in a number of possible faults that can occur in a swarm robotic system which effects depend on what task the swarm is assigned to. The main idea is to have an adaptable fault recovery system that takes into consideration, most faults and swarm behaviours.

1.1 Hypothesis and Goals

Based on the goal of the thesis to develop an adaptive fault recovery approach for swarm robotic systems based on predefined recovery behaviour selection, the following hypothesis can be formulated:

> `‘Can a combination of Reinforcement Learning and Self-Organising Maps used in conjunction with predefined recovery behaviours aid in improving fault tolerance in swarm robotic systems?’`

Following on this hypothesis, the following goals can be identified:

- To develop a fault recovery approach that is based on predefined behaviours to be used as recovery solutions
- To apply reinforcement learning techniques and self organising maps to implement the recovery approach
- To extend the testing and testing between two general approaches: centralised and decentralised
- Test learning approach in extended fault scenarios between different tasks, single and multiple faults

The methodology used in the thesis is described as follows:

The method proposed here allows robots to learn how to select the most appropriate recovery strategy for any given system state or task. We refer to our approach as 'pre-fault learning'. This involves learning predefined recovery mechanisms for different possible swarm states before a fault occurs. The swarm's state is defined by the distances of the nearest three robots closest to the faulty robot, their energy levels, the level of importance of the faulty robot, how busy the nearest three robots are, and the distance of the faulty robot to a repair station [d_1, d_2, d_3, b_1, b_2, b_3, p_1, p_2, p_3, I, d_{rs}]. These values represent the input of the learning algorithms that is used to learn the best action for any possible state. We use two learning algorithms that work well together: Reinforcement learning and Self Organising Maps and these approaches are tested in both centralised and distributed settings.

Reinforcement learning in robotic systems is typically used in path or task planning [11], [12], [13]. For each possible state, an action is chosen and a reward is allocated based on how successful that action is for that state. The state, action and corresponding reward is stored in a lookup table. However, robotic systems exist in continuous spaces, while reinforcement learning is traditionally implemented for discrete spaces. It is thereby necessary to extend the present reinforcement learning architecture to allow for learning to be done on a continuous state variables. Our approach uses self-organising maps to solve this problem by clustering similar states to produce a discrete set.

1.2 Contribution

In this paper, we assume that the swarm robotic system is capable of detecting and diagnosing faults, and instead propose a novel approach to fault recovery that

involves intelligent selection of predefined recovery strategies. These recovery strategies cover faults enumerated by [5] that commonly occur in swarm robots. From the point at which a fault has been detected and diagnosed, the swarm must decide upon an appropriate recovery strategy. We assume that each robot has the ability to repair other robots in the swarm, therefore the problem reduces to choosing which non-faulty robot(s) should be recruited to repair the faulty robot, and which predefined behaviour is most appropriate given the current scenario. It should be noted that this thesis is about the mechanisms for efficiently guiding a swarm of robots to make decisions about whether to repair broken robots or not, but it would NOT be about the detailed mechatronics of effecting the repair itself. The most appropriate recovery strategy will depend on a number of factors, such as proximity to the faulty robot or remaining battery power, thus some method of assessing the quality of a strategy and its future effect on the swarm is required. We present a solution to this problem that uses machine learning techniques to inform decisions at run-time based on the results of offline training. It should be noted that this idea is intended as a proof-of-principle that demonstrates the value of reinforcement learning for the selection of fault recovery strategies. There are two architectures: the centralised architecture where a global observer (computer) to calculate the system state and select the most appropriate recovery action and the distributed or decentralised architecture where each robot in the swarm makes decisions based only on local information.

There are additional contributions that this thesis presents in fault recovery for swarm robotic systems

- A library of fault recovery behaviours in swarm robotic systems
- Implementing a simulation environment for testing the approach in ARGoS (robot simulator)
- Develop a Reinforcement learning approach for fault recovery in swarm robotics
- Developing and implementing the fault recovery approach in centralised settings for different tasks: aggregation, collective phototaxis and foraging
- Developing and implementing the fault recovery approach in decentralised (distributed) settings for different tasks
- Developing the fault recovery approach across a variety of fault scenarios across different tasks

1.3 Thesis Outline

This thesis consists of three parts. Part 1 is divided into three chapters: the introduction, the literature review (discussion on related work on fault tolerance, fault detection and fault recovery whilst also discussing possible evolutionary techniques that can be linked to learning that can be used in the fault recovery approach that is being discussed in this thesis) and the preliminary results. Part 2 details the learning approach which is divided into multiple chapters: the fault recovery approach done in centralised settings, distributed settings and also an extension of the experiments done in both the centralised and distributed settings. Finally, Part 3 comprises of the final two chapters, the conclusion and future work.

Chapter 2

Literature Review

This literature review discusses a range of topics from swarm literature and algorithms to fault tolerant systems including fault detection, diagnosis and recovery for single, multi and swarm robot systems. Additionally, possible solutions to solving the fault recovery problem are introduced; an overview of artificial immune systems and evolutionary robotics which introduces the different evolutionary computation techniques that can be considered for the solution.

2.1 Introduction to Swarm Intelligence

Swarm intelligence is an artificial intelligence (AI) method that revolves around the study of swarm behaviour in decentralised self-organised systems [14]. Swarm robotics is the application of the principles of swarm intelligence in the control of a collective of robots. There are main concepts that involve the swarm intelligence field such as decentralisation, stigmergy, self-organisation and emergence. Stigmergy involves communication (direct or indirect) by the environment. Ants communicate to other ants, the quality of a path by marking with pheromones so that a positive feedback mechanism ends eventually in most insects following the best path [15]. There is no centralised control system controlling the individual robot, however the interactions between the robots should lead to the emergence of global behaviour which is a characteristic of swarm intelligence. We can classify these emergent properties into four components: coordination, cooperation, deliberation and collaboration [14].

Grasse at al. [16] describes stigmergy in building activity in social insects. They show that the individual's coordination and building activities depend mainly in the structure of the nest, that is, the environment rather than the individuals themselves. Every instance a robot performs an action, the local environment changes and this

action influences the actions of the overall swarm [14].

Another concept of self-organisation is a set of mechanisms where behaviours can be observed at the global level of a system due to interactions among the lower-level components without these components being explicitly coded. It relies on four components [14]:

- Positive feedback which is derived from behavioural rules that promotes creation of structures, that is, recruitment and reinforcement.
- Negative feedback which counters positive feedback which stabilises the collective swarm.
- Amplification of fluctuations by positive feedback. Social insects frequently perform random actions which is very important as this helps the swarm discover new solutions.
- Multiple direct or stigmergic interactions among the individuals to produce random outcomes that could produce enduring results.

There have been various attempts to characterise the properties of self-organisation, specifically [14]:

- Self-organised systems are dynamic in that when promoting creation of structures, permanent interactions between the individuals and their environment is required.
- Self-organised systems exhibit emergent properties. The system display properties that are more complex than each individual.
- The interactions between the individuals and local environment lead to bifurcations. "A bifurcation is the appearance of new stable solutions when some of the system parameters change".
- Self-organised systems can be multi-stable. This means that for a set of parameters, initial conditions and random fluctuations, there can be different stable states.

There are collective behaviours of social insects and they can be broken down into four types of tasks:

- Coordination: This is the appropriate organisation in space and time of the tasks needed to solve a specific problem.

- Cooperation: This happens when the individuals work together to achieve a task that an individual cannot solve on their own. The swarm must "cooperate" to succeed.
- Deliberation: This refers to mechanisms that allow the swarm to make decisions when faced with choices.
- Collaboration: This means that different activities are performed at the same time by groups of specialised individuals.

The reason behind defining the intelligence of a swarm is because we are pointing out a swarm is not a trivial system or a system that could easily be mapped into other well known systems. Swarm intelligence is an emergent property and it shows that a swarm system could not only be equally capable of doing what single robots do, but more.

Swarm robotics has been described as a novel approach to the coordination of large numbers of robots. It can also be described as the study of how large numbers of relatively simple physically embodied agents can be designed such that a desired collective behaviour emerges from the local interactions among agents and between the agents and the environment [4]. The main inspiration for swarm robotics comes from observing social animals such as ants, bees, birds. The reason behind the inspiration is because these animals show a sort of swarm intelligence; they appear to be robust, scalable and flexible. Robustness is the ability of the swarm to cope with the loss of individuals. In social animals, robustness is promoted by redundancy and the absence of a leader. Scalability is the ability to perform well with different group sizes. The performance of the swarm should not drastically change after the introduction or removal of individuals from the swarm. Scalability is promoted by local sensing and communication. Flexibility is the ability of the swarm to cope with a different environments and tasks. In social animals, flexibility is promoted by redundancy, simplicity of the behaviours and mechanisms [7].

There are various main characteristics of a swarm robotics systems [7]. The robots are autonomous and are usually homogeneous. The individuals are situated in the environment and they can act to modify it. They have local sensing and communication and therefore have decentralised control and no global knowledge of the states of the other robots in the swarm. The swarm also have to cooperate to complete a given task.

The key characteristics of swarms that are inspired from insect societies can be described as: decentralised, not-synchronised, with simple homogeneous units to an extent and not in large numbers. Swarm robotic systems are appealing as

robotic systems because, compared to centralised systems designed for the same task, they have very simple components. Therefore, the robotic units could, in principle, modularised, mass produced, and could be interchangeable and possibly disposable.

There are various applications of swarm robotic system in the real world. Swarms can operate on and under Earth's surface, under water or even in space and other planets. Practitioners have applied swarm intelligence to particle swarm optimisation (PSO), ant colony optimisation, unmanned underwater vehicles (UUV), swarm-casting, space exploration by NASA etc. Some of these application areas in swarm robotics involve foraging, surveillance, taxis and aggregation [17].

2.2 Swarm Taxis Algorithms

Aggregation involves robots having physical coherence when performing a task. Robots are randomly placed in an environment and are required to interact with one another which can be challenging with a distributed approach. Nembrini et al. [18] and Bjerknes et al. [8] developed a class of aggregation algorithms that make use of local wireless connectivity information alone to achieve swarm aggregation named α algorithm, β algorithm and ω algorithm. These algorithms are called swarm taxis algorithms. Swarm taxis involve a group of robots moving towards a beacon, usually a light source. The ω algorithm is the preferred algorithm to be used in swarm taxis because it has a more stable performance [19].

In swarm taxis algorithms, in order for the robots to move towards a beacon, Nembrini et al. [18] and Bjerknes et al. [8] allow the swarm to move towards an infrared (IR) beacon. Only the robots who are directly in line with the IR beacon are attracted to the beacon and illuminate the beacon sensor. An emergent property is swarm taxis towards the beacon. However, the robots do not individually have the necessary sensing capability to determine heading towards the beacon. The robots must work together to achieve movement in the right direction. To achieve this, there are three mechanisms that must be in place [19]:

- There has to be something that prevents the swarm from disintegrating, that is, if a robot moves too far it should be able to return back to the swarm
- The robots should have a minimum distance between each other so as to avoid collision.
- Once these two conditions have been met, a "symmetry breaking mechanism" is to be introduced to ensure that the swarm moves in the correct direction.

In swarm taxis algorithms, swarm maintains aggregation by the following [19]:

- Coherence Behaviour: Each robot has a limited range wireless communication and sending and listening to 'I am here' messages within their communication range. If a robot loses connection and the number of neighbouring robots fall below a predefined number, then we assume that the robot is moving away from the swarm and therefore it turns 180 degrees. We say that the swarm is coherent if there is a break in the overall connectivity lasts less than a predefined time constant.

- Avoidance Behaviour: Each robot has short range avoidance and long range beacon sensor. The robots use this short range sensor to avoid collision with each other or any blocks in the environment. The long range sensor is used to detect the light source.

- Symmetry Breaking Behaviour: This is how the information on the direction towards the beacon is captured by the robots on the swarm. In a swarm, not all the robots in the swarm would be able 'sense' the beacon as they could be blocked by the other robots.

Before we discuss the ω algorithm [10], a short summary of the α and β algorithm needs to be described so that we can understand the reason behind choosing the ω algorithm.

2.2.1 The α algorithm

When working with the α algorithm, the robots track the number of robots within their communication range by sending and listening to 'I am here' messages. If the number of robots fall below a predefined number, α, then we assume that the robot is moving away from the swarm and therefore it turns 180 degrees. However, this algorithm has some problems. If the value of α is too low, then the swarm disintegrates. Also, implementation of the α algorithm on real robots has not been successful; it has only proved successful in simulation [19].

2.2.2 The β algorithm

For the β algorithm, it also uses radio connectivity to maintain the swarm cohesion. The α algorithm makes use of the number of robots within their communication range while the β uses the number of shared connections. As the robots move around, they

send their unique IDs and a list of the IDs within their communication range. Therefore, when a robot loses a connection with a robot, it checks with the list gotten from other robots to see if they have ‘seen’ or also lost connection with this robot. With all this information, they can calculate the shared connections. If the connection to a robot is lost and the number of shared connections is less than a predefined value, β, the robot turns around. This algorithm requires a much more complicated communication system and also increased processing power although it maintains swarm cohesion much better than the α algorithm. Nembrini et al. succeeded in implementing the β algorithm on seven robots but they had to simulate the short range wireless communication. Both the α and β use wireless communication which is limited in range and it is essential that the difference between good communication and no communication is clearly defined. However, this technology is not readily available today [19].

It can be observed that although α and β algorithms are quite useful, when trying to implement on real hardware, another algorithm is needed which is how the ω algorithm was conceived.

2.2.3 The ω algorithm

In the ω algorithm [10], there is no wireless communication channel but rather sophisticated sensors and a timer. The default behaviour is to move forward and as this occurs, a timer is increased called the aggregation-timer. Every instance the robot performs an avoidance movement; that is avoiding collision with another robot, the aggregation timer is reset to zero. If the aggregation timer reaches a predefined value, this means that the robot has moved a certain distance away from the swarm and it needs to re-orient its position and move towards its perceived centre of the swarm. This algorithm is more stable and has been successfully tested on real hardware. Another advantage of this algorithm is that it frees up wireless communication bandwidth so as to fully optimise the sensor network.

In [8], Bjerknes et al. uses the ω algorithm in analysing various fault modes and their effects on swarm behaviour. From his analysis, it was concluded that when a robot is partially failed, it would have a serious effect on the overall swarm behaviour. Therefore, it is desired that the swarm should be fault tolerant.

2.3 Fault Tolerance

Fault tolerance is important in the field of robotics, especially in remote and dangerous areas. We can define it as the ability of the system to continue operating

when faults are present in the system. The system could be functional at a reduced level of efficiency regardless of the changes in the internal structure or the environment [20]. We can classify fault tolerance into three sub-categories: Fault detection, fault diagnosis and fault recovery. Fault detection involves discovering an anomaly in the system. There are two ways we can use fault detection in robotic systems: exogenous and endogenous. Exogenous fault detection is the process whereby robots detect faults in other neighbouring robots [21] while endogenous fault detection is the process whereby robots detect faults that occur within itself [22]. We can breakdown fault diagnosis into fault isolation and identification. Fault diagnosis requires knowledge of the set of possible failures and the relationship between observations and these failures. Fault recovery involves having a set of actions for a set of faults allowing the system to continue its operation with faults present [23]. There is a need for robots to have the ability detect, diagnose and recover from faults autonomously and continue on with their tasks. It has been assumed previously that swarm robotic systems are robust and scalable, that is fault tolerant, by default. It should be noted that robot swarms do exhibit a good amount of tolerance in the failure of individual robots compared to conventional systems, however it is incorrect to assume that these properties are automatic to all swarm robotic systems. The ω algorithm serves as our case study in our analysis of fault tolerance in swarm robotics. Analysis done by Bjerknes et al. [8] has disputed this assumption by using both reliability modelling and experimental trials on a swarm of robots that have had failures injected. The swarm size plays a role in this analysis. As the swarm size increases, the system reliability decreases. Therefore a more reliable method of fault tolerance is needed, especially for a larger swarm size.They describe three failure modes that could have a major effect on the operation on the swarm [8]:

- **First Case - Complete Failure of Individual Robot:** In this situation, the functional robots treat the failed robots as obstacles in the environment and avoid them completely. The effect they have on the overall performance of the swarm is minimal. The only way they can be detrimental to the performance of the swarm is that it reduces the number of functional robots in the swarm

- **Second Case - Failure of IR sensors:** It is highly unlikely that all the IR sensors on the robot would cease to function and the robot would leave the swarm and wander around the environment. However, if this occurs, the remainder of the swarm treats this failed robot as an obstacle as in the first case. Also, as in the first case, the main issue is that it would reduce the number of functioning robots in the swarm.

- **Third Case - Failure of robot motors only:** In this case, the motors

are faulty however all other sensors are functional. This has a serious effect on the overall performance of the swarm as these robots are partially failed robots. This partially failed robot anchors the swarm, disrupting the swarm taxis towards the beacon. This is because the robot is still active but stationary. Even if only one motor fails and the robot is turning on the spot, it is still has the same effect. This is the most serious case, due to the anchoring problem and is the basis of the research proposal.

In [5], Winfield et al. implemented a failure mode and effect analysis (FMEA) to analyse the common hazards that occur in swarm robots and their impact on the global swarm behaviour (aggregation is used as the case study):

- **Motor Failure:** It is assumed that the other functions are still working. When this happens, either one or both of the robot's wheels stop functioning.
- **Communications Failure:** This is when the wireless connections sub-system fails and the robot gets disconnected from the swarm.
- **Avoidance Sensor Failure:** This is when robots cannot sense other robots or obstacles around them causing them to bump into them.
- **Beacon Sensor Failure:** This occurs when the robot can not 'see' the beacon.
- **Control Systems Failure:** Generally, the control system is simple and consists of three behavioural layers: forward, avoidance and coherence. The failure usually manifests itself as motor failure or communications failure.
- **Total Systems Failure:** This occurs when a robot fails completely due to, for example, power failure, the robot becomes inactive and completely disconnected from the swarm.

There has been some work done, as explained below, in fault tolerance in swarm robotics and in robotics in general. This has been shown for various swarm behaviours. Most of the work done in respect to fault tolerance in swarm robotics is primarily in fault detection. From the Receptor Density Algorithm (RDA) [24], which is an immune inspired (which uses the artificial immune system as inspiraton) fault detection algorithm, to the Firefly algorithm [25], which takes inspiration from the synchronised flashing that has been observed in some species of fireflies, there are various fault detection methods in both multi-robot and swarm robotic systems.

2.4 Artificial Immune Systems (AIS)

Artificial immune systems (AIS) is the result of immunology and engineering working together. AIS has a plethora of benefits though it can be argued that in most of the application areas already established, other biologically-inspired algorithms such as evolutionary algorithms, neural networks, etc. might solve the same problem more effectively [26]. There are various application areas that AIS can be used in, but they can be categorised into three: Anomaly Detection (fault detection), Optimisation (clustering) and Learning (pattern recognition, robotics, adaptive control systems) [27]. The immune system can be divided into innate and adaptive immune system. The innate immune system deals with common problems without any heavy computation. They respond to pathogens generically without knowing the specific kind of pathogen that is being dealt with. They are used to quickly destroy diseases. Adaptive immune system is pathogen specific, referring here to biological systems. There are various immune processes which are mentioned in [28] that would be useful for a more thorough review.

AIS has a lot of potential to be an important aspect in biological-inspired algorithm, however to create an effective AIS algorithm, there are various features that are unique to it[26]:

1. They will be embodied, that is, they should not work in isolation but the AIS should work with other systems to create a fully functional system.
2. The innate and adaptive immune system models should be used together as this might make the system more functional. By exploiting the full 'natural' immune system in the models, the true value of the immune metaphor would be revealed.
3. They will consist of multiple, heterogeneous interacting, communicating components.
4. The components should be easily and naturally distributed
5. They should be able to perform life-long learning, that is, they should have some sort of memory to remember past states, as would be seen in the natural immune system.

There is a significant lack of theoretical work in AIS. How can creative, useful algorithms be properly created if the underlying concept is not understood. By fully

understanding the science behind the immune system, AIS practitioners can successfully create immune algorithms for specific application areas. It is very difficult to create a generic AIS without knowing what application area that it would be used in. The immune algorithms are inspired by immune processes and they are divided into four major groups: negative selection, clonal selection, immune networks, danger theory (this theory makes use of both the innate and adaptive immune models. Most AIS systems make use of just the adaptive immune models which is not an ideal feature of an AIS feature according to the list above).

AIS has been used in a fault tolerance capacity, [29], although it has not been used for fault recovery. The idea would be to evolve immunity/fault tolerance of the swarm robotics system. Immunity is being insusceptible to faults or hazards. The concept of immunity can be derived from the immune system which is the theory behind how my PhD would work.

Each fault would be handled in a different way; diverse problems require a range of specialist solutions. For every type of commonly known fault, as discussed in [5], there would be a corresponding predefined fault recovery strategy. There would however be only a single set of fault recovery strategies for all the commonly known faults as mentioned above: sensor failure, communications failure etc., which would be generic for most swarm behaviours. The swarm would try these generic behaviours first. This is an adaptation of the innate immune system. If these strategies do not work, if for example, the swarm is in a new environment or the swarm could be assigned with a different swarm behaviour, their parameters of the specific type of fault recovery strategy are to be evolved first. This is an adaptation of the adaptive immune system.

Before the proposal is discussed, it is necessary to review work done in various fault tolerant capacities.

2.5 Fault detection in Multi-robot systems

Roumeliotis et al. [30] presents an approach to fault detection and identification on board in mobile robots, especially sensor failure, called Multiple Model Adaptive Estimation (MMAE). They present three stages of sensor failures: we can say two are ‘hard’ in the sense that the sensor is stuck on a value and subsequent values are ignored from then on. The last failure can be classified as ‘soft’ in that the sensor degrades but could still be useful. The computation necessary to detect all these faults, ‘hard’ and ‘soft’, was not available [30]. Kalman filtering is a popular technique for state and parameter estimation. The MMAE techniques uses Kalman

filters and is used to fault detect in mobile robots. A Kalman filter give a measure of the difference (could be called residual) between the measured sensor value and the value that has been predicted by the filter that contains the model. The residual is used in the filter to update this difference and has been classified as an excellent way to indicate that the robot has fault. This has been demonstrated using three sensor faults. This architecture is in the fault detection and identification modules and the output from the fault detection module notifies if a fault has occurred. The module is already active even after a fault has been detected in case the fault has been good or a new fault is detected.

Fagiolini et al. [31] addresses a distributed Intrusion Distribution System (IDS) that can be used to detect faulty robots in a multi-robot system. They use monitors to share information that has been collected, which means it has a global view of the system. The motivation behind this paper is to provide an infrastructure that would be able to detect complicated behaviours of a robot that deviate from the main strategy. It could be expected that a robot could deceive the model of cooperation and neighbouring robots that monitor its progress by using their knowledge of the system's state. The systems that are considered are systems where the individual robot plans its movement based in the actions of the robot itself and the neighbouring robots. The objective is to evolve a synthesis technique that makes it possible to build the IDS for the multi-robots. There are two concepts of the IDS: a decentralised monitoring system whereby each robot allocates a reputation to all its neighbours. Reputation is a measure of the robot's cooperativeness. The second concept is the agreement mechanism which is when all the monitors share all the local information that has been collected can meet to make a network decision. They decide not to use a completely decentralised system because the robots only have access to a partial part of the overall state of the system. and the faulty robot could be affected by a robot outside the range of the robot monitoring it.

Heredia et al. [32] [33] addresses sensor and actuator fault detection techniques in small autonomous helicopters. The helicopters that the experiments are conducted on are called the MARVIN helicopters. This paper investigates detection of both the possible sensor and actuator faults. In detecting sensor faults, it is achieved by monitoring the sensor outputs continuously. Typically, the measurements are predictably with some degree of uncertainty due to noise and random disturbances. They observe the measured value form the predicted value. There are five different failure types that were considered:

- **Total sensor failure.** This is a serious type of failure. This happens when the sensor stops working and outputs zero. This failure type could occur due to communication or electrical issues.

- **Stuck with constant bias sensor failure.** This happens when the sensor is stuck at a constant bias and the outputs a constant.
- **Drift sensor failure.** This failure type is particularly common in analog sensors. This could happen due to internal temperature changes or calibration issues and the output has a constant term added to it.
- **Multiplicative-type sensor failure** This failure type occurs due to a scaling error in the output of the sensor (it is 'added' to the sensor value).
- **Outlier data sensor failure.** This is a failure type that occurs mostly in GPS sensors. It happens at a single point but results in a large error that is given by the GPS sensor. However after this, the subsequent measurements are correct. This could be caused by various reasons such as internal signal processing algorithms, temporary satellite signal blocking.

In detecting the actuator faults, the detection technique is designed by using the flight data gotten from two different sources: a non-linear mathematical model and the real flight data from the MARVIN helicopter. They are able to generate datasets using the mathematical model in simulation. They make use of the simulation data to detect faults in all the actuators: main rotor, tail rotor and both the rolling and pitching cyclic inputs. The main problem however with these techniques, if the errors are considered small, the fault detection technique might not be able to differentiate it from noise and other disturbances.

These approaches to fault tolerance for multi-robot systems do not work for robot swarms due to the design and implementation of multi-robot system. For swarms, each robot is simple and the swarm work collectively to achieve their goal however, multi robot systems consists of robots that are each specialised for a specific task.

2.6 Fault detection in Swarm Robotic Systems

Tarapore et al. [29] discusses fault detection in multi-agent systems which has ranges including multi-robot systems. According to this paper, majority of the fault tolerant systems available specify a characterisation of normal behaviour, and train a model to recognise them but these models require specific knowledge of the normal behaviour. Any behaviour that is not recognised is considered abnormal. However, multi-robot systems that make use of these models do not consider variations in normal behaviour and it would consider these variations abnormal. They tackle this problem by taking inspiration from the regulation of tolerance and autoimmunity

in the adaptive immune system. The adaptive immune system or acquired immune system is a subsystem of the overall immune system in the human body. The system remembers the immune response after the first time a pathogen is discovered. The idea behind the adaptive immune system is that there is no previous knowledge of the pathogens, however, the immune system is able to adapt to their presence. They propose using the Cross-regulation model (CRM) which captures the robust maintenance of immunological tolerance by allowing the system to discriminate based solely on their density and persistence in the environment. They used an agency-based simulator to model a situation where the agents have to tolerate certain behaviours and activate an immune response against the others. They demonstrate the capacity of the systems to tolerate the normal swarm behaviour that could be characterised as having a high density while also activating an immune response against the abnormal robots. The system also accommodates the variations in normal behaviour.

Alan Millard et al. [34] proposes a novel method of exogenous fault detection that is capable of detecting partial failures, based on the comparing expected and observed robot behaviour. The robots do not learn the expected behaviour of every other robot's behaviour online, rather they each have a copy of the robot's controller which they can instantiate in an internal simulator. Therefore, if the robot's controller can be instantiated in an internal simulator, then the behaviour can be predicted. They assume however that the the swarm is homogeneous, that is, each robot has an identical controller. In order to compare the behaviours, there has to be a method that each robot would use to observe one another. To simplify the problem, they decided to provide each robot with information about the position/orientation of other robots, collected using a tracking infrastructure that observes the swarm from a bird's-eye view. The robot controller is instantiated within the simulation, embodied in a simulated model of the real robot, and is used to generate predictions of non-faulty behaviour. Any major difference between these predictions and the robot's observed behaviour may indicate fault. They investigate partial motor failure as this would be more difficult to detect.

Christensen et al. [35] proposes a new approach on fault detection for autonomous robots by using neural networks. The idea is that hardware faults would change the sensory data and also the actions performed by the control program. If we are able to detect these changes, we can confirm the presence of faults. They tested this by collecting data from three different tasks, find perimeter, follow the leader and connect to robot, performed by real robots. During training runs, they record the sensory data from the robots while they are operating normally and also after a fault has been injected into the robots. By using back-propagation neural networks, they are able to synthesise fault detection components based on the collected data from

the training runs. They also extend the possible faults and show that a single fault detector can be trained to detect several faults in the robot's sensors and actuators. The fault detectors can be synthesised to be robust to the variations in the task and also can be trained to allow one robot to detect faults in another robot.

Another paper by Christensen et al. [25] proposes a distributed algorithm that detect faults exogenously in other non-functional robots in a swarm robotic systems. This algorithm is inspired by some species of fireflies. By making use of local communication between the swarm, the robots are able synchronise and reach a stable state whereby they flash an light emitting diode (LED) regularly together. The time it takes for the robots to synchronise depends on its size and it does not have to be synchronised at a global level for the algorithm to function. When a robot develops a fault, the LED stops flashing. By observation, functional robots can detect the failed robots by the lack of flashing over a certain period of time. This is done by robots analysing camera images to detect changes in each other's LEDs. Due to this, the functional robots also know the location of any LEDs in its vicinity. It should be noted that a robot's LED can also stop flashing voluntarily if it detects a fault within itself. However, the robots would not be able to tell the difference between a failed robot or a robot that that its LED simply stopped functioning. This algorithm detects if a robot is faulty but does nothing in regards to fault diagnosis and recovery in the swarm.

Lau et al. [24] uses the Receptor Density Algorithm (RDA) in fault detection in swarm robotics. The authors propose using statistical classifiers, such as the RDA (an artificial immune system), to enable the swarm to be fault tolerant in dynamic environments. They address this through comparison in which a robot would compare its behaviour to other robots, in its communication range. By doing this, the robot would be able to self-detect if a fault has occurred endogenously and would be able to differentiate whether the robot has developed a fault or whether the change of behaviour is due to a change in the environment. The RDA is inspired by the T-cell receptor and its ability to discriminate between healthy and unhealthy cells. It works by taking in an input spectrum, dividing it into discrete locations. A receptor is placed at each location and takes an input. From this, it produces a binary classification which informs us whether that location is faulty. The binary classification is performed via the receptor location and negative feedback. More information on the RDA can be gotten from [36] and [37]. This fault detection mechanism is quite impressive as it takes into account different failure modes to the wheels in dynamic environments. However fault detection is the only aspect of fault tolerance that work is being done on. Again, this algorithm does nothing in regards to fault diagnosis and recovery.

2.7 Fault Recovery

This section discusses some work done in the area of fault recovery. When understanding fault recovery in multi-robot systems, it should be noted that not all robotic systems have a mechanism to detect or diagnose faults, however, they can still perform a kind of fault recovery strategy that give an inherently fault tolerant system.

2.7.1 Single-robot systems

Bongard et al. [38] discusses a fault recovery strategy when dealing with a fault in a singular robotic system. This paper deals with a four legged robot with eight motorised joints, eight joint angle sensors, and two tilt sensors that can autonomously recover from damage through continuous self-modelling. In this paper, they describe a continuous process whereby a robot still performs after injury through self-modelling. The robot is able to indirectly deduce its own morphology through "exploring" itself and using these self-models to synthesise new behaviours. If the robot?s structure changes suddenly, the process above is used to restructure its internal self-models, leading to the robot being able to generate different behaviour. This enables the robot to continuously diagnose and recover from faults. This method does not involve any form of redundancy or even other contingency plans designed for anticipated failures. This process is made up of three algorithmic components that are always executed by the physical robot regardless whether it is moving or resting: Modelling, testing and prediction.

2.7.2 Multi-robot systems

An example of fault tolerant in multi robots systems is work done by Parker et al. called ALLIANCE [39]. ALLIANCE is a software architecture that allows heterogenous robot teams, which are individually highly functional to make necessary decisions throughout the task given by taking into account the environment, interactions of the other robots and their internal states. These teams can work in dynamic environments and are able to respond adaptively and reliably to changes in the robots, most especially mechanical faults, and changes in the environment. These robots can individually cope with selecting and performing various actions that can be expected in a dynamic environment. There is no centralised control so it is a fully distributed system. This is so as to utilise the robustness characteristic that can be associated with distributed robotic systems. ALLIANCE has been tested on

an actual robot team in a laboratory environment performing waste dump. It is a good example of a fault tolerant system that performs some sort of fault recovery mechanism with or without any fault detection. However, this mechanism deals with a multi heterogenous robot that individually has high level functionality, therefore would be expensive to produce and maintain. Also, each robot in the team have to explicitly coded for a specific task. If the order of the tasks is important, it also has to be explicitly programmed. Any sub-tasks is not fault tolerant in that the main priority of the team is to finish the task irrespective of how efficient the action is.

Gerkey et al. [40] is another example of fault tolerance in robotic systems, discussing achieving intentional cooperation in robotic systems that comprises of autonomous heterogeneous robots that are prone to failure in noisy and dynamic environments called MURDOCH. This paper deals with dynamically allocating tasks to these robots. The inspiration behind this is from the distributed artificial intelligence (DAI) community. The main problem is multi-robot cooperation in tasks however they focus primarily in task allocation. Given resources and sub-tasks, MURDOCH assigns the resources to the necessary tasks as efficiently as possible. The three aspects that MURDOCH takes into account during task allocation is the resource usage, time taken to complete task and overhead communication between the robots. They demonstrate fault tolerance in that if a robot takes too long to execute an assigned task, it can be assumed that the robot has failed and therefore the task can be reassigned. MURDOCH is able to demonstrate a fault tolerant system in dynamic task allocation for heterogeneous robot teams. As discussed in ALLIANCE, each robot is assigned a specific sub-task and although this can change, depending on the performance of the robot, these behaviours have already been explicitly written.

Dias et al. [41] presents a distributed architecture, "TraderBots" that can also form centralised sub-groups so as to improve efficiency. They use a market approach to coordinate a robot team to complete a task. This architecture is similar to MURDOCH and ALLIANCE. They classify each robot as a self-interested agent while the robot team is an economy. The goal of the team is to complete the task while minimising the cost. Each robot works to minimise its individual robot cost but the revenue is gotten from completing the team's tasks. The overall cost is calculated by summing up the individual's robot cost and any deal made by any robot (only profitable deals) will add up to a reduction of the global cost. The individual robots compete for different tasks which enables the system to read the competing local costs of each robot. This architecture is fault tolerant in the sense that the robots have to bid for tasks so as to complete them. If a robot is not "healthy" enough to perform a task, then it does not 'win' the bid, as it would not be an efficient way to reduce the overall cost of the team.

We have seen the work done so far in fault detection and recovery in robotic systems including swarm robotic systems. Another way of doing recovery would be a trial-and-error process such as stochastic optimisation. One such technique would be evolutionary computation. Evolutionary techniques involves randomised search heuristics; this is where the technique is executed in a loop, comparing the solutions until the best solution is discovered. The reason why this is a popular choice is because it provides novel solutions to problems that the designer might not necessary think about. It removes the necessity of breaking down the controller code that leads to the emergent global behaviour. It relies on evaluating the overall swarm behaviour that started from the individual behaviours. The approach that evolutionary algorithms take in designing controllers is that the controllers are evaluated for their ability to produce the needed behaviour. The algorithm selects the controllers and evaluates their fitness in the environment they are working in.

2.8 Evolutionary Robotics

Evolutionary robotics (ERs) is a derivation of optimisation, therefore, before discussing ERs, general optimisation techniques need to be discussed so as to give an overview of these techniques and how they relate to ERs.

2.8.1 Random Optimisation

In [42], metaheuristics, can also be called randomised search heuristics, is described as a term that can be used to discuss the primary field of random optimisation. Stochastic optimisation is the general category of algorithms and techniques that employ some randomness to discover optimum solutions to difficult problems. These algorithms are used to address problems where there is little knowledge known about the problem and the practitioner has no idea what the optimal solution looks like however the search space is too large for the brute-force search to be considered; the practitioner refers to the person that is trying to find the optimal solution, that is, the developer.The simplest way to perform a search is by doing random search. This involves randomly selecting and evaluating solutions till some sort of termination criteria has been met and then returning the best solution that has been discovered. However, there are better alternatives to this, such as Hill-Climbing [42]. This is done by starting a random set of solutions and then making a small, random modifications to this set and testing it. If this version is better, discard the previous one, otherwise discard the new version. Afterwards, make another modification to

the current version, the version not thrown away, and if his version is better keep it otherwise discard it. This process is repeated till the termination criteria is met.

Hill-climbing is a simple algorithm that utilises the belief that similar candidate solution tend to behave similarly, similar quality, so that by adding the modifications you get a set of small changes in the quality of the solutions. This allows us to “climb the hill” to good solutions. This is a main feature of metaheuristics; most metaheuristics are basically some combination of hill-climbing and random search. There is also the Gradient-based optimisation technique [42],: Gradient Ascent which is a mathematical method that can be used to find the maximum of a function. The main idea behind this is to discover the slope (optimum) and move to it. It does not assume that the function is known, however it assumes that the slope can be computed. This algorithm can be run till time runs out or it discovers the best solution which is when the slope is zero. However, if there are various points on a function that has a slope of zero, which could lead to premature convergence, which is an important issue when using the Gradient Ascent optimisation technique. There is also the Gradient Descent technique that is used to find the minimum of a function. It also follows the same principles as the Gradient Ascent.

There is also Simulated Annealing which was developed by researchers in the mid 1980s [43]. This is similar to Hill-Climbing but it differs in its decision as to when to replace the old version of the solution set with the new version. If the old version is better that the new one, then replace the old version with the new one as normal. However, if the new version is worse than the old version, you could still keep the old version. This is determined by a probability. The algorithm performs a random walk in space which allows the algorithm to discover new solutions irrespective of how good the solution is. It has a tunable parameter, t, that causes the probability to discard the current solution to be close to zero when t is close to zero and if t is high, the probability is close to one. The tunable parameter starts at a high number and is decreased slowly. The slower the rate of decrease, the longer the algorithm resembles a random walk and explores the search space more. There is also the Tabu Search which was created by Fred Glover [44]. It employs another approach to exploring the search space. The algorithm keeps a list of the recently considered solution, of a considerable length (tabu list) and does not return to these solutions until they have been far in the past in the optimisation process. If the tabu list is too large, the oldest candidate solution is removed from the list and is therefore not a taboo to reconsider the solution.

Population-based methods are different from the methods discussed above because they use a sample of multiple solutions rather than a single solution. Each of these solutions are randomly modified and assessed for their quality. Most of the

population-based methods are inspired from biology: a set of techniques known as Evolutionary Computation [45]. An algorithm from this collection is the evolutionary algorithm (EA). Evolutionary algorithms utilise the collective learning process of a population of individuals where each individual represents a search point in the space of potential solutions to a given problem. In swarm robotics, evolutionary techniques have been used in order to obtain robust and efficient group behaviours based on self-organisation [46], and might possibly be used for fault detection or recovery. Genetic algorithms also belong to evolutionary computation techniques that perform optimisation or learning tasks with the ability to evolve; there are other algorithms that also fall under evolutionary computation techniques which includes evolution strategy (ES), evolutionary programming (EP), and genetic programming (GP) [47].

Evolutionary Robotics (ER), usually in single robot systems, is a technique that uses evolutionary computation to develop controllers for autonomous robots. It uses evolutionary algorithms (EAs) to develop, modify or completely change the controller of the robots thereby increasing the autonomy of robots [48]. The time taken to evaluate solutions for evolutionary robotics is very long because EAs usually require hundreds and sometimes thousands of iterations. This has lead to two evolutionary robotics research approaches: Online Evolution (Situated method) [49], [50] and Offline Evolution (Simulate-and-transfer method) [51].

2.8.2 Online Evolution

The emphasis is on robots experiencing the world directly through the sensors and therefore evolving the exact needs of the robot and the task environment. The advantage to this is that there is no aspect of modelling required from the designer, a situated evolutionary algorithm can cater for design tolerances between robots and a situated algorithm can discover elegant solutions related to the robot. However, EAs require a population of solutions to select from. Some of the methods used [52] is to maintain a population of controllers on a single robot and sequentially evaluate the controllers. This is called time-sharing or even encapsulated evolution. This method has a major disadvantage in that if a good solution has been found, it still keeps looking for a new controller solution making the real world performance of the robot unreliable. It should be noted that if the environment changes often, the evolutionary aspect of the robotic system would be trapped in a continuous evolution and this is a waste of time. It is desirable to be robust to change in the task environment. However, sequentially testing controller solutions on a single robot is not an efficient method. Also, because EAs require using both explorative and exploitative, there is

potential for the evolutionary exploration of controller solutions to create unreliable and dangerous mutations of a previously stable controller.

2.8.3 Offline Evolution

Generations of controller solutions are evaluated virtually through a simulator which avoids taking time doing real world analysis. After a sufficient amount of generations, the best evolved solution is transferred to the robotic system. Simulations also help the robots gain a wider and global perspective of the world that is being observed and integrates new information back into the evaluation process. This allows for an accurate fitness assessment as the simulation takes into account interactions between the robot and the environment. Of course, this method isn't without its challenges which involves the accuracy of the simulation and transferring the solutions to actual hardware. By using simulation, there is a risk that the EA would discover and exploit inaccurate features which are not present in reality or the simulation would not take into account important features that are present in reality. There is a term called reality gap coined by Jakobi et al. [53], which is when there is a discrepancy between the real world and the simulation. This is present in both single robots and swarm robots. This can be considered in 3 categories of correspondence: robot-robot correspondence (differences in morphology), robot-environment correspondence (differences in their interactions such as sensor data and actuators) and environment-environment correspondence which is representation of the relevant features of the environment.

Another argument is that the evolutionary development happens as a distinct process before deploying it on real robotic hardware [51]. Because EA is an offline process, evolutionary adaptation stops once the solution is found. This is unlike the evolution of the situated evolutionary robotics which is continuous open-ended evolution where the EAs forms the main process of lifetime adaptation. Some researchers have looked at transferring the simulated offline process to a distributed (spread across multiple agents) online process so as to re-tune the solution for the real world. Another approach is to combine evolution and learning like evolving the controllers offline but using online learning algorithms to shape the phenotype once the solution is transferred and operating on real robots. Combining offline evolution and online learning is a promising approach but the reality gap still exists. The designer must isolate the key characteristics to simulate, which subsequently forms a limit on any post-transference optimisation. A more recent approach is to allow a robot to autonomously adapt a simulated representation in direct correspondence to the robot's experience of the real world. The hypothesis is that exploitation of

simulated features that do not correspond to reality can be marked as not evolutionary profitable. Thereafter, realising exploitations can be used to efficiently discover discrepancies in correspondence and amend them to a better fit for the real world.

2.8.4 Evolutionary Swarm Robotics

Evolutionary swarm robotics involves using EAs to optimise robot controllers to give us a desired global group behaviour. There are few examples of distributed online approaches of EAs in swarm systems [54], [50]. Distributed online approaches involve using a distributed (shared amongst the robots) evolutionary approach on many robots. A robot would maintain and evaluate only a single controller solution but selectively reproduces with many of the other robots working with the same distributed algorithm. Basically, the group of robots represents the physical evolutionary population of controller solutions. This reduces the time needed to evaluate the entire evolutionary population. In this approach, the population size depends on the number of robots and how they interact with one another in the environment. There is an emphasis for evolutionary process to be localised so that the system remains scalable. Also if the fitness is localised, the individuals can assess themselves and there is no need to get the fitness of the whole swarm. Because we are dealing with swarms, the fitness of the individual must, in a way, relate to what other robots are doing. However, because a swarm is decentralised, this is very difficult to achieve.

However, there is a lot that can benefited from distributing the EA [54]. The individual robots in the swarm are computational units from which there can be an increase in processing power by gains in processing can be made by splitting the evolution into parts but then running them simultaneously. The robots represent many experiences and can therefore be used as a form of parallel assessment. From the first argument, parallelism can accelerate the process and make situated evolution more timely. An online evolutionary method is embodied in the robot and therefore specific to the requirements of the situated environment.

Konig et al. [55] evolve obstacle avoidance behaviour and highlight some immediate problems with an on-line, distributed evolutionary method:

- The fitness of the phenotype is coupled to the task environment and cannot be anticipated. Therefore, there is no exact measure of a solution to relate between robots.
- We can not measure the global fitness as a swarm is decentralised and localised.
- The fitness value is an approximation that occurs after a time period of assessment and therefore, robots exchange out of date information.

There have been some investigations done on on-line distributed evolutionary method which can be listed below [50]:

- Evaluating in the real world can be very slow, therefore one must have a strong evolutionary method of selection.
- Robots are mobile and they evaluate physically therefore the initialisation of a new evaluation does not have controlled circumstances.
- The performance of the robot to complete a task is very important and is disrupted if a robot keeps on evaluating unreliable behaviours. This causes some difficulty in choosing a mate for evolutionary selection.

Performing evolution offline means that evolutionary development can be analysed on a simulator while online evolution can only be analysed on real robots.

Some work [56] has been done to conduct evolution online by using the robot's limited computing power that is onboard. This enables the robot to operate in a self-contained way without relying on external for assessment or even additional computational power. The EA mechanism is carried out without any external supervision. The main advantage of online evolution is that if there is a change in the environment or task, robots can modify their behaviour accordingly. Each evaluation takes a significant time on real robots which is why the solution is not feasible. Another method is conducting the evolution offline, as discussed previously, in simulation and then transfer to real robots after evolution to avoid the delay when performing all the evaluations on real robots. However, one of the problems with offline evolution is the concept of reality gap.

There have been two techniques, [53], that have been developed to aid with the concept of reality gap: a) improve the simulation model and b) make the evolved solutions more robust to discrepancies. First, the categories above (online and offline evolution) are sampled from the robotic hardware to create a realistic representation in simulation. The second is that the categories are supplemented in simulation with enough noise so as to make them not reliable for exploitation. The main problem is that using these approaches listed above, the designer must know what features should be sampled and what should be covered in noise. Using simulation would definitely reduce the representation's accuracy and therefore, some aspect of choice from the designer which would be a limit to the development of the system.

2.8.5 Issues with Evolutionary Robotics

Evolutionary robotics, although promising, have some issues which could compromise how efficient evolutionary robotics are in actual robotic systems.

2.8.5.1 Reality Gap

Reality gap occurs because after the controllers have been evolved in simulation, there are no longer valid when transferred to real robots. This is because the features in the simulated world does not exist or is not as accurate as what is in the real world. The evolved controllers may fail to achieve the desired behaviour after transference [57]. This difference could be due to inaccurate sensor modelling to oversimplifying the features of the real world in simulation. The work done in this thesis is done entirely in simulation and reality gap is one of the reasons for this. This phenomenon is quite common in ER. In the instance where there is no noise present in the simulator, the evolution may produce a behaviour that only uses a range of narrow sensor values. After transference from simulation to the real robot with a wider range of sensor values due to electrical and mechanical reasons, the robot may not function in the way the simulated robot functioned. There have been some solutions that have been brought forward therefore allowing for significant progress in the area.

Jakobi et al. [53], proposes introducing noise into the simulation to make the simulation robot more realistic as real robots would be noisy. They tested this and were able to prove that is plausible to evolve controllers and transfer it to a real robot successfully and the real robot would produce similar behaviours to the simulated robot. They made use of network-based control systems for developing basic behaviours are evolved in simulation of different levels of accuracy and it is then transferred to a real robot. It is argued that the simulation should be completely based on a large number of collected empirical information, and should be constantly checked to confirm it is accurate. Also appropriate amount of noise at all levels needs to be taken into account. Using networks of noise tolerant units that adapt as main component of the system helps reduce the differences between the simulated world and the real world. Noise is to be added to the empirically random properties of the robot which would help to deal with the weakness of using the simulation by making them indistinctive. The control system resilient enough to deal with the noise envelope may be better suited to deal with transferring the evolved controller from simulation to the real world. It should be noted that as the robot interactions with the environment becomes increasingly complex, the simulations becomes harder to develop.

2.8.5.2 Bootstrap Problem and Deception

Another issue related to evolutionary robotics is the bootstrap problem and deception [58]. Fitness functions are an important part of evolutionary algorithms and there are some difficulties when evolving difficult behaviours which is not usually

discussed which is not usually discussed by practitioners. This could be caused by the bootstrap and deception problem. The bootstrap problem happens when the task at hand is too difficult for the fitness function to apply any sort of selective pressure to a random population of initial candidate solutions. [59] describes selective pressure as a term mostly used to describe selecting best individuals in a population where high selective pressure signifies a strong emphasis on best individuals and low selective pressure signifies a weak emphasis on the selection on the best individuals. All the individuals, at the start of the evolutionary procedure, may perform equally badly, thereby causing the evolution to search in the wrong part of the search space. Deception happens when the fitness function is unable to build a gradient that leads to a global optimum. Instead, the evolution is lead towards a local optima. This can also be called premature convergence and it leads to a sub-optimal solution. A solution that has been proposed to address these problems is to directly aid the evolutionary procedure. There are three approaches that have been adopted: incremental evolution, behavioural decomposition and a semi-interactive approach (human-in-the-loop).

Incremental evolution [60] involves breaking a task into different components that are easier to solve individually. This can be difficult to achieve because as the order of the tasks is important. Also, when switching from one component to another, we have to keep in mind the time taken for execution of each component of the evolution. Behavioural decomposition [61], [58] involves breaking down the robot controller into sub-controllers and each of these sub-controllers are either preprogrammed or evolved separately to solve this sub-task. At the end of the evolution, the sub-controllers are combined together by doing a second evolutionary process to give the final controller. Detailed knowledge of the task has to be known because the controller has to be divided and therefore multiple evolutionary procedures has to be set up. Another approach used to assist in the evolutionary process is using the human-in-the-loop [62], [58] creating a semi-interactive evolutionary procedure. This is done by letting the users to guide the evolution towards the global optima by indicating to the procedure intermediate steps that the robot must go through during the task. A gradient is created guiding the evolution through these intermediate states, assuming that by reaching the intermediate states, it's a stepping stone towards the controller moving to advanced intermediate states. This approach is appealing because the user can interrupt to add his/her knowledge but there are multiple open questions as to how this can be applied to difficult tasks. As the difficulty increases, more human knowledge might be needed but the user faces the issue of human fatigue when using this technique.

Another solution that has been proposed in relation to the bootstrap and de-

ception problem is to introduce a diversity technique. The main idea is that by introducing diversity into the evolutionary process, the EA might be able to avoid being deceived because it would be exploring other areas in the search space. There are various methods that can be used to promote diversity in the evolutionary process such as: novelty search and reconciling exploration or exploitation of the search space.

Novelty search [63] is an idea where how novel a solution is compared to the other solutions is how the evolutionary algorithm evaluates the solutions. This is an example of an approach that is not fitness based. The algorithm functions by: a) measuring the *novelty score* of a behaviour by measuring the nearest neighbours (the number of the neighbours is determined by running experiments) and b) the solution is added to a file randomly or whether if the novelty score is above a predefined threshold. Solution from unexplored regions in the search space usually receive high novelty scores, therefore the evolutionary algorithm moves towards solutions that are new and different. An application of this algorithm, presented by Christensen et al. [64] is described below. Another method that can be used reduce the bootstrap and deception problem is to direct the evolution to increase its exploration and exploitation of the solution search space. Novelty search focuses more on exploring the search space while fitness based evolution focuses more on exploitation which is usually a narrow region in the search space. One approach to this is to combine novelty search with fitness based algorithms as discussed by [65]. Another approach is called the minimal criteria novelty search (MNCS) [66]. It is an extension of novelty search in which the solutions must meet one or more domain-dependent criteria before it can be selected to reproduce.The reason for this is to reduce the search space that to allow for suitable behaviours. This approach however has some issues because it restricts the novelty search from exploring possible fruitful regions in the search space and therefore the criteria has to carefully selected. Also if no solutions meet the requirements, including the initial population, then there is no selection pressure and the evolutionary process starts a random drift. [67], [68] describes another approach called the Pareto-based multi-objective EAs (MOEAs) that optimises both the diverse behaviour and the fitness. This approach automatically switches between the exploration and exploitation phases based on the behaviour diversity objective and the fitness objective. Through the evolution, the fitness could be maximises whilst reducing the diversity of the behaviour or the diversity of the behaviour could be maximised whilst reducing the effect of the fitness objective. Therefore there are numerous tradeoffs between the performance of the evolution and its ability to produce diverse behaviour. Using both objectives allow the evolutionary process to move in multiple directions allowing it to not get stuck at local optima [69].

The fitness function is an important part in the evolutionary process. However, it is also one of the most difficult aspect of the evolution. The evolutionary process depends on the fitness function as this is what is used to evaluate the candidate solutions and would later determine the best and worst solutions. If the fitness function is not as accurate as possible, we would not be able to get optimal solutions; the solutions could even be wrong (the evolution could be searching the wrong part of the search space). Practitioners have come up with some new ways where we can evaluate the candidate solutions without depending on the fitness function.

2.8.6 Other methods of evaluating Candidate Solutions

There have been other methods, not fitness functions, that have used in evaluating candidate solutions which are explained below. However, before discussing that, the effect of the fitness function in evolution has to be analysed.

2.8.6.1 Analysis of fitness function in evolutionary robotics

Soorati et al. [70] discusses the effect of the fitness function in evolutionary robotics. In this paper, they study the effect that different fitness functions have on the robot's performance. The challenge in evolutionary robotics, especially in the design of the fitness function is to maximise the complexity of the task and also minimise the necessary the deductive knowledge needed. When designing the fitness function, a lot of effort is put into the design of the fitness function as this is used to summarise how close the genomes are to the best solution. The researcher would perform some initial experiments so as learn some specifics about the goal the robots are to accomplish (deductive knowledge). Knowledge from this could be used in the design of the fitness function but we want to minimise the need to perform initial experiments; ideally, we want the evolutionary computation to be a black-box optimiser.

Nelson et al. [71] defines the quality of an evolutionary robotics if it accomplishes two things: a) it is measured by how complex the task is and b) the amount of deductive knowledge is used in the fitness function so as to generate successful evolved controllers. They actually suggest that an ER approach can be classified as an improvement over an ER approach done previously on the same task if the new approach uses less deductive knowledge in its design of the fitness function. Soorati et al. [70] investigates four different classes of fitness functions with varying degrees of deductive knowledge: behavioural fitness functions (high deductive knowledge integrated, BFF), functional incremental fitness functions (moderately high deductive knowledge integrated, FIFF), tailored fitness functions (average deductive knowledge integrated, TFF) and aggregate fitness functions (low deductive

knowledge integrated, AFF). It is difficult to measure this deductive knowledge which makes classifying the fitness functions difficult.

AFF is the lowest degree of deductive knowledge and only evaluates and selects the best robots on whether the task has been completed. How it is done is not relevant. The main problem with this technique is that there may be bootstrapping and there is no guide for the evolution through intermediate solutions. BFF measures how the task is completed. When designing this fitness function, deductive knowledge on the various options available to complete the task is required. This approach is complex as it requires knowing effective solutions to the task or initial experiments that help to determine solutions. TFF combines both BFF and AFF therefore being more complex than both AFF and BFF because the components are usually multiplied or added together. FIFF are sets of fitness functions that are used separately and slowly in stages. The evolutionary procedure is done in intervals of generations where only a single fitness function is used. That is, for every interval, there is a corresponding fitness function. The controllers are evolved in simulation using the NeuroEvolution of Augmenting Topologies (NEAT) which is used for neuroevolution approaches.

These fitness function classes [70], where implemented on three tasks: movement with object avoidance, goal homing (move as close as possible to a light source that is located in the environment) and periodic goal homing. After executing the evolutionary procedures on these tasks, they concluded that TFF and BFF influences the evolution positively. However, these two fitness function classes require a high degree of deductive knowledge in their design which contradicts what we want: the evolutionary computation technique as a black-box optimiser. More work needs to be done in this aspect of evolutionary computation (minimising the deductive knowledge required in designing the fitness function). The standard approach to evolutionary techniques in robotics involve fitness functions however some studies have been shown the evolution can still be done without the need of a fitness function by substituting it with some sort of behavioural diversity or some other method of evaluation. Some of them are discussed below.

2.8.6.2 Implicit fitness function examples

Bredeche at al. [72] proposes a new approach in evolutionary computation that eliminates the need of the fitness function. The motivation behind this paper is to have a fixed number of swarm robots and allow them face unknown environments. In this scenario, they specify that the human designer has no idea what environment the robots would have to face before they are executed or it could be that the environment

changes unexpectedly. To achieve this, Bredeche et al. proposed a distributed online optimisation algorithm that allows for the robots to self-adapt and is also able to withstand the pressure form the environment. From the problem, we can observe there is no need for an explicit evaluating function as we do not know what the goal is due to the dynamic environment. Traditional ER techniques involve performing the optimisation technique first before execution so an approach is needed to enable the optimisation to be performed online, which brings us to Embodied Evolution (EE) [73], [54].

Embodied evolution helps to solve the aspect problem that has to with the distributed online algorithm. In various setups [52], [56], [74], each individual robot in the swarm executes the evolution board and might even exchange individuals between each other if there are within range with themselves. However, EE requires a fitness function. Therefore they propose a distributed algorithm that is environment driven and also self-adaptable for a long period of time based on the evolutionary operators which takes into account the selection pressure from the environment. This can be illustrated as follows: we can classify an individual or genome as successful when it spreads across the total swarm which requires it to minimise the risk and also maximise the number of mating opportunities. The key to this approach is the implied fitness function which can be seen due to two motivations: a) a robot must be able to cope with the dynamic environment limitations, which is due to the interactions of the robot and the environment or other robots, so as to maximise survival b) as mentioned previously, the individual must spread across the population to survive. Therefore, the individuals are biased to produce efficient mating behaviours. The larger the number of robots that are met, the larger the opportunity for the robot to survive. An efficient environment driven algorithm must achieve a balance between these two motivations. That is, the individual spread should be maximum whilst also considering the survival efficiency.

Based on this, they introduce the mEDEA (minimal environment-driven distributed evolutionary adaptation). This algorithm describes how the evolution is done locally and is run simultaneously on all the robots in the swarm. It is also run with a communication routine that is used to receive incoming genomes from other robots and store them in a list for use at a later time. At a point in time, a robot would be driven by a program whose parameters are gotten from an 'active' genome which remains unchanged for one generation. This genome/individual is continuously broadcasting to other robots within its communication range. The algorithm was executed in two different setups: a) the "free-ride" setup where the swarm is put in an environment with few obstacles and the robots should be able to learn to avoid the obstacles and wander around to improve mating opportunities b) the "energy"

setup where food item (energy sources) are spread around environment and then can be collected by the robots. Each agent has an energy level which is based on the collected food and power usage.

This approach towards using an implicit fitness function proved successful although future work could be done in surviving a more complex environment which would result in more complex behaviours. This paper describes how in problems where the fitness function cannot be explicitly expressed, there are ways around that (implicit fitness functions). In this case, the environment is used to drive the evolution towards the best possible solution as the idea was for the robot to function in dynamic environments. There could be other ways to drive an evolutionary algorithm towards its global optima which would be useful for my thesis as trying to evolve fault recovery strategies in a swarm with a fitness function is a non-trivial issue. Novelty search is another approach that has been introduced in evolutionary techniques, which could be used rather than fitness function.

Christensen et al. [64] presents the novelty search as an evolutionary technique that differs from traditional evolutionary methods that uses fitness functions as a way to discover the best solution. Rather for this approach, individuals/genomes are scored based on how how novel it is rather than their quality based on a fitness function. They use NEAT in conjunction with novelty search to evolve the neural network controllers for swarm robots. This was conducted on two tasks: aggregation and sharing an energy recharge station. For fitness-based evolution, the evaluation of the individuals are done independently however in novelty search, the individuals are scored based on how different they are from the individuals that have been scored so far with respect to their behaviour. Due to this, the evolution does not get stuck at a local optima which is a problem usually associated with fitness based evolution. Novelty search has been successfully applied to various tasks [75], [76] and is able to provide capable and broad solutions to these problems. This paper studies the application of novelty search in the evolution of neural network controllers in swarm robotics. The reason is that swarm robots can be classified as a high level unit due to its multiple interacting sub-units and this causes it to usually have a deceptive fitness landscape [77].

In all experiments carried out by the authors, aggregation and sharing the energy recharge station, they compared the results fitness-based evolution with evolving using novelty search. One of the main aspects of novelty search is called the *novelty measure*. This “measures” the novelty of each solution and is based on a behaviour characterisation (a vector) that is linked to an approximation of the representation of the individual’s behaviour. This characterisation is dependent on the domain and task. It captures the macroscopic swarm-level behaviour and is therefore independent

of the swarm size. When implementing novelty search, it does not require a huge change from the traditional evolutionary algorithm apart from substituting the fitness function with a "domain-dependent novelty metric". This metric measures how different the individuals are from each other with respect to behaviour. Previous behaviours are stored in a file. This file is empty at first but new behaviours are added to this file only if they are different from what is already present in the file (the measure has to be above a predefined threshold).

An issue with the novelty search approach is that a lot of the effort could be spent exploring new but non-productive regions in the search space so some methods combine the exploratory attribute of the novelty search and the exploitation attribute of a fitness-based evolutionary approach [65].

The results show that in both the aggregation and energy station sharing task, novelty search was able to find broad, diverse and successful solutions earlier in the evolutionary process compared with a fitness based solution. Although both approaches performed similarly in the aggregation task, in the energy station sharing task, the fitness based evolutionary process was deceived and help up at a local maxima however novelty search was not deceived and was able to find appropriate solutions. The results also show that when the behaviour representation is directly related to the task bot excessively detailed, the evolution performs better and is more efficient.

Novelty search is an approach that can be successful used in evolutionary swarm robotics as displayed in this paper. It gives another perspective when looking at evolutionary procedures when it relates to robotics. Although fitness-based evolutionary processes are not without its problems, a possible improvement could be to add a fitness function with novelty search and the evolution would use each of them in different stages in the evolutionary procedures. Novelty search is a technique that could be useful in my thesis because of its ability to ability to produce diverse solutions however more research would have to be done in this area to see if it would be applicable in fault recovery.

2.8.7 Fitness-based Evolution

Some other work has been done in evolutionary robotics and although all these are fitness based algorithms, they are still relevant. They show that although there are some issues with fitness functions as discussed above, there have been successful results gotten using these set of algorithms. It just shows that some problems would work better with an evolutionary algorithms that are not fitness based. Francesca et al. [78] discusses a new approach towards evolving control behaviours for au-

tonomous swarm robots. Most swarms are designed by hand by trial and error. The individual level behaviour is written and tested until the desired global behaviour is achieved. This process is costly, time-consuming and not very consistent. This paper proposes automatically developing the robot swarm behaviours using evolution. In this paper, they used evolutionary algorithms to get parameters that feed into a neural network which maps the sensor readings of a robot into values that are fed into its actuators. They develop a new automatic design approach for robot swarms called AutoMoDe. It generates individual behaviour in the form of a random finite state machine by searching through the best combination of preexisting parametric modules. AutoMoDe develops this control algorithm by using an optimisation algorithm to select the arrangement of the random finite state machines, what modules that would be included and the value of the parameters. It should be noted however that by using these preexisting modules , AutoMoDe introduces a bias thereby reducing the representational power: they are constrained due to the fact the optimisation algorithms only use preexisting modules (a restricted search space). Due to this, it limits the possibility to fine-tune the dynamics of the robot-robot and robot-environment interactions. However, they show that if the modules are properly defined, this bias introduced reduces the variance and increases the control algorithm's ability to generalise without causing to lose its effectiveness. They also introduce another concept called AutoMoDe-Vanilla which is simply the application of the AutoMoDe approach that is used on an e-puck robot.

Dorigo et al. [79] is an example of the application of evolutionary robotics in the field of swarm robotics. This experiment dealt with evolving aggregation behaviours of the *s-bots* and also evolving the coordinated movement of the *swarm-bot*. A swarm-bot is composed of simple autonomous robots called *s-bots* that is capable of self-asembling and self-organising. They can connect and disconnect from each other. The experiments are run using a simplified version of the *s-bots* by using evolution to develop effective controllers that would be used for both aggregation and the coordinated motion. The *swarm-bot* has been tested on hardware and simulation. The *s-bots* have simple sensors, motors and also limited computational capabilities just like actual robots in a regular swarm but they have physical links to connect and reconfigure themselves to solve problems where single *s-bots* cannot solve the problems. The difficulty in the design of the controllers is the breaking down of the global behaviour into the individual components of the *swarm-bot*, which is one of the main reasons that evolution is used as it is able bypass this problem of finding the individual mechanism for the controller so as to get the desired emergent global behaviour.

Duarte et al. [80] describes a study proving that evolved controllers can be trans-

ferred to actual robots and achieve a performance similar to the performance during simulation. They evolve neural network-based controllers in simulation to perform some environmental monitoring tasks on ten simple, inexpensive aquatic surface robots such as homing, dispersion, clustering and monitoring. They conclude the study with a proof-of-concept experiment whereby the swarm performs a complete environmental task by combining the multiple evolved controllers enabling the robots to develop more sophisticated controllers.

Evolutionary algorithms generally provide new and diverse solutions to tasks that deal with optimisation. In swarm robotic systems, they aid in designing the individual controller code to give the desired global behaviour which has been proven in various examples that have been discussed above (aggregation, coordinated motion). EAs have been successfully used to evolve parameters in neural network controllers and also finite state machines as discussed in AutoMoDe. Examples have also been in both fitness based evolutionary process and non-fitness based evolutionary process. My thesis is based on the evolution of fault recovery strategies in fault recovery systems and the reason why i decided to go through the evolutionary route is because of its optimisation also its ability to give a desired global behaviour by searching through the search space. As the designer of the algorithm, i do not know the best way for the swarm to handle fault recovery strategies and hoe the individual robot controller code would be written to achieve the desired behaviour. Evolutionary algorithms would aid in breaking down the global behaviour into individual codes.

From the literature review, we have discussed the importance of having a fault tolerant system and the work done presently to achieve this. Fault tolerance is divided into sub-categories: fault detection, fault diagnosis and fault recovery. There are key lessons to be taken from the literature review:

- There is limited work done in fault diagnosis and fault recovery especially in swarm robotic systems.
- The majority of the work mentioned in the literature review is fault detection. There have been various methods that have been used in detecting faults such as using the artificial immune system as a source of inspiration, using the on-board simulator, monitoring sensor data etc.
- This is a limitation because it has been discussed that fault tolerance is an important part of robotics. As mentioned, so much work has been done in fault detection but that is not all that is needed to have a fault tolerant system. Most of the work done, after the fault has been detected, nothing is done in the robotic to discover what fault has occurred and how to get the robotic

system to recover from these faults especially when they have disastrous effects on the system. If fault tolerance (fault detection, diagnosis and recovery) is not done appropriately, the robotic systems break down and would not be able to complete the task that they have been assigned. This is the reason why there is a need to have a system that is completely fault tolerant, not just in detecting fault but also in diagnosing and recovering the faults.

- Also, it should be noted that in evolutionary swarm robotics, most of the work done involves evolving swarm behaviours and there has not been work in any sort of fault tolerance: fault detection, fault diagnosing and fault recovery capability. This is also a limitation as evolutionary robotics has not been applied to fault tolerance. From the analysis done in evolutionary robotics, a lot of novel solutions have been developed and successful when deployed on actual robots in the evolution of swarm behaviours. Evolution has the ability to create solutions that might not occur to a human designer and could optimise ideas efficiently.

Based on the literature review, we decided to do some research into this area to explore using evolutionary techniques in a fault tolerant context using the ω-algorithm as a case study.

Chapter 3

Preliminary Results

3.1 Methodology

We are presented with the problem of trying to evolve a fault tolerant strategy in a swarm of robots performing a collective phototaxis behaviour. We injected complete motor faults but left their sensors functional. 'Anchoring' occurs and prevents the swarm from completing the task. It is discussed in the previous chapter that the initial results failed so therefore, moving forward, I would be attempting to fix this and get this fault tolerant system working. That is, i want to prevent the anchoring problem from occurring when robots have partially failed and allow the robots to complete the task. This is a fault tolerant strategy for one type of fault and one specific swarm behaviour; however, i would be building on this experiment for this PhD by expanding it to doing actual fault recovery, moving towards other types of faults and swarm behaviours.

This section introduces the swarm behaviour algorithm (collective phototaxis) and various aspects of the evolutionary algorithm that is used.

3.1.1 The Collective Phototaxis Algorithm.

So far, i have attempted to evolve a fault recovery strategy for the swarm phototaxis algorithm in ARGoS [81]. ARGoS is a robot simulator. We injected complete motor failure faults to a robot swarm size of 15. The arena size is 20x20m^2 and the robots are initialized at the center of the arena and the robot orientation is set randomly in range [-π, π] radians. We make use of foot-bots [82] that have been modelled in ARGoS and the maximum speed each robot is 10cm/s. The robot has a coloured blob omni-directional camera sensor that has been set to have an aperture, the viewing

range of the robot, of 80cm. At the start of the experiment, the swarm is randomly positioned within a 1.4x1.4 m^2 region at the centre of the arena.

Each robot has light sensors that are used to 'see' the beacon, which emits a yellow light. The robot also has omnidirectional cameras that are able to see all colours around however it is set to filter out colours except the colour each robot emits: red. Each robot also has four actuators: wheels with differential steering. Each robot gets the light reading and then calculates a normalised vector, which comprises the distance of the robot to the beacon and the angle of the robot to the beacon. The vector is normalised with magnitude 0.25 times the maximum speed (a predefined value), that points towards the beacon.

To calculate the cohesion vector, which is the normalised Lennard Jones interaction force (this is a mathematical function that approximates the interaction and repulsion forces between the robots) and angle between each robot, the robot goes through all the camera readings which is used to calculate the vector of interaction between the robots. The camera only considers red blobs (other robots) and the robot only considers the closest neighbours so as to avoid the attraction of the far ones, which is determined by the target distance. The blob distance and angle is recorded and these are used to calculate both the Lennard Jones interaction force, which in turn is used to calculate the normalised cohesion vector. We evolved the parameters of the Lennard Jones function and the hard turn, soft turn and no turn turning threshold angles of the robots. They interchange between these angle parameters depending on the switching conditions.

3.1.1.1 The Lennard Jones Function.

For the attraction and repulsion between the robots, the swarm phototaxis algorithm uses the Lennard Jones mathematical function to calculate the Lennard Jones potential. The Lennard Jones potential is a mathematical function that approximates the interaction between two atoms. There are three constants that are used in the calculation of the interaction force: gain, exponent and the target distance. The target distance is the maximum distance between two entities, in our case, foot-bots, before either attraction or repulsion occurs. If the robots are at a distance less than the target distance then they repel each other and vice versa. This is shown in detail in Fig.1.

Both the cohesion and light vectors are added together and are used to set the wheel speeds. What is used to consider the wheel speed are the turning angles: HARD_TURN, SOFT_TURN and NO_TURN. The wheel speeds are based on the current turning state. If the current turning state is NO_TURN, then the robot sim-

ply goes straight while if the turning state is SOFT_TURN, both wheels go straight however one wheel moves faster than the other. The speed is calculated by multiplying the current speed by 2 and also a small variable to keep the value less than or equal to the maximum speed. If the turning state is HARD_TURN, then both wheels have opposite wheel speeds. The added normalised vectors components are made up of length (speed) and the angle. This angle is used to determine what turning state is required while the length is used to calculate the speed of the robot.

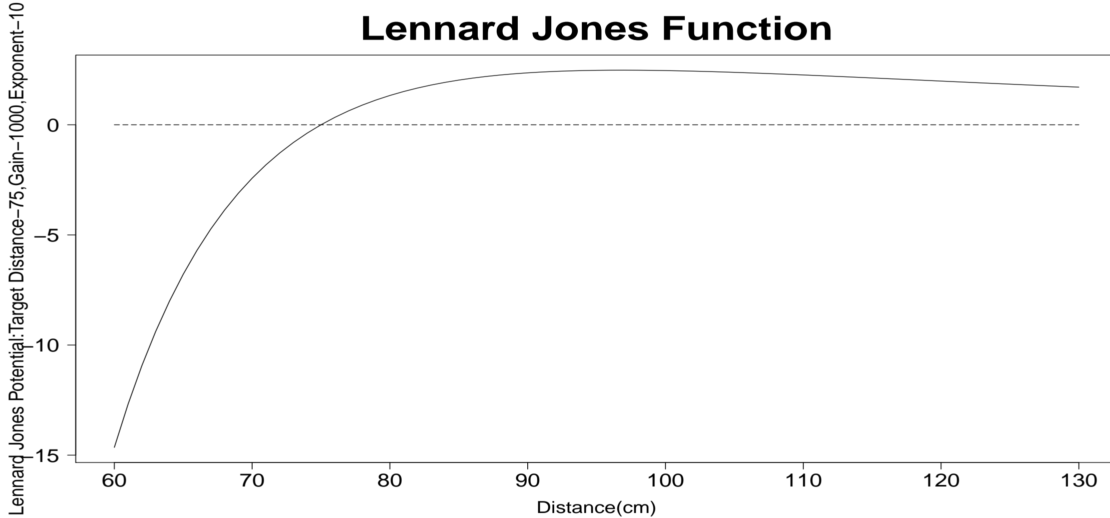


Figure 3.1: Lennard Jones Function using the Predefined Values

The parameters we decided to evolve are the constants of the Lennard Jones potential: gain, exponent and target distance; we also decided to evolve the turning state angles. The reason behind this is that when the functioning robots are anchored, they can escape it with the combination of the right interaction force and also the right criteria that allows the robot to perform NO_TURN, SOFT_TURN and HARD_TURN turning states.

3.1.1.2 Random Beacon Positions.

A new light position is initialised every fitness evaluation. The first step in initialising the light position is to check if there is already a light entity in the arena. If there is, this entity has to be removed to create a new one. An attribute that enables the light position to be changed is the random seed value. However, once the random seed has been set, the random number generated is fixed. Therefore, to create a new

light position, we have to use random numbers for the random seed attribute. This is done by creating a new random number generator (RNG) for that category. We want the beacon to be initialised from a set radius from the centre of the arena. We set the position of the beacon by converting spherical coordinates to vector coordinates. The radius of the circle is 7.0m, the angle between the z axis and the vector is 90 degrees. The angle on the xy plane is being randomised by dividing 360 degrees by a random value, between 1 and 90, each time the light is initialised. A new light entity instance is created and the vector coordinates that have been converted from the spherical coordinates are passed into this instance. The colour of the beacon is set to yellow and the intensity is set to 3.0. The reason why we randomise the angle of the beacon is so that the genetic algorithm does not overfit.

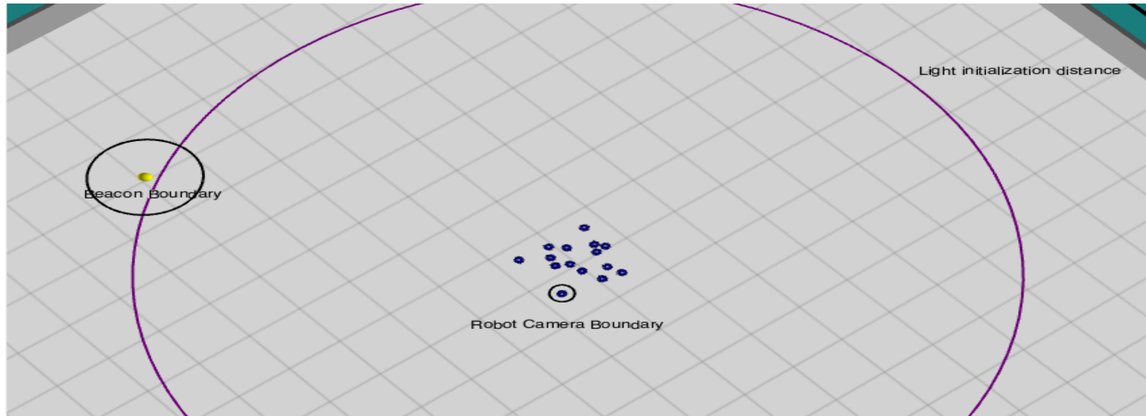


Figure 3.2: The arena shown from above. The Light initialisation distance has a radius of 7m from the center of the swarm and the beacon boundary has a radius of 1.5m from the beacon

From figure 2, we can observe the swarm initialised at the centre and the boundary around the beacon that signifies the distance around the beacon that is used to determine the number of robots that made it to the beacon. This figure also shows the radius which the beacon position is initialised.

3.1.2 The Evolutionary Algorithm

The algorithm used to evolve the Lennard Jones parameters and turning state threshold angles is a simple genetic algorithm. Genetic algorithms are heuristic search algorithm inspired by Darwinian evolution [83]. As we have discussed earlier, this experiment is run on the ARGoS simulator. We uses the GALIB [84] evolutionary library as an example, which is also written in C++. There are six variables that are being evolved; the turning states and Lennard Jones function parameters. The genome ranges for each of these parameters are shown below:

The turning state angles have to be in a specific order: HARD_TURN >SOFT_TURN >NO_TURN angle. To allow the genetic algorithm to evolve these parameters without affecting this order, the mutation, crossover and initialisation operators have to be customised. Each genome is evaluated by the fitness evaluation and for each of these fitness evaluation, three trials are run. The three trials are run and the worst performance(fitness) is taken as the final value. How the fitness is calculated

is described in the next subsection.

3.1.2.1 Fitness Function.

The fitness of the genome is evaluated by calculating the distance of the centre of the swarm to the beacon, from the final position at the end of the experimental run. We make the assumption that the algorithm already knows the robots are faulty as we are dealing with fault recovery at the moment. We go through all the healthy robots, and calculate the centroid of the healthy robots at each time step. We also calculate the number of robots that are within the beacon boundary. This determines how many robots made it to the beacon, that is, how many robots successfully complete the task. We ran the experiment, with no faults, using the predefined values. Thereafter, we measured the distance from the beacon to the edge of the swarm when the robots make it to the beacon. This value is the radius that we used to determine if the robots make it to the beacon.

3.1.2.2 The initialisation function.

The initialisation function is structured as follows: each parameter is randomly randomised. The HARD_TURN angle is randomised first from 10 to 180 degrees then the SOFT_TURN angle is randomised in a loop till it satisfies the condition that its value is lower than the HARD_TURN angles', from between 0-180. The same process is applied to initialise the NO_TURN angle. The Lennard-Jones potential parameters are randomised as normal. The range of the genes are specified already.

Gene Name	Range
Hard Turn Angle Threshold(Degrees)	10 - 90
Soft Turn Angle Threshold (Degrees)	5 - 70
No Turn Angle Threshold (Degrees)	5 - 10
Target Distance (m)	10 - 100
Gain	100 - 1000
Exponent	2 - 10

Table 3.1: Genome Ranges where HARD_TURN angle is a sharp turn, SOFT_TURN is a softer turn and NO_TURN means the robot goes straight. Target Distance, Gain and Exponent are parameters in the Lennard Jones Potential equation that is used to calculate the level of interaction between the robots in the swarm

Table 3.2: GA Parameters

Evolution Data	Value
Size of Genome (Number of Parameters)	6
Population Size	50
Generation Number (Population Iteration)	20
Crossover Probability	0.45
Mutation Probability	0.2
Number of Trials	3
Number of Runs	30

3.1.2.3 The crossover function.

We create a new class that inherits GSimpleGA from the GALIB library and override the default step function; GALIB is a C++ genetic algorithm library that contains tools for doing genetic algorithm optimisation. We are doing single child crossover while the default is two children crossover. After each crossover, we check if the order of the turning angles are kept. If the order has not been maintained, keep trying to execute a crossover till the genes are in the right order. After the crossover occurs, it is added to the population and the worst individual is deleted. It is a steady state genetic algorithm where only a few individuals in the population are replaced every generation.

3.1.2.4 The mutation function.

The mutation function undergoes a similar process. When the GA is mutating a genome, it runs through each gene of the genome. At each gene, we get the range of the gene and use it to calculate a mutation variable. The pseudocode is mutated_gene = gene_value + randomGaussian(mean = 0, standard deviation = gene_range * 0.10). After each mutation, we check if the order of the turning angles are kept. If the order is not kept, then keep mutating till the genes are in the right order.

3.2 Results

We evolved the controller with zero, two, five and nine faults present in the swarm. The controller with zero faults present in the swarm would act as the baseline while other controllers would be compared to it to observe the difference, if any. Thereafter,

we tested these evolved controllers against a range of faults from zero to nine present in the swarm. From each test run, we obtained the number of robots that made it to the beacon and we used it to the analyse how the evolved controller fared. We made use of two tests for analysis and compare the results: Mann-Whitney U Test and Vargha-Delaney A-test.

The Mann-Whitney U Test or the Wilcoxon rank sum test is a non-parametric test used when asked to compare the means of two groups that do not follow a normal distribution [85] . It calculates the probablilty p, that two samples are gotten from the same distribution. We use the null hypothesis which is a term used in statistics to describe when there is no significant difference between specified samples, that is, they have the same statistical properties: same medians, quartiles etc. The rank sum test calculation is done in R. If $p < 0.05$, we can reject the null hypothesis because the medians are different. However, it should be stated that if $p > 0.05$, it doesn't mean that we have to accept the null hypothesis. The Vargha-Delaney A test is a non-parametric effect magnitude test that can differentiate between two samples of observations [86] . This test is a measure of importance, scientific significance between two samples. A lies between 0 and 1. if A is 0.5, then there is no effect, that is, both samples have the same median. If A is 0.71, then there is a large effect and the effect steadily reduces till it gets to 0.5. On the the other side of the graph, from 0.5, the effect steadily increases as we descend to 0.29.

Table 3.3: Rank Sum and A Test(Between the 0 fault evolved controller and 2 faults evolved controller), Median and InterQuartile Range (IQR) of the number of robots that made it to the beacon

Scenario	P	A	Median ± IQR (0 Fault)	Median ± IQR (9 Faults)
Zero Faults	0.114	0.384	83.33 ± 26.67	86.67 ± 20
One Fault	**0.019**	**0.329**	**85.71 ± 21.43**	**92.86 ± 14.29**
Two Faults	**0.026**	**0.336**	**84.67 ± 15.39**	**84.62 ± 7.69**
Three Faults	0.525	0.547	87.50 ± 25	83.33 ± 25
Four Faults	0.626	0.537	81.82 ± 18.18	72.73 ± 27.27
Five Faults	0.708	0.528	70 ± 20	70 ± 30
Six Faults	0.238	0.588	66.67 ± 44.45	55.56 ± 33.34
Seven Faults	0.786	0.479	50 ± 25	50 ± 25
Eight Faults	0.423	0.441	42.86 ± 42.86	57.14 ± 28.57
Nine Faults	0.526	0.453	50 ± 33.34	50 ± 33.34

We can observe from the rank sum test, there is a statistical difference at one and two faults as the value of p is less than 0.05 therefore we can reject the null hypothesis. However as the faults increase, the probability increases and although we cannot say that we accept the null hypothesis, we can say that there is not much, if any, statistical difference between the two samples. From the A test, we can observe, for the data that has statistical difference, there is a medium effect between the two samples. We can conclude that the evolved controller with two faults present in the swarm fares worse in general than the evolved controller with no faults present in the swarm.

We can conclude that the evolved controller with five faults present in the swarm fares worse in general than the evolved controller with no faults present in the swarm. We can observe from the rank sum test, there is a statistical difference at only zero fault as the value of p is less than 0.05 therefore we can reject the null hypothesis. However as the faults increase, the probability increases and although we cannot say that we accept the null hypothesis, we can say that there is not much, if any, statistical difference between the two samples. From the A test, we can observe, for the data that has statistical difference, there is a large effect between the two samples as the A value is greater than 0.7.

We can conclude that the evolved controller with nine faults present in the swarm fares worse in general than the evolved controller with no faults present in the swarm. We can observe from the rank sum test, there is a statistical difference at zero, one, two and three faults as the value of p is less than 0.05 therefore we can reject the null

Table 3.4: Rank Sum and A Test(Between the 0 fault evolved controller and 5 faults evolved controller), Median and InterQuartile Range (IQR) of the number of robots that made it to the beacon

Scenario	P	A	Median ± IQR (0 Fault)	Median ± IQR (9 Faults)
Zero Faults	**0.005**	**0.708**	**83.33 ± 26.67**	**73.33 ± 13.33**
One Fault	0.125	0.614	85.71 ± 21.43	78.57 ± 14.28
Two Faults	0.099	0.621	84.67 ± 15.39	76.92 ± 15.38
Three Faults	0.041	0.652	87.50 ± 25	75 ± 16.67
Four Faults	0.176	0.601	81.82 ± 18.18	72.73 ± 18.18
Five Faults	0.667	0.468	70 ± 20	70 ± 20
Six Faults	0.523	0.548	66.67 ± 44.45	66.67 ± 22.23
Seven Faults	0.096	0.377	50 ± 25	62.50 ± 25
Eight Faults	0.228	0.411	42.86 ± 42.86	57.14 ± 28.57
Nine Faults	0.054	0.358	50 ± 33.34	50 ± 50

Table 3.5: Rank Sum and A Test (Between the 0 fault evolved controller and 9 faults evolved controller), Median and InterQuartile Range (IQR) of the number of robots that made it to the beacon

Scenario	P	A	Median ± IQR (0 Fault)	Median ± IQR (9 Faults)
Zero Faults	**0.003**	**0.716**	**83.33 ± 26.67**	**73.33 ± 13.33**
One Fault	**0.001**	**0.747**	**85.71 ± 21.43**	**71.43 ± 14.29**
Two Faults	**0.002**	**0.728**	**84.67 ± 15.39**	**69.23 ± 7.69**
Three Faults	**0.010**	**0.691**	**87.50 ± 25**	**75 ± 16.66**
Four Faults	0.068	0.615	81.82 ± 18.18	72.73 ± 27.28
Five Faults	0.305	0.576	70 ± 20	70 ± 30
Six Faults	0.496	0.551	66.67 ± 44.45	66.67 ± 22.22
Seven Faults	0.557	0.456	50 ± 25	50 ± 22.50
Eight Faults	0.429	0.441	42.86 ± 42.86	57.14 ± 28.57
Nine Faults	0.214	0.408	50 ± 33.34	50 ± 33.34

hypothesis. However as the faults increase, the probability increases and although we cannot say that we accept the null hypothesis, we can say that there is not much, if any, statistical difference between the two samples. From the A test, we can observe, for the data that has statistical difference, there is a large effect between the two samples as the A value is greater than 0.7 though at three faults, there is medium effect.

We are presented with the problem of trying to evolve a fault recovery strategy in a swarm of robots performing a collective phototaxis behaviour. We injected complete motor faults but left their sensors functional. From the results, we were able to ascertain that simulating using the evolved solution with no faults present in the swarm provides a better result compared to the other evolved controllers (2, 5 and 9 faults) which signifies that this experiment failed. This leads to the the proposed methods on how this problem would be solved and also future experiments that would produce more efficient results and strategies.

These experiments and solutions are presented in the next chapter where we discuss approaching the recovery problem from another angle. In the next part, learning is discussed as the new approach to be considered for the fault recovery process and the results following the implementation of the learning approach.

Part II

Learning Approaches and their Implementations: Fault Recovery Solution (Centralised and Distributed)

How do we define learning, especially in regard to how it is being applied in this thesis? Learning is defined as changes in the individual's 'mental' model or knowledge that represents a specific area in the learners 'brain'. It's more of a change in knowledge structure not in behaviour per se; at the end of the learning process, the individual should have a change in knowledge. For this thesis, we approach learning from a machine learning point of view utilising reinforcement learning algorithms and also self-organising maps to create an optimal learning process for the fault recovery problem that we aim to solve especially when introducing predefined pre-learning behaviour in a given environment.

There are various learning techniques that are present in literature from neural networks to deep learning; where deep learning is one of the most popular learning techniques today. Deep learning is one of the leading research areas in Artificial Intelligence that utilises Artificial Neural Networks to learn unstructured data in an unsupervised form. It is modelled after the human brain and is capable of dealing with complex learning situations, similar to but not quite like the human brain as the human brain is the most complex 'computer' in the world. Due to its ability to train complex data and information that needs not be supervised and learn in complex scenarios, deep learning can be a contender when selecting the appropriate learning strategy. It should be noted that deep learning is not the be-all and end-all learning technique; some learning situations would not work appropriately with it. This argument is discussed more in this chapter.

Chapter 4

Learning Techniques

4.1 Reinforcement Learning

Reinforcement learning (RL) involves learning which steps to take to achieve a goal, by interacting with the environment [87]. The learner, in this case the robot, is not told what to do but it must discover which steps to take based on those which result in the highest reward. In order to learn this, the robot takes an action, and is then 'scored' on its performance. Reinforcement learning uses *states, actions, rewards* in their simplest form. Any method that is suited to solving problems presented in this form is considered to be a reinforcement learning method. Temporal Difference Learning (TD) and Q-learning [87] are particularly powerful reinforcement learning methods.

One of the main issues that arises in reinforcement learning is finding the right balance between exploration and exploitation. Exploration involves evaluating more of the action space. The aim of RL is for the agent to obtain a high reward by performing the action, however for this to occur, the learning agent must perform actions that have not been tried before lest the action with the highest reward has not been tried yet. Exploitation involves trying an action that has already been experienced and has a high reward. The agent has to be able to exploit experienced actions but also has to explore the action space so as to make more efficient selections later. The agent cannot exclusively use either exploration or exploitation.

The learning agent must explore the action space and also continuously try the best action. At each time step t, the agent receives a representation of the environment state, $S_t \in S$ and maps it to an action, $A_t \in A(S)$, selected using a learnt probability distribution. This mapping is called a policy, π_t, where $\pi_t(S_t,\ A_t)$, represents the probability of selecting action A_t when in state S_t at time step t, based on

Algorithm 1 Pseudocode Q-Learning

```
1: procedure Q-LEARNING ALGORITHM
2:     Initialise Q(S_t, A_t), ∀ S_t ∈ S, A_t ∈ A(S), arbitrarily
3:     for <each run> do
4:         Initialise S_t
5:         for <each step> do
6:             Implement action A_t, given by π_t(S_t, A_t). Observe R_{t+1}
7:             Select subsequent state S_{t+1}
8:             Update the Q-value in Q-table
9:             S_t ⇐ S_{t+1}
```

a trade-off between exploitation and exploration. One time step later, the agent receives a reward $R_{t+1} \in R \subset \mathbb{R}$ based on how the agent performed in the environment before moving on to the next state S_{t+1}.

For each possible state, each action must be tried many times to obtain a reliable estimate of the expected reward. The ϵ variable defines the exploration factor, where $0 \leq \epsilon \leq 1$. The agent selects a random action with a probability of ϵ and exploits the best action as decided by $\pi_t(S_t, A_t)$ with a probability of $1 - \epsilon$. The value of ϵ must be selected so as to allow a good trade-off between exploration and exploitation.

In evaluating how good a selected action is for a selected state, a value function, $V(S_t, A_t)$ is defined which maps the state-action pair to a value that represents the reward. As the agent interacts with the environment, the value function is updated, showing how well the action chosen progresses based on the rewards calculated. Q-learning [87] is a subset of the Temporal Difference learning method, whereby an agent takes an action in a specific state, and the performance of the state-action is calculated as the reward. The simplest TD method updates the value function as:

$$V(S_t) \Leftarrow V(S_t) + \alpha(R_{t+1} + \gamma(V(S_{t+1})) - V(S_t)) \tag{4.1}$$

where α is the learning rate, γ is the discount factor which determines how far into the future rewards are considered. To consider immediate rewards, γ must be closer to 0. It is updated with the reward obtained from the action and the discounted value that is expected in the next state.

Q-learning uses a value table called a Q-table, where the entries are called Q-values. Each entry represents the maximum reward from the state-action pair implemented. Selecting the best behaviour simply involves selecting the action that has the highest Q-value for that specific state. The Q-value is updated as follows:

$$Q(S_t, A_t) \Leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma(Q_{max}(S_{t+1}, A_t)) - Q(S_t, A_t)) \tag{4.2}$$

Reinforcement learning in robotic systems is typically used in path or task planning [11], [12], [13]. For each possible state, an action is chosen and a reward is allocated based on how successful that action is for that state. The state, action and corresponding reward is stored in a lookup table. However, robotic systems exist in continuous spaces, while reinforcement learning is traditionally implemented for discrete spaces. It is thereby necessary to extend the typical reinforcement learning architecture to allow for learning to be done on continuous state variables.

There are two ways to deal with continuous state or action spaces in reinforcement learning:

- The Q-value function is approximated for the tested state or action spaces. A Q-table is not used; rather a function (linear, polynomial or a neural network depending on what learning is being done) is used to represent the Q-table and the Q-value function. This is called function approximation. It should be noted that function approximation in reinforcement learning is also used for more complex or dynamic environments.
- The dimensionality of the space or action space is reduced.

Function approximation is a powerful way of generalising large state spaces that is larger than the available memory or computational resources. Utilising this in reinforcement learning is a form of supervised learning where the states and actions are fed into a function (neural network, linear, quadratic) that computes their values and trains the function. With this, the function learns from a small subset of states provided and it attempts to generalise for a larger subset of states by estimating the Q-value which is used to determine the optimal solution to a problem.

4.2 Deep Learning

Deep Learning (DL), a type of neural network, uses this concept of function approximators but rather than using linear or quadratic functions, deep learning makes use of neural networks to approximate the action-value function (Q-table) [88]. Deep Learning allows machines and systems to mimic the human brain to make autonomous decisions based on learnt experiences. Due to this, it is recognised as one of the leading research areas in Artificial Intelligence (AI).

Deep Learning has gotten a lot of publicity in recent years and has been shown to beat human experts in many games such as Atari as can be seen in Figure 4.1 [89]. They have also been used in speech recognition programs such as Siri, Google Speech and Amazon Alexa as well where people are able to ask AI for different things and

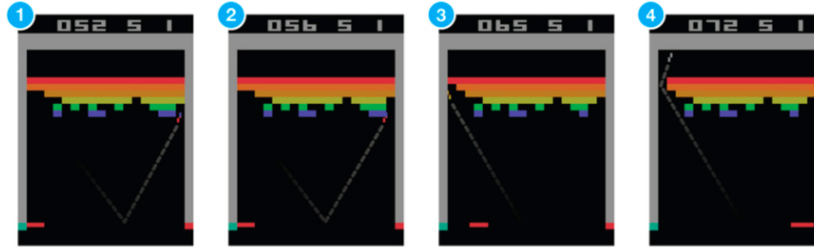


Figure 4.1: Playing Atari Games using Deep Q-Learning [1]

they are able to interpret what has been said and make decisions to do or not to do certain actions based on the input that has received [90].

Due to the neural networks used, deep learning is able to learn via what we can call an hierarchy of categories [91]. The neural network architecture of the deep learning is made up of hidden layers where the deep learning algorithm learns these categories from low-level categories such as letters and moves to higher levels such as words before being able to recognise sentences and so on.

The Deep learning narrative has been pushed in recent times especially in complex learning scenarios especially when it comes to learning and for good reasons; there have been studies and reports on how deep learning has improved how learning is achieved and the results of the learning process when compared to traditional machine learning techniques [91] [89] [92].

Deep Learning differs from traditional Machine Learning algorithms in different ways; at the core of these algorithms, they are still quite machine-like where you have to be a domain expert so as to represent the domain (input space) accurately and sometimes human intervention in areas where it is not completely autonomous in its decision making [91].

One of the advantages of deep learning is that it depends on a lot of data (this is also a disadvantage which is discussed later in this chapter). Depending on the application of a deep learning technique, with the advancement of technology, we have the ability to produce/provide copious amounts of data which can be used in the deep learning infrastructure. The data often has been collected over many years. For example, when Google, Amazon or Apple are training their speech recognition system, they have to record many words, spoken in many dialects and accents so as to allow for the variations in the word and they have access to these kinds of data. Each of these speech recognition systems is continuously learning where any word spoken to them is kept forever and also part of the learning process where it adds to the data used in the training phase [93].

Additionally, deep learning eliminates the need for domain expertise where an expert is needed to break down the complexity of the input data to something more suitable to what the traditional learning algorithm is able to use. With deep learning, the algorithms learn from low-level abstraction to an increasingly step by step higher level of abstraction which has been discussed earlier [91]. Deep learning generally performs better when there is a lack of domain understanding which significantly reduces feature engineering whereas traditional machine learning techniques depend heavily on 'feature engineering' [94]. Feature engineering is the process of using domain knowledge of the available dataset by breaking it down to suitable features that allow traditional machine learning to function appropriately. A feature can be defined as an attribute that can be used to solve the proposed problem, that is, the better the feature representation, the better the learning result. This process could be difficult and time-consuming; trying to figure out how the input data would be represented. There are steps to the processing feature engineering [95]:

- Reflect and Test the features
- Make a decision of what features to create
- Create these features
- Check how well features fare with present learning algorithm
- Improve features as needed
- Continue to reflect and create features as necessary

Another aspect of Deep Learning that differs from traditional machine learning is how it solves the problem that is presented. Deep learning usually solves problems 'end to end' where for example, in a object identifying problem, deep learning would take an image as the input and would output the names of all the objects in the image at once. However, in traditional machine learning algorithms usually needs the problem broken down first and then their results are combined at the end; considering the same object identifying problem, where the image has to be broken down and the results are given in parts and then combined [91].

Deep learning is one of the major advancements in machine learning that has been applied to complex problems such as games like Go, Atari and also in speech recognition systems as seen in Apple' Siri, Amazon's Alexa and so on, where we can attribute most of the remarkable improvements done in these systems to deep learning. Additionally, recent deep learning algorithms has been developed to a point

which makes them run faster before which allows more data to be used during the training phase. With recent breakthroughs and future developments, deep learning would play a huge part in a lot of learning problems [91] [92].

However, when it comes to determining what learning method to use, traditional machine learning procedures or deep learning, it depends on the type of problem that is to be solved [90]. There is no perfect machine learning algorithm that works for all problems; different aspects needs to be thought about from the state representation to how optimal actions is to be selected to the complexity of the problem. As powerful as Deep Learning seems to be, there are some general disadvantages and it would be explained as a justification as to why deep learning is not used in this thesis. This thesis talks about fault recovery in swarm robotic systems where the each robot in the swarm is small and basic in terms of computational power; it cannot handle heavy computational processes. It also has limited attachments because the idea is that each robot in the swarm is simple like an ant in a swarm of ants where each ant is simple but by working together they are able to accomplish a lot of things, which has been discussed earlier in the literature review (the concept of swarm intelligence).

Machine Learning can be very powerful however it requires massive amounts of data for a difference to be seen (for deep learning to be relevant) between traditional machine learning algorithms and deep learning. It would require millions of samples. We do not have access to these huge amounts of data that would take massive amounts of time to train. In this thesis, we do have access to a large data set however, compared to other deep learning algorithm training data, the dataset available is not large enough. The reason for this is because due to the concept pre-fault learning where the idea is that as the robots start working on their tasks, the learning of the recovery strategies commence. Additionally, we do not have millions of input samples to train as can be seen in the results section where the algorithm is able to converge after about 2000 episodes. Deep learning usually takes much longer and requires more steps before it is able to make accurate decisions [96] [97]. Therefore, it is not appropriate to solve this specific problem.

Another issue with implementing Deep Learning is that it requires massive high computational power for this specific scenario is that deep learning techniques need to have high end infrastructure to train the neural network [91]. As stated earlier, deep learning is not the magic answer to every complex learning problem; traditional machine learning algorithms outperform certain learning problems especially when the training data is not large enough [97].

These are the major reasons why Deep Learning is not used in this thesis when attempting to solve fault recovery in swarm robotic systems. It needs a huge amount of data that in turn would require massive computational power. For the level of

learning that is being done, Deep learning might not necessarily outperform traditional machine learning algorithms.

As stated earlier, a function approximator can be used to deal with continuous state or action spaces however, this thesis' approach uses self-organising maps rather than a function approximator to solve this problem by clustering similar states to produce a discrete set. Function approximators could be a possible future work to allow for more complex and dynamic environments that the swarm robots would realistically encounter.The next section introduces self-organising maps, how they function and how they would be implemented for this thesis' application.

4.3 Self-Organising Maps

Self Organising Maps (SOMs) [98] are a type of artificial neural network (ANN) that provide a way of representing multidimensional data in smaller dimensional spaces. They typically consist of a two dimensional grid of neurons and unlike regular ANNs they apply competitive learning. They are trained using an unsupervised method to produce a discretised representation of the input space of the training samples, called a map. This can be seen in Figure 4.2.

Each unit in the SOM has a specific topological position and contains a vector of weights of the same size as the input vector. That is, for each unit u, the weight vector is:

$$w^u = [{w_1}^u,\ {w_2}^u, \ldots,\ {w_D}^u]$$

where D represents the dimensionality of the input vector. The SOM does not need a target output, rather where the unit weights match the input vector, the map is optimised to match the input data. It automatically finds structure within the data such that similar inputs are paired closer together. For any input vector $x = [x_1,\ x_2, \ldots,\ x_D]$, the distance between each unit t of the SOM and the input vector is given by:

$$\sum_{d=1}^{D}(x_d - w_d^u)^2 \tag{4.3}$$

This is simply the squared Euclidean distance. The unit with the shortest distance is the closest match to the input vector and is therefore the *winner unit.* The weights of the winner unit are updated as:

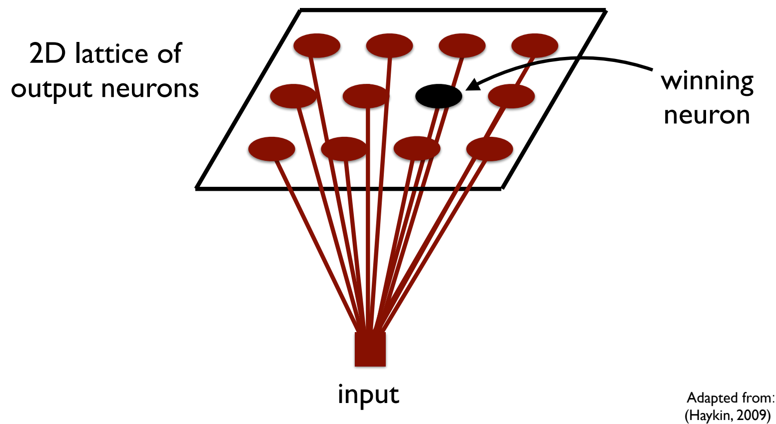


Figure 4.2: Two-dimensional Representation of Self Organising Maps [1]

$$w^{\text{winner}} = w^{\text{winner}} + \beta(x - w^{\text{winner}}) \tag{4.4}$$

where β represents the learning rate of the SOM. However, not only the winner unit is updated, the neighbouring units are updated as well, but the amount change to their weights decays the further they are from the winner unit. All weights are initialised at random and the learning is iterated for all the input vectors. The network is able to distribute the input vector information appropriately within the input space of the map.

Chapter 5

Centralised Approach to Fault Recovert

5.1 Empty Environment

This section describes a novel approach to fault recovery that involves the intelligent selection of predefined recovery strategies where the learning is done in an empty environment. In this section, the current architecture of the fault recovery process uses a global observer to calculate the system state and select the most appropriate recovery action, but the next section extends this to a decentralised architecture where each robot in the swarm makes decisions based only on local information that they receive from the environments and other robots in its immediate vicinity. It is assumed that the swarm robotic system is already capable of detecting and diagnosing faults so that the focus is mainly on recovering from faults. These recovery strategies cover faults enumerated by [5] that commonly occur in swarm robots.

From the point at which a fault has been detected and diagnosed, the swarm must decide upon an appropriate recovery strategy. We assume that each robot has the ability to repair other robots in the swarm, therefore the problem reduces to choosing which non-faulty robot(s) should be recruited to repair the faulty robot, and which predefined behaviour is most appropriate given the current scenario. The most appropriate recovery strategy will depend on a number of factors, such as proximity to the faulty robot or remaining battery power, thus some method of assessing the quality of a strategy and its future effect on the swarm is required. I present a solution to this problem that uses machine learning techniques to inform decisions at run-time based on the results of offline training. It should be noted that the current fault recovery architecture discussed in the section is intended as a proof-of-

principle that demonstrates the value of reinforcement learning for the selection of fault recovery strategies.

As stated earlier, we are utilising reinforcement learning and self organising maps in this fault recovery approach. Q-table (a property of reinforcement learning; q-learning) is usually represented as a two-dimensional table but in this approach, as the two possible actions sets that are different from each other, how the state-action pairs are stored in the Q-table are different from the convention. It is represented as more of a three-dimensional table where the states, action set 1 and action set 2 are stored which can be seen in Figure 5.1.

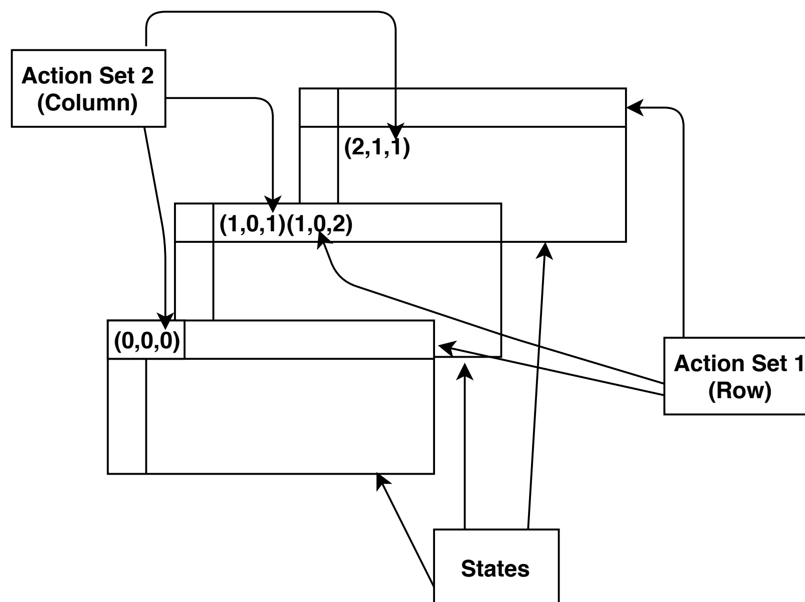


Figure 5.1: This figure describes the three-dimensional representation of the Q-table for this specific reinforcement learning technique. As can be seen, it is different from the traditional representation of the Q-table which is represented as a two dimensional table.

The method proposed here allows robots to learn how to select the most appropriate recovery strategy for any given system state or task. This approach is referred to as 'pre-fault learning'. This involves learning predefined recovery mechanisms for different possible swarm states before a fault occurs. The swarm's state is defined by the distances of the nearest three robots closest to the faulty robot, their energy levels, the level of importance of the faulty robot, how busy the nearest three robots are, and the distance of the faulty robot to a repair station [d_1, d_2, d_3, b_1, b_2, b_3, p_1, p_2, p_3, I, d_{rs}]. These values represent the input of the learning algorithms that is

used to learn the best action for any possible state.

5.1.1 Learning the Best Recovery Strategy

To test the proposed solution towards fault recovery, we use Autonomous Robots Go Swarming (ARGoS) [99], a widely used swarm robotics simulator. A swarm of 10 foot-bots (a particular configuration of modules based on the marXbot robotic platform [82]) is simulated in a 10x10m arena free from obstacles, undertaking case study behaviours of collective phototaxis, aggregation and foraging.

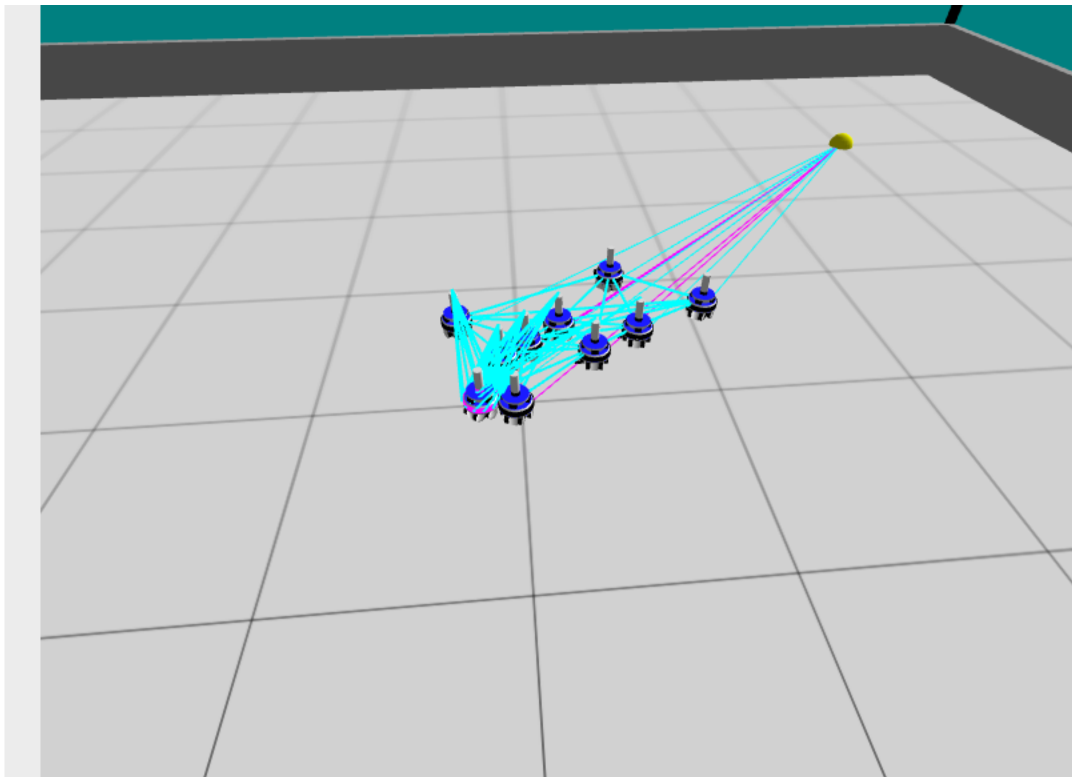


Figure 5.2: This figure represents one of the tasks, collective phototaxis. This is the ability for a robot swarm to sense a light source in the environment and collectively move towards it.

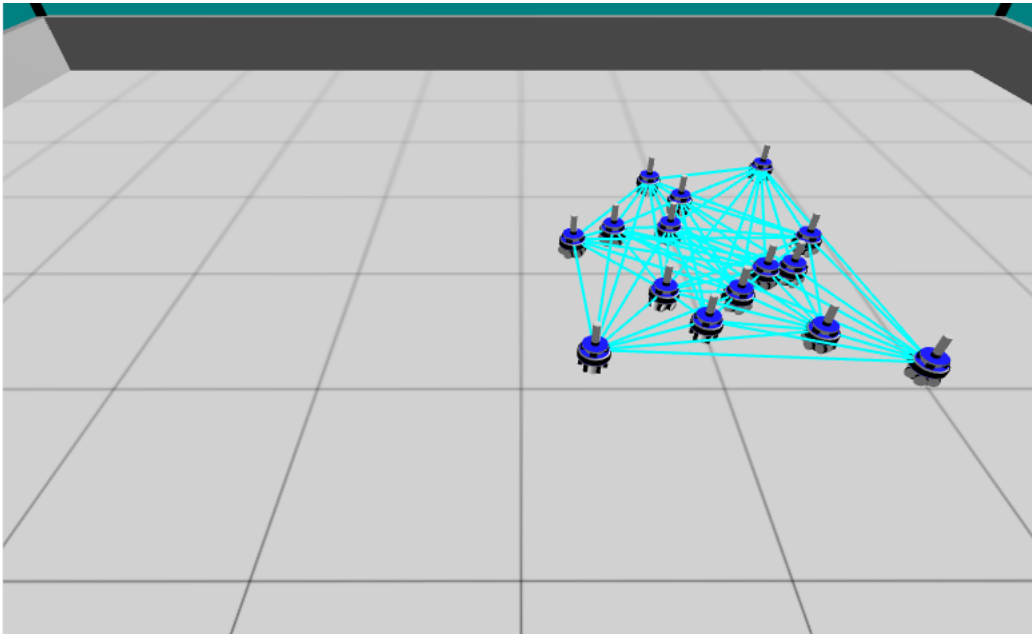


Figure 5.3: This figure represents the second task that is tested, aggregation. This is the ability for the robots in the swarm to cluster together no matter how spread out in the environment that they are.

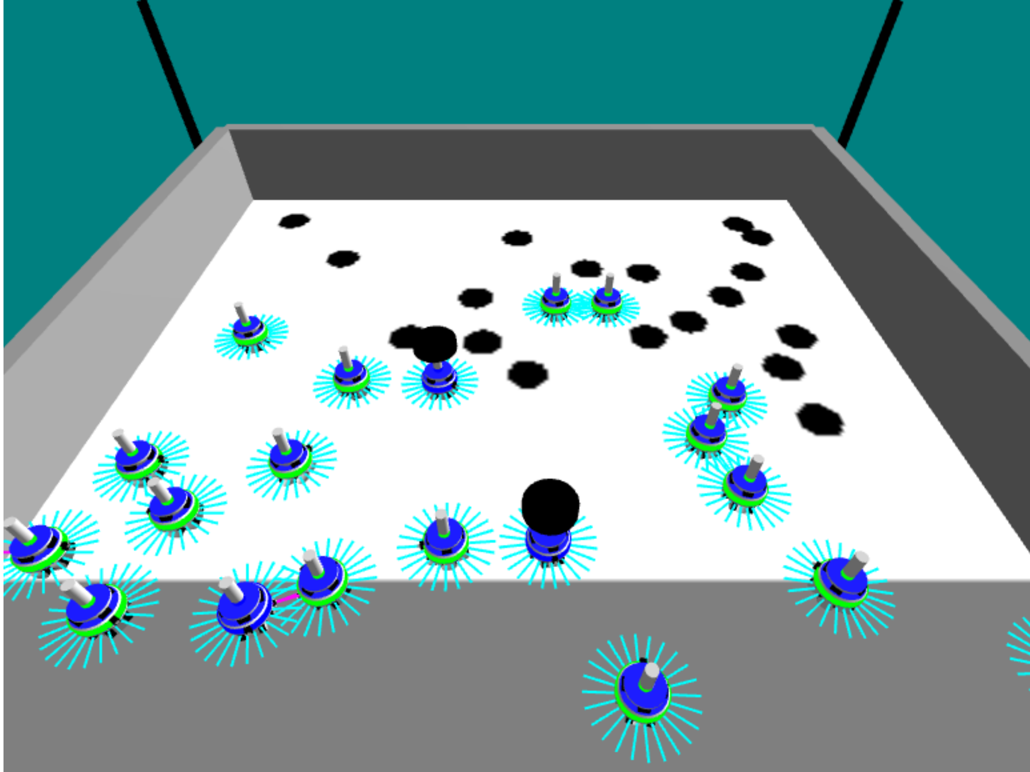


Figure 5.4: This figure represents the second of the task, foraging. This is the ability for a robot swarm to 'forage' or to search for items around an environment and bring them back to a 'base' or 'nest'.

Collective phototaxis is the ability of a robot swarm to sense a light source in the environment and collectively move towards it. The swarm is deemed successful if all robots reach the beacon. Foraging is the ability of the swarm robots to search for, and if found, transport 'food' items to a 'nest' (co-located with a light source). The swarm is deemed successful if all robots make it to the nest, and if all the items are collected. Finally, aggregation is the ability for the robots in a swarm to 'aggregate' or cluster together in the environment.

For the learning, the system was setup with a swarm of 10 robots and one fault is injected at one time during the learning. For each state, the simulation is run 50 times for the selected actions and the mean of the rewards are thereafter calculated and used in the Q-value calculations.

Algorithm 2 SOM + RL Method

```
1: procedure For Training/Learning
2:     for <All input vectors in training data> do
3:         Send an input vector state X(S_t), to the SOM
4:         Identify the winning unit in the input map
5:         Select possible action using π_t(S_t,A_t)
6:         Choose action, A_t to calculate reward
7:         Receive reward based on performance in environment
8:         Calculate Q-value and update Q-table using update rule in equation (2)
9:         Update the winner unit in SOM using the update rule in equation (4)
10:        Update neighbouring units in the SOM
```

5.1.1.1 States

For this initial proof-of-principle, the initialisation of states is performed by a global observer, as is the selection of actions from the Q-table at run-time.

The robots are initialised based on the input vector state $X(S_t)$ chosen. The input state $X(S_t)$ is defined as: $[d_1, d_2, d_3, b_1, b_2, b_3, p_1, p_2, p_3, I, d_{rs}]$. The input state vector passed to the SOM describes the features of the swarm that are important in choosing the best recovery strategy. The SOM is used to quantise the continuous state space to a discrete state space. Each unit in the input map represents a discrete state in the Q-table while the actions remain discrete in the Q-table.

Distance from the faulty robot d_1 ... d_3 describes how far the nearest robot(s) is from the faulty robot.

'How busy' nearby robots are b_1 ... b_3 describes how busy the nearest robot(s) is. The designer can decide how the swarm assigns the 'busy' rating. This is rated on a discrete scale from 0 to 5, where 0 means the robot is not busy and 5 signifies that the robot is very busy. If the robot is not busy, then it can tend to the faulty robot immediately, but as the robot 'busyness' increases, the longer it takes for the robot to be deployed. Although swarms are homogeneous in nature, there are some tasks where different robots have different capabilities and also different sub-tasks. This property is especially useful in these areas; for example in foraging, where other robots are searching for food, some robots have found food items and are carrying them back to the base.

Power left p_1 ... p_3 describes the amount of power left in the robot at the time that the fault is detected. This is in percentage, so as to make calculating the

Parameter	Value
Arena size	10x10 metres
Size of SOM	10x10 units
Number of states in training data	2000
Simulation time	1300 seconds
SOM Input map learning rate, β	0.6
Number of state-action repetitions	50
Q-learning learning rate, α	0.2
Exploration factor, ϵ	0.3
Neighbouring function	$\sigma = \sigma_o exp(\frac{-t}{\lambda})$

Table 5.1: The table describes the parameters used for the learning process. All Q-values are set to zero and the weights if the input SOM are initialised randomly in the range [0,1]. The set of parameters described above were empirically derived. The value σ represents the width of the neighbourhood which shrinks over time, σ_o represents the initial width of the neighbourhood, t represents the current iteration loop, λ represents the time constant.

reward easier.

Importance of the faulty robot I describes how important a faulty robot is. For example, if it is actively busy with a task, e.g. transporting an object in a foraging or search-and-rescue task, it will be considered more important. The designer can decide how the swarm assigns the importance rating. This is rated on a discrete scale from 0 to 5, where 0 means not important and 5 signifies that the robot is very important. If a faulty robot is not important, the recovery process does not aim for fast repair, but at the same time, we want to reduce the overall cost of the repair.

Distance to repair station d_{rb} describes how far the faulty robot is from a repair station (in meters).

5.1.1.2 Actions

There are two separate possible action sets are:

- Select any combination of the three robots chosen closest to the faulty robot,

which will be involved in the fault recovery. There are seven possible robot combinations: (A, B, C, AB, AC, BC, ABC).

- Select one of the four predefined recovery mechanisms below:

Transport to repair station: There is a repair station where faulty robots can be taken to be repaired. This behaviour involves the assisting robots gripping the faulty robot and dragging it to the repair station. The chosen robot(s) returns to the task, leaving behind the faulty robot to be fixed.

Repair on the spot: Following [25], we assume that each robot has the ability to repair other robots in the swarm, and that the robots have access to a repertoire of recovery mechanisms which can fix common faults. This behaviour could be especially useful if a faulty robot is very important and needs to resume its task immediately. However, this takes a significant amount of time and energy. Each fault takes a different amount of time to fix, it takes less time to fix the faulty robot if more assisting robots are recruited. However, there is a limit to how many robots make a difference for the 'cost' of the repair.

Drag Along: This behaviour requires only one robot to drag a faulty robot along. When the 'helper' robot gets to the faulty robot, it grips it and continues on with its task. It should be noted that it takes energy to drag a robot along; therefore it needs to be included when calculating the reward.

Leader-Follower: This behaviour also requires only one robot and does not work for specific faults: complete/partial motor failure and power failure. The faulty robot copies behaviour of helper robot.

If multiple robots are chosen as the first action, drag along and leader-follower behaviours are not allowed. The swarm must first select from action set 1 (choosing robots) and then from the action set 2 (repair mechanism).

5.1.1.3 Rewards

To quantify the expense of a particular action, we use a cost function 5.1, defined as a weighted sum of objective measures that can be evaluated in simulation. The reward of the reinforcement learning in this experiment is based on this cost. The cost is to be minimised is because we want the fault recovery to be done as efficiently as possible, that is, low reward is better. The cost is calculated based on the time it takes to get to the faulty robot, time it takes to finish the predefined recovery

Algorithm 3 Learning process

1: **procedure** FOR TRAINING/LEARNING
2: Initialise ARGoS simulation using state from training data.
3: Choose action from action set (1): move to faulty robot
4: Calculate cost for taking action
5: Choose action from action set (2): run recovery mechanism and calculate cost
6: Calculate total reward
7: Update the *Q-values* in the Q-table
8: Terminate the episode after a fixed duration

mechanism, the energy it takes to get to the faulty robot, the energy it takes to complete the predefined behaviour. There are also punishment/reward clauses:

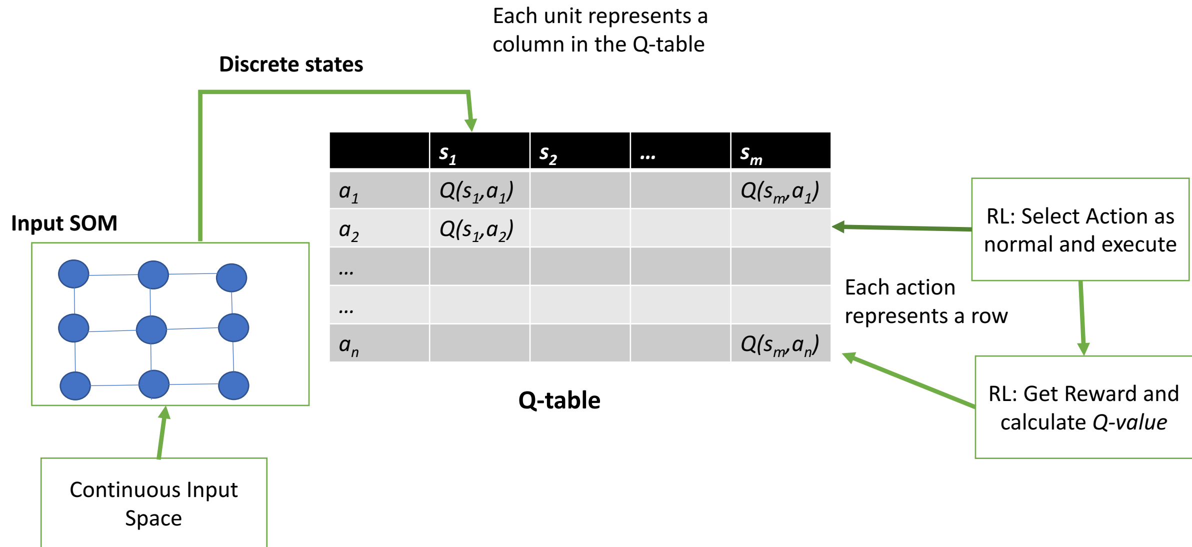
- If *all* robots complete the task, reward by reducing the cost (- 1000)
- If only some robots complete the task, punish by increasing the cost (+ 1000)
- Punish robots that take a long time to fix an 'important' robot (+ 1000)
- Punish recruited robot(s) that run out of power before completing the task (+ 1000)

The cost equation is defined as follows:

$$c(t, e) = aT_{fr} + bT_{pb} + cE_{fr} + dE_{pb} + eP \tag{5.1}$$

where a,b,c,d,e are scalar weights. T_{fr}: time to get to the faulty robot T_{pb}: time to finish predefined behaviour E_{fr}: energy it takes to get to the faulty robot E_{pb}: energy it takes to finish the predefined behaviour P: punishment/reward

Algorithm 3 describes the complete learning process. In order to demonstrate proof of principle, the learning is initially done off-line using a global perspective.



	s_1	s_2	...	s_m
a_1	$Q(s_1, a_1)$			$Q(s_m, a_1)$
a_2	$Q(s_1, a_2)$			
...				
...				
a_n				$Q(s_m, a_n)$

Figure 5.5: Representation of how the self organising map works with reinforcement learning in our approach

5.2 Experimental Setup

Work done for 10, 20, 30 robots in a swarm with increasing number of faults introduced. Also a mixture of different faults introduced later in the thesis.

For the learning, the system was setup with a swarm of 10 robots and one fault type is injected at one time during the learning. For each state, the simulation is run 100 times for the selected actions and the mean of the rewards are thereafter calculated and used in the Q-value calculations.

The system was tested on 50 randomly generated scenarios where each scenario is run with 100 different random seeds and the average performance is reported. We consider four different treatments that describes the steps taken by the swarm when getting results. This is done for three possible failure modes: motor failure, communication failure, light sensor failure and is described in the list below. It should also be noted that the robots are not allowed to leave any robot behind; they have to select from any of the predefined behaviours. To iterate, from the beginning

of the task, the swarm generates imagined scenarios, using a simulator, and learns the best recovery strategy from them before a fault actually occurs and is detected. When a fault is detected, the swarm selects the best recovery strategy based on what has been learnt during the imagined scenarios.

The following list describes the steps taken by the swarm from the beginning of the task to the end during the testing phases.

- Run experiments for collective phototaxis and foraging with no faults injected to give baseline performance
- Then test performance with two faults injected with faults; this is enough to break behaviour.
- Test the performance random selection of actions. This might help to recover swarm but could be suboptimal.
- Then test performance on the SOM and RL solution.

The algorithm to test the SOM + RL infrastructure is as follows: when a fault has been detected and diagnosed, the swarm state at this point is used as the input vector state, $X(S_t)$. The unit of the SOM with the smallest distance from the input vector is the winner unit. Next, the winner unit is identified in the Q-table and the action with the smallest Q-value is selected. This action is the best learnt recovery strategies for that particular state.

The termination criteria was 'out of time' and the 'no fault' fault scenario was used to determine the runtime. Additionally, at the start of the simulation, all robots are at full battery level that steadily degrades to NULL which occurs at the end of the runtime.

5.2.1 Results

The graph in Figure 5.6 describes the total average rewards over 2000 episodes. During each episode, the swarm generates the imagined scenario which is represented as the state that goes into the SOM and RL. The swarm thereafter selects the actions from both action sets and the cost from selecting these actions are calculated. As can be seen from the graph, at the beginning, the rewards fluctuate heavily as the swarm is exploring different actions because actions are not learnt yet. As the episodes go on, the swarm learns more and starts to select actions more intelligently, less exploratory and more exploiting though it still strikes a balance between exploration and exploitation. Towards the end, it can be seen that the rewards level out, as

the swarm starts to pick more optimal solutions based on previous rewards for the different states.

Statistical analysis for the experiments are also described below; the Mann-Whitney U Test and the Vargha-Delaney A Test are used for this analysis.

The Mann-Whitney U Test or the Wilcoxon rank sum test is a non-parametric test used when asked to compare the means of two groups that do not follow a normal distribution [85] . It calculates the probablilty p, that two samples are gotten from the same distribution. We use the null hypothesis which is a term used in statistics to describe when there is no significant difference between specified samples, that is, they have the same statistical properties: same medians, quartiles etc. The rank sum test calculation is done in R. If $p < 0.05$, we can reject the null hypothesis because the medians are different. However, it should be stated that if $p > 0.05$, it doesn't mean that we have to accept the null hypothesis. The Vargha-Delaney A test is a non-parametric effect magnitude test that can differentiate between two samples of observations [86] . This test is a measure of importance, scientific significance between two samples. A lies between 0 and 1. if A is 0.5, then there is no effect, that is, both samples have the same median. If A is 0.71, then there is a large effect and the effect steadily reduces till it gets to 0.5. On the the other side of the graph, from 0.5, the effect steadily increases as we descend to 0.29.

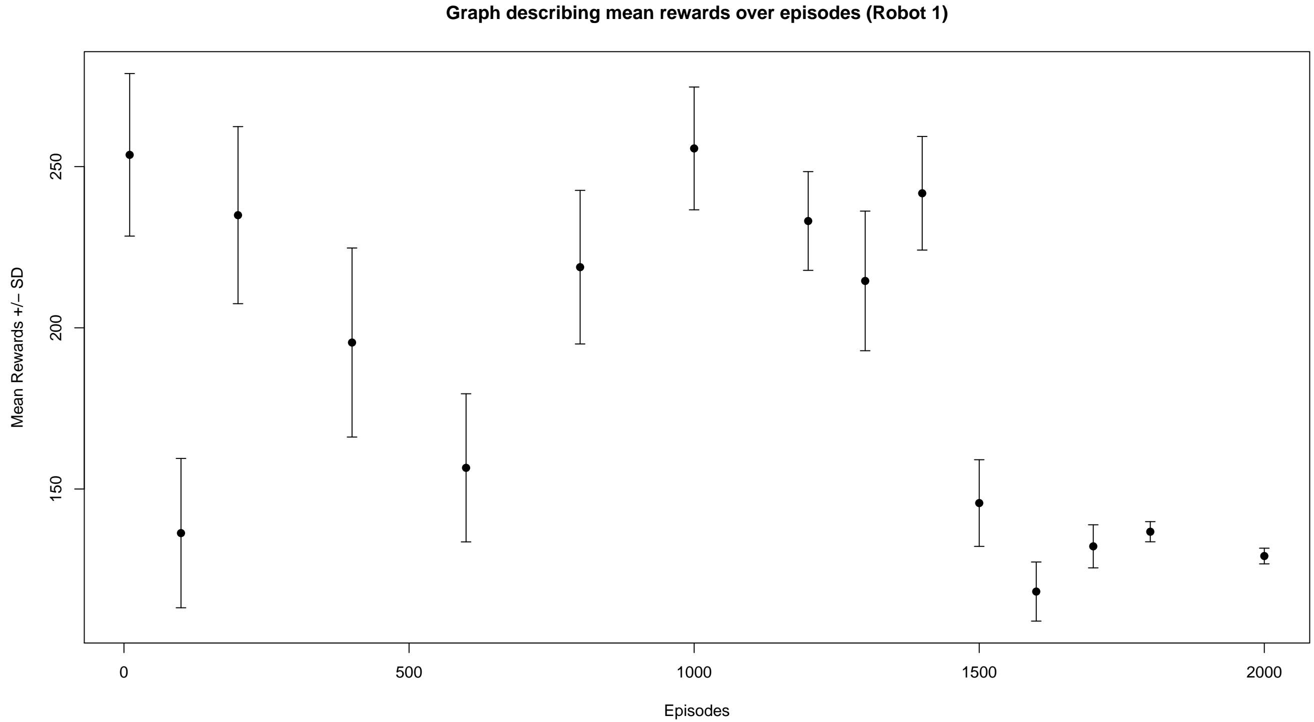


Figure 5.6: Rewards Convergence

This is tested for different sizes in the swarm (10,20 and 40 robots) to test the scalability of the learnt recovery strategy. It is also tested for three tasks aggregation, foraging and collective phototaxis. The results for the robot swarm sizes, 20 and 40 are in the appendixes.

10 Robots in the swarm

Table 5.7 shows that for the 50 test states, 23% of the states chose the strategy to drag to base, 29% chooses to fix on the spot, 34% chooses to drag the faulty robot along and 14% chooses to follow the leader. The reason why the Leader-Follower is chosen less is because it is limited for what faults it can be used on while the other predefined behaviours can be used on all the faults. The swarm picks the best recovery strategy based on the fault type and also the input state. The swarm never leaves a robot behind, they select the best strategy that has been learnt by the swarm. These results are collected across collective phototaxis, aggregation and foraging.

Recovery Type Chosen	Average Percentage Chosen
Drag to Base	23%
Fix on the Spot	29%
Drag Along	34%
Leader-Follower	14%

Table 5.2: The table describes the percentage of the test input that is selected during the testing phase of the RL + SOM and the recovery type selected. The recovery type column represents what predefined behaviour was chosen for the 50 sample inputs. The percentage column represents what percentage of the test inputs selects the corresponding recovery type.

Collective phototaxis

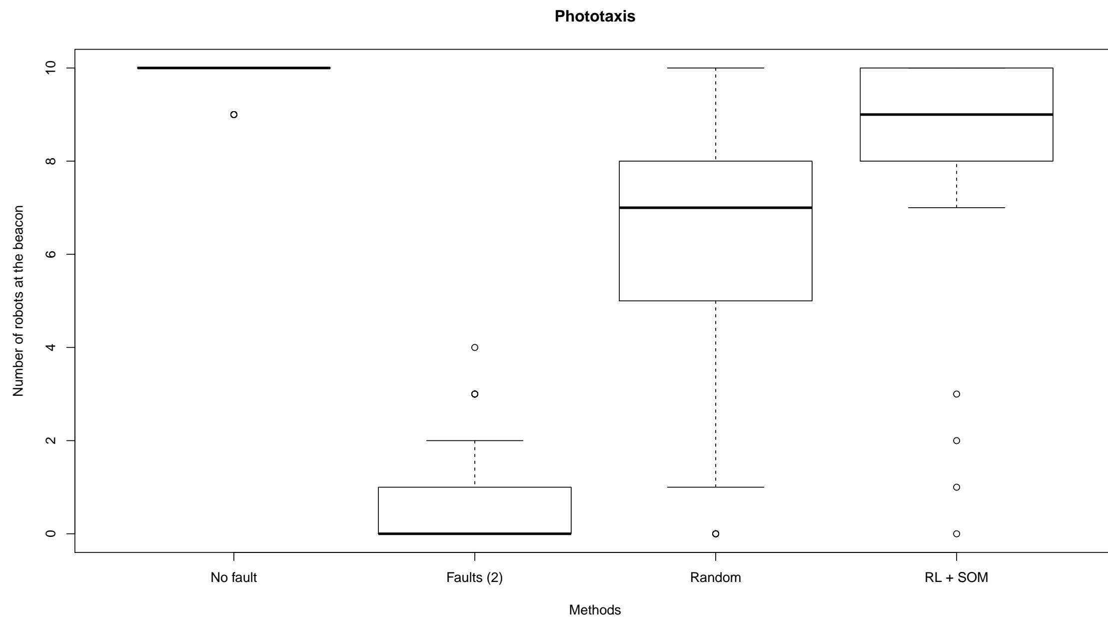


Figure 5.7: Results for collective phototaxis: Motor failures

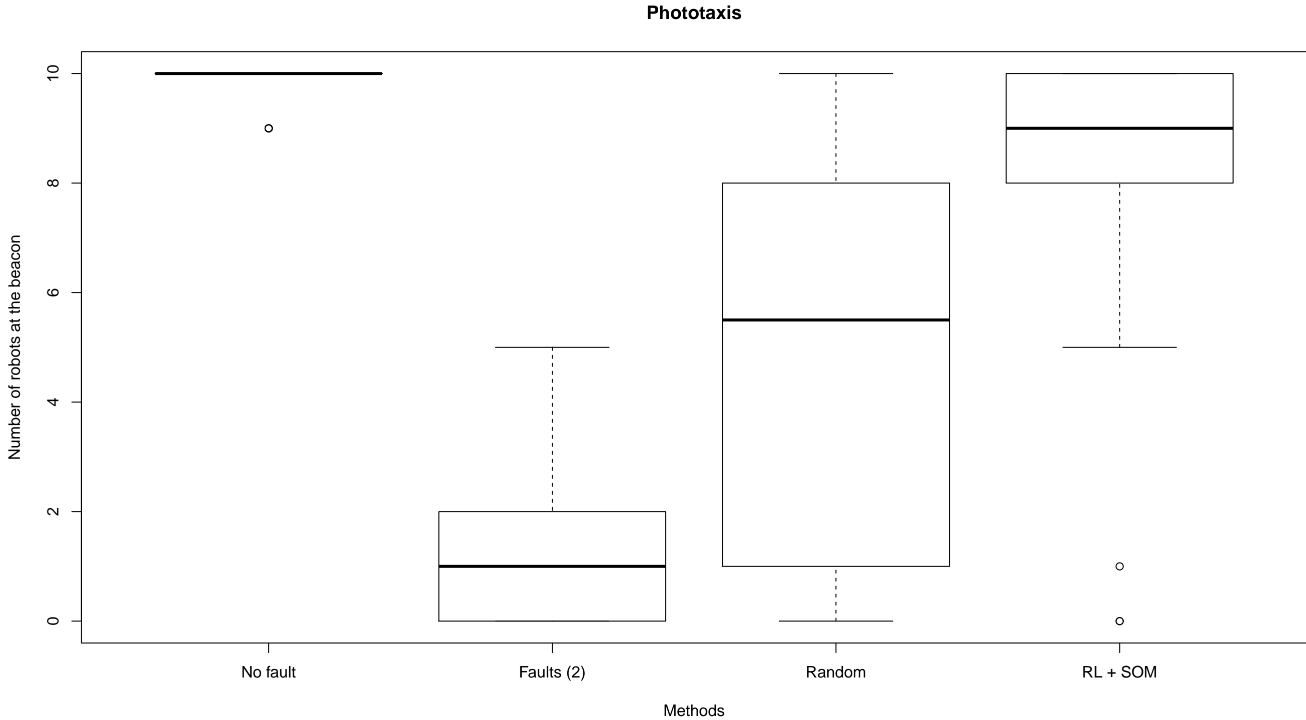


Figure 5.8: Results for collective phototaxis: Communication sensor failures

Table 5.3: Rank Sum and A Test for the Collective Phototaxis Experiments

Scenario	P	A
Collective Phototaxis (Motor Failure)	0.00086	0.784
Collective Phototaxis (Communication Sensor)	0.0018	0.761

The box plot in Figure 5.7 displays the number of robots that are either at the beacon or at the repair station (depending on what recovery mechanism is chosen). When there are no faults in the system, all of the robots in the swarm reach the light source most of the time. When two faults occur in the swarm, the faulty robot anchors the healthy robots as their sensors are still functioning like in [8]. In some cases, some robots are able to get to the light source; as they manage to escape the pull of the faulty robots. When a random behaviour is chosen, sometimes all robots make it to the light source though in some cases, the worst behaviour is chosen. When the fault recovery strategy chosen is based on what is learnt using the SOM

and RL, effective strategies are often chosen and the majority of the robots in the swarm complete the task.

For communication sensor failure (see Figure 5.8), the results show that when no faults are injected, the robots are able to complete the task i.e. are able to reach the light source. When faults are injected, the faulty robot cannot sense other robots accurately so it collides with other robots or obstacles. The swarm does not make it to the light source. The robots are anchored near the faulty robots preventing the swarm from reaching the light source. Random selection sometimes chooses a good strategy, but in a lot of cases sub-optimal or bad strategies are chosen; energy runs out for the 'healthy' robot causing fewer robots to make it to the beacon. The results show that our SOM + RL method allows the swarm to recover successfully and most of the robots are able to make it to the light source. Fewer robots are lost compared to random selection. The outliers are due to situations where the test scenarios was highly dissimilar to the scenarios to the training data used to train the system.

Aggregation

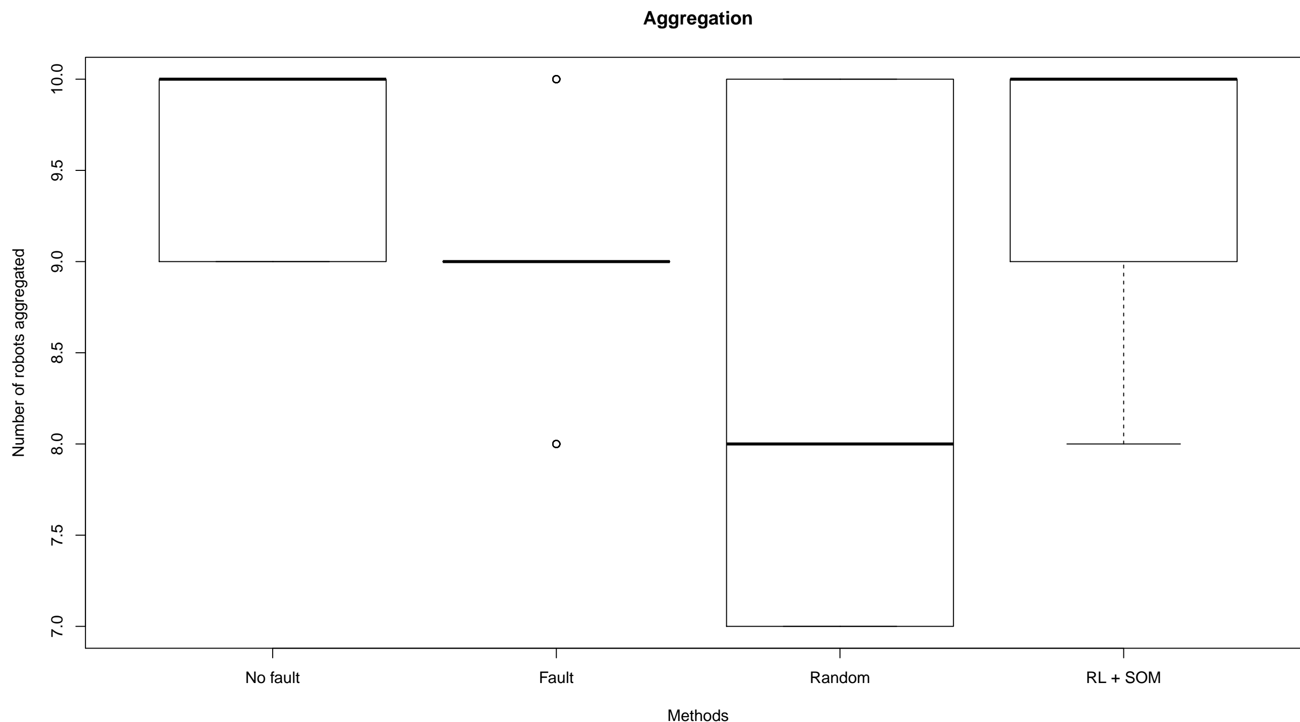


Figure 5.9: Results for aggregation when the range and bearing sensors (Communication sensors) on the robots are faulty.

Table 5.4: Rank Sum and A Test for the Aggregation Experiments

Scenario	P	A
Aggregation	0.0285	0.609

The box plot in Figure 5.9 describes the number of robots that are either in maximum number of aggregates or at the repair station which depends on what recovery mechanism is chosen. When there are no faults in the system, all robots are able to aggregate together to form just one aggregate. The discrepancy with the number of robots in one aggregate occurs on how tightly compact the robots are in the aggregate. When multiple robots have faults, the robots are not able to sense the nearby robots to create one aggregate; the robots break up and multiple

aggregates would be formed where robots would be divided between the faulty robots that move haphazardly around the environment. When it comes to recovering from this fault, random selection sometimes chooses a good strategy, but in a lot of cases sub-optimal or bad strategies are chosen. The results show that our SOM + RL method allows the swarm to recover successfully which allows the robots to form one cohesive aggregate.

Foraging

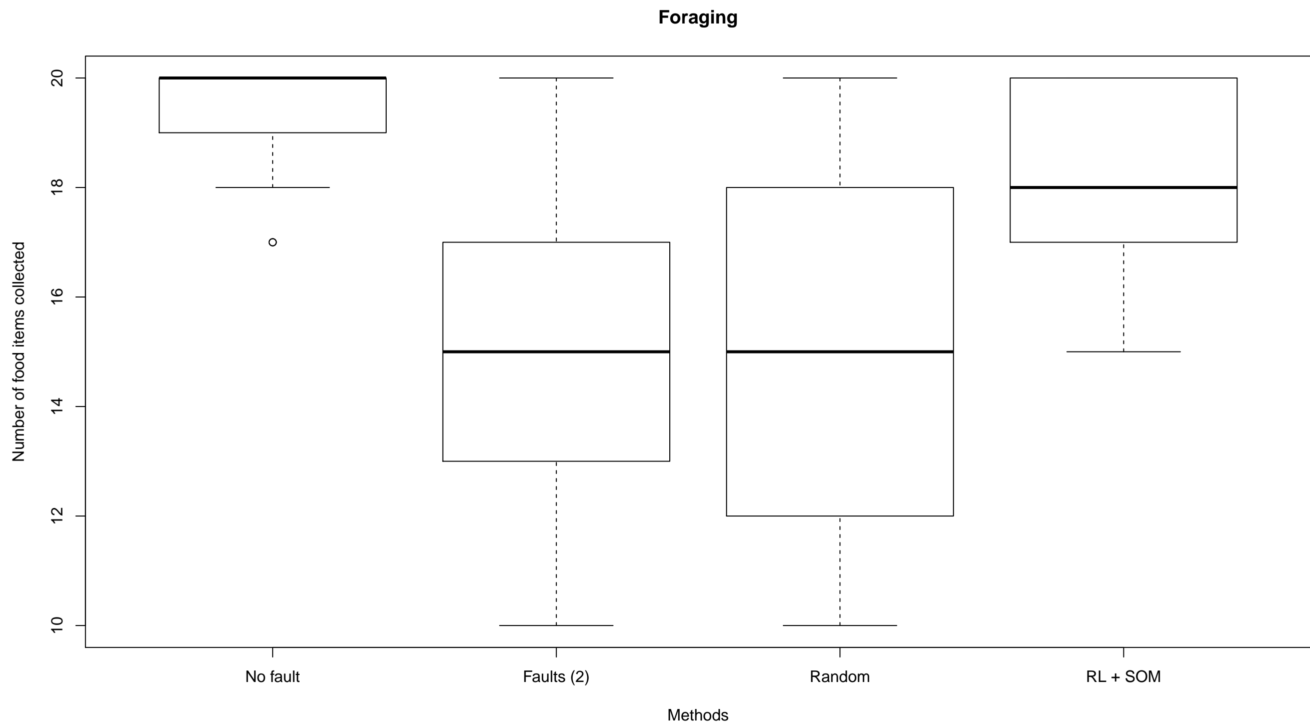


Figure 5.10: Results for foraging when light sensor failures are injected: Number of robots that reached the light source.

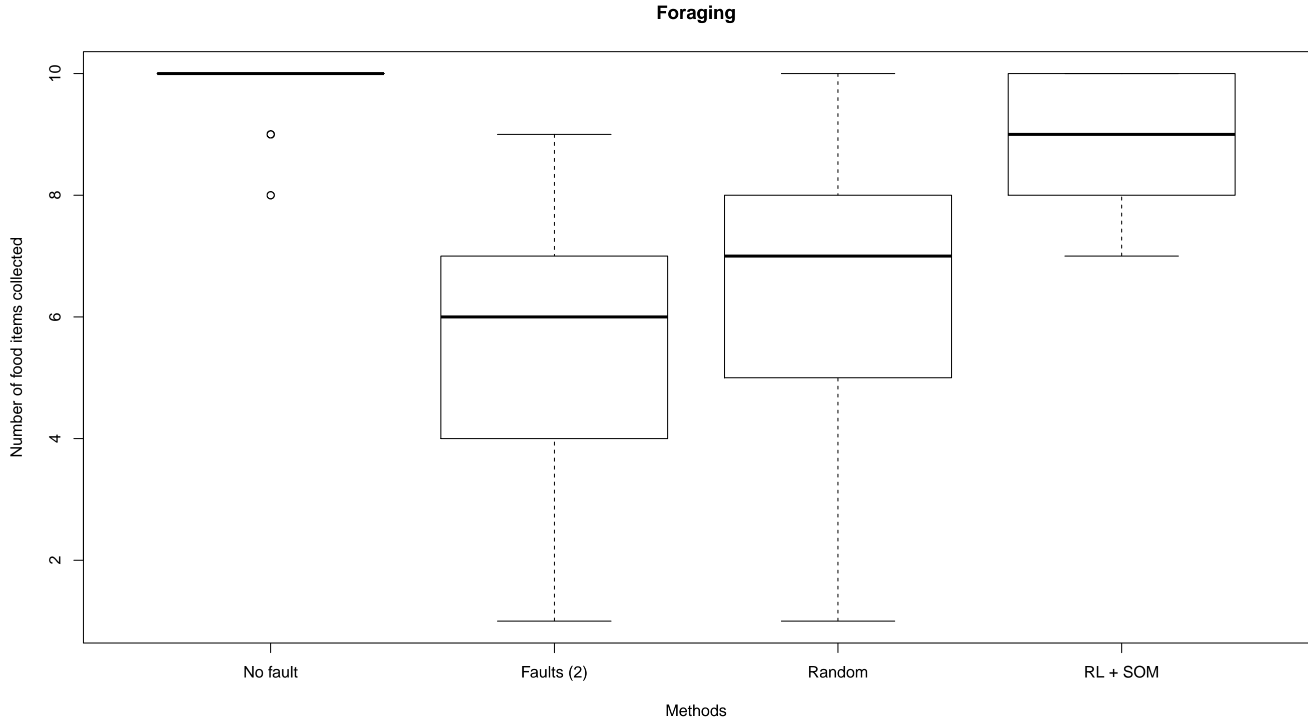


Figure 5.11: Results for foraging when light sensor failures are injected: Number of items collected.

Table 5.5: Rank Sum and A Test for the Foraging Experiments

Scenario	P	A
Foraging (Food at base)	0.0004	0.696
Foraging (Robots at light source)	0.021	0.733

Figure 5.11 shows the performance of foraging, when light sensor faults are injected. The robots use the lights to know where the nest is, so when there is no fault recovery, the faulty robots cannot sense the light and therefore cannot make it to the nest. The faulty robots wander aimlessly around the environment and reduces the probability that other robots will find and collect the rest of the items. The results show that sometimes the robots find their way back to the nest, and the remaining robots are able to collect the rest of the items, but that does not happen frequently. Random selection sometimes chooses a good strategy, but in many cases sub-optimal

or bad strategies are chosen. In the corresponding figure 5.10, the results also show that for our SOM + RL method, the swarm is able to collect most items and most robots make it back to the nest. There are more robots to collect items. More robots are able to return to the nest and collect items in the environment compared to random selection. The reason why this result differs from phototaxis is because the swarm does not suffer from anchoring effect here when faults injected; rather the swarm is unable to effectively search the environment for items to be transported back to the nest allowing items to be left behind.

5.3 Inclusion of Obstacles

The work done in part one is a simplified version of learning that can be done. In part one, the recovery process is done in an uncluttered and basic environment. The primary reason for this is to test the validity of the learning algorithms when applied to this specific scenario. Following the conclusion from the Part 1, the hypothesis is valid and, for completeness, the difficulty level of the learning is increased. The learning procedure is tested with the following different fault types:

- Complete Motor Failure
- Communication Failure
- Partial Sensor Failure
- Complete Sensor Failure
- Complete Power Failure

There are two additional aspects

State Space Increase As the local environment is cluttered, there are new states that the robots would need to consider when selecting the appropriate recovery strategy. The new states include: objects that are positioned randomly around the environment, a blockage obstructing the robots from getting to the repair station and also a fault diagnosis state that describes what type of fault is being learnt from. Remember, we are assuming that the swarm has the capacity to detect and diagnose faults.

Reinforcement Learning Rule The selection of action and state space has been straightforward so far, however, with the introduction of new states, there

is a need to increase the complexity of the reinforcement learning rule. In this new scenario, actions can now change before the episode finishes, only if there is something actively interrupting the 'helper' robots from completing the recovery process. The robots are monitored continously, and if they meander around an obstacle for longer than 10 seconds, change the action. This prevents the 'helper' robot from meandering till it runs out of energy. This would allow the 'helper' robot to change action to something more suitable for the present scenario.

From the action selected, you have your new state, (get reward) which then takes you to a new action and you get your new reward.

5.3.1 States

For this new learning scenario, the robots are initialised based on the input vector state $X(S_t)$ chosen. The input state $X(S_t)$ is defined as: [d_1, d_2, d_3, b_1, b_2, b_3, p_1, p_2, p_3, I, d_{rs}, f_d, n_o, p_{o1}, p_{o2}, p_{o3}, p_{o4}, p_{o5}, p_{o6}, p_{o7}, p_{o8}, p_{o9}, p_{o10}, $b_{y/n}$, p_b].

Distance from the faulty robot d_1 ... d_3 describes how far the nearest robot(s) is from the faulty robot.

'How busy' nearby robots are b_1 ... b_3 describes how busy the nearest robot(s) is. The designer can decide how the swarm assigns the 'busy' rating. This is rated on a discrete scale from 0 to 5, where 0 means not busy and 5 signifies that the robot is very busy. If the robot is not busy, then it can tend to the faulty robot immediately, but as the robot 'busyness' increases, the longer it takes for the robot to be deployed.

Power left p_1 ... p_3 describes the amount of power left in the robot at the time that the fault is detected. This is in percentage, so as to make calculating the reward easier.

Importance of the faulty robot I describes how important a faulty robot is. For example, if it is actively busy with a task, e.g. transporting an object in a foraging or search-and-rescue task, it will be considered more important. The designer can decide how the swarm assigns the importance rating. This is rated on a discrete scale from 0 to 5, where 0 means not important and 5 signifies that the robot is very important. If a faulty robot is not important, we do not necessarily care about how fast the repair is, but at the same time, we want to reduce the overall cost of the repair.

Distance to repair station d_{rs} describes how far the faulty robot is from a repair station (in meters).

Fault diagnosis f_d describes the type of fault that has occurred in the swarm. In this scenario, we use numbers to represent the type of fault. 0 - Complete Motor failure, 1 - Communication Failure, 2 - Partial Sensor Failure, 3 - Complete Sensor Failure, 4 - Complete Power Failure. Although we are using simple integers here to represent the fault types, ideally, features of the robot behaviour after the fault has been detected would be used here. As we are assuming that fault detection and diagnosis are available, if a fault that has been identified is not part of the 'commonly known faults', it should choose the closest features that best represents the fault from the fault diagnosis. This is randomly generated.

Number of obstacles n_o describes the number of obstacles to be initialised in the arena. The maximum possible m=number of obstacles in the arena are

Position of obstacles p_{o1} ... p_{o10} describes the positions of the obstacles that are randomly generated around the arena excluding repair station and area where blockage is initialised as shown in the diagram below. The obstacles are small cylinders that are initialised all over that arena randomly. The size of the cylinders are fixed

Blockage $b_{y/n}$ describes whether a blockage would initialised in the simulation. It's a 0 or 1 option. The size of the box is fixed.

Position of blockage p_b describes where in the section of the arena illustrated below, the blockage is initialised.

5.3.2 Updated Rewards

The calculation of the reward is similar to the previous rewards calculation where there are no obstacles. To iterate, the reward of the reinforcement learning in this experiment is based on this cost. The cost is to be minimised is because we want the fault recovery to be done as efficiently as possible. The cost is calculated linearly based on the time it takes to get to the faulty robot, time it takes to finish the predefined recovery mechanism, the energy it takes to get to the faulty robot, the energy it takes to complete the predefined behaviour. As it has been stated earlier, the actions can change mid-execution depending on if they are obstructed by any obstacles and are unable to finish the recovery strategy. However, we do not want the

actions to change mid-execution as this contributes to power used during the recovery process. In addition to the present punishment/reward clauses, more reward clauses are added due to the added complexity of this experimental setup.

- If *all* robots complete the task, reward by reducing the cost (- 1000)
- If only some robots complete the task, punish by increasing the cost (+ 1000)
- Punish robots that take a long time to fix an 'important' robot (+ 1000)
- Punish recruited robot(s) that run out of power before completing the task (+ 1000)
- Punish recruited robot(s) that change action
- Reward recruited robot(s) that finish recovery strategy (action set 1 and action set 2) without changing any acton mid-execution.

5.3.3 Experimental Setup

The learning setup is similar to the setup where there are no obstacles. The learning is done with 10 robots where the states are set and initialised in the swarm. For each state, the simulation is run 50 times for the selected actions and the mean of the rewards are thereafter calculated and used in the Q-value calculations.

The learning algorithm for this is similar to the algorithm where there are no obstacles however, there is an added component where the action can change mid-execution. Algorithm 4 describes the updated learning algorithm for this second experimental setup.

Table 5.6 shows the updated experimental setup where we can see that most of the present learning setup is the same except for the duration of the state-action repetitions and the training data number. These properties needed to be increased due to the new complexity of the states.

5.3.4 Results

Figure 5.12 describes the average rewards over 3000 episodes. It can be noticed that the number of episodes it takes to reach convergence in this experimental setup is more than the previous setup with empty environment. We can assume that due to the increase in the state space; it is logical to assume that it would take longer for the learning to converge. As can be seen from the graph, the rewards fluctuate at

Algorithm 4 Learning process

```
1: procedure For Training/Learning
2:     Initialise ARGoS simulation using state from training data.
3:     Choose action from action set (1): move towards faulty robot
4:     If obstructed by obstacle (status does not change in 10 secs), choose another
   action from action set (1)
5:     Calculate cost for taking action
6:     Choose action from action set (2):
7:     If obstructed by obstacle (status does not change in 10 secs), choose another
   action from action set (2)
8:     Run recovery mechanism and calculate cost
9:     Calculate total reward
10:    Update the Q-values in the Q-table
11:    Terminate the episode after a fixed duration
```

the beginning of the learning process which is what we expect because the algorithm is in the exploratory stage, trying different actions for the different states before it finally levels out. As the episodes increase, the algorithm starts to select actions more intelligently. Towards the end, we can see that the rewards level out, as the swarm starts to pick more optimal solutions based on previous rewards for the different states.

Parameter	Value
Arena size	10x10 metres
Size of SOM	10x10 units
Number of states in training data	3000
Simulation time	1300 seconds
SOM Input map learning rate, β	0.6
Number of state-action repetitions	100
Q-learning learning rate, α	0.2
Exploration factor, ϵ	0.3
Neighbouring function	$\sigma = \sigma_o exp(\frac{-t}{\lambda})$

Table 5.6: The table describes the parameters used for the learning process. All Q-values are set to zero and the weights if the input SOM are initialised randomly in the range [0,1]. The set of parameters described above were empirically derived. The value σ represents the width of the neighbourhood which shrinks over time, σ_o represents the initial width of the neighbourhood, t represents the current iteration loop, λ represents the space constant (distance measure).

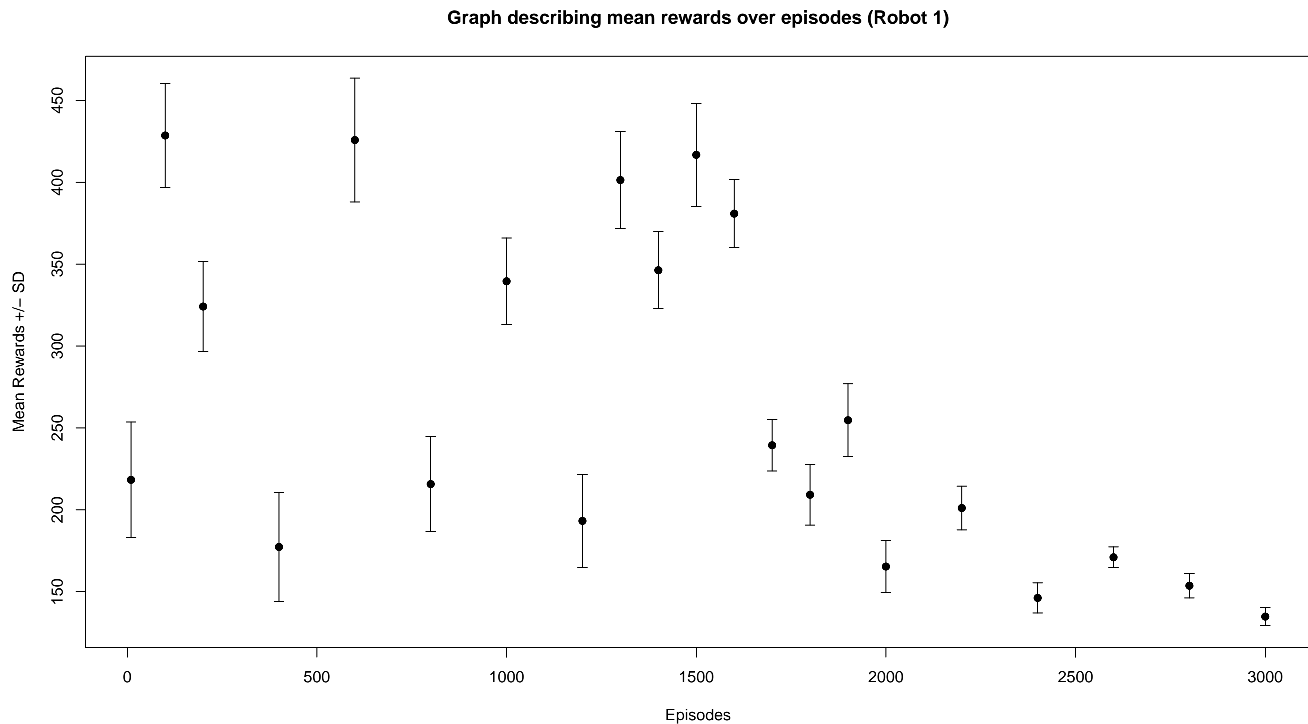


Figure 5.12: Rewards Convergence for New Experimental Setup.

Recovery Type Chosen	Average Percentage Chosen
Drag to Base	32%
Fix on the Spot	26%
Drag Along	24%
Leader-Follower	18%

Table 5.7: The table describes the percentage of the test input that is selected during the testing phase of the RL + SOM and the recovery type selected. The recovery type column represents what predefined behaviour was chosen for the 50 sample inputs. The percentage column represents what percentage of the test inputs selects the corresponding recovery type.

This is tested for the following tasks: collective phototaxis, aggregation and foraging.

5.3.4.1 10 Robots in the swarm

Table 5.7 shows that for the 50 test states, 32% of the states chose the strategy to drag to base, 26% chooses to fix on the spot, 24% chooses to drag the faulty robot along and 18% chooses to follow the leader. The reason why the Leader-Follower is chosen less is because it is limited for what faults it can be used on while the other predefined behaviours can be used on all the faults. The swarm picks the best recovery strategy based on the fault type and also the input state. The swarm never leaves a robot behind, they select the best strategy that has been learnt by the swarm. These results are collected across collective phototaxis, aggregation and foraging.

Collective Phototaxis

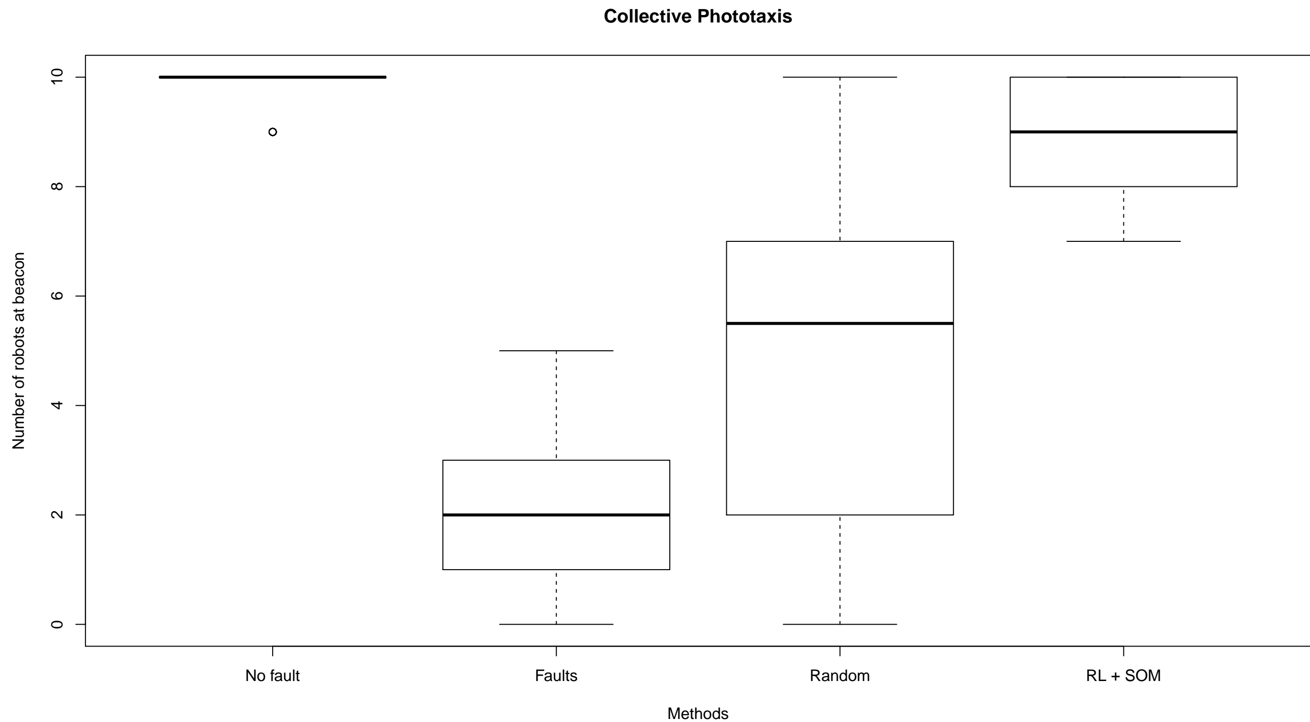


Figure 5.13: Results for collective phototaxis: Communication sensor failures

Figure 5.13 shows the results for collective phototaxis when the range and bearing sensor (communication sensor) has failed. The results are similar to what we saw earlier when the tested the learning algorithm in an uncluttered environment. The box-plot shows that when no faults are injected, the robots are able to complete the task i.e. are able to reach the light source. When faults are injected, the faulty robot cannot sense other robots accurately so it collides with other robots and obstacles in the environment. The robots are anchored near the faulty robots preventing the swarm from reaching the light source. Random selection works sometimes but it shows that SOM + RL perform better and it allows the robots to recover successfully from the faults and the robots are able to complete the task and make it to the light source.

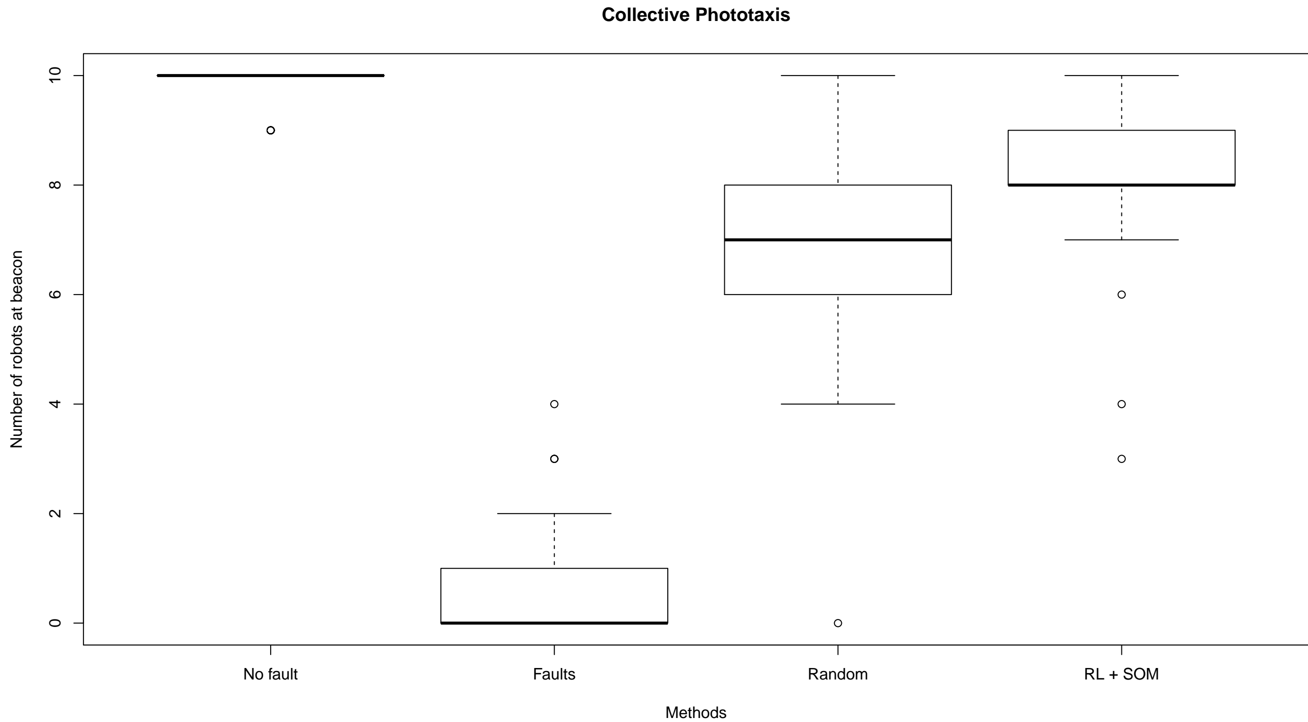


Figure 5.14: Results for collective phototaxis: Motor failure

Figure 5.14 shows the results for the collective phototaxis when the motor(s) have failed. The results, as we expected, are similar to what was observed earlier in the uncluttered environment using the learning algorithm. Again, the box-plot shows that when no faults are injected, the robots are able to complete the task, that is, they are able to reach the light source. When faults are injected, we can observe that the faulty robot(s) anchor the swarm as all the sensors on the robot is functioning. In certain scenarios, a few robots are able to make overcome the anchoring effect present and can make it to the light source, therefore completing the task. In the next box-plot, when a random behaviour is selected, it is able to select the optimal behaviour, where all the robots are able to make it to the swarm though in some cases, a less than optimal behaviour is chosen. However, when using learnt fault recovery strategy to select optimal solutions, effective strategies are usually chosen where majority of the robots in the swarm, if not all, are able to complete the task given.

Aggregation

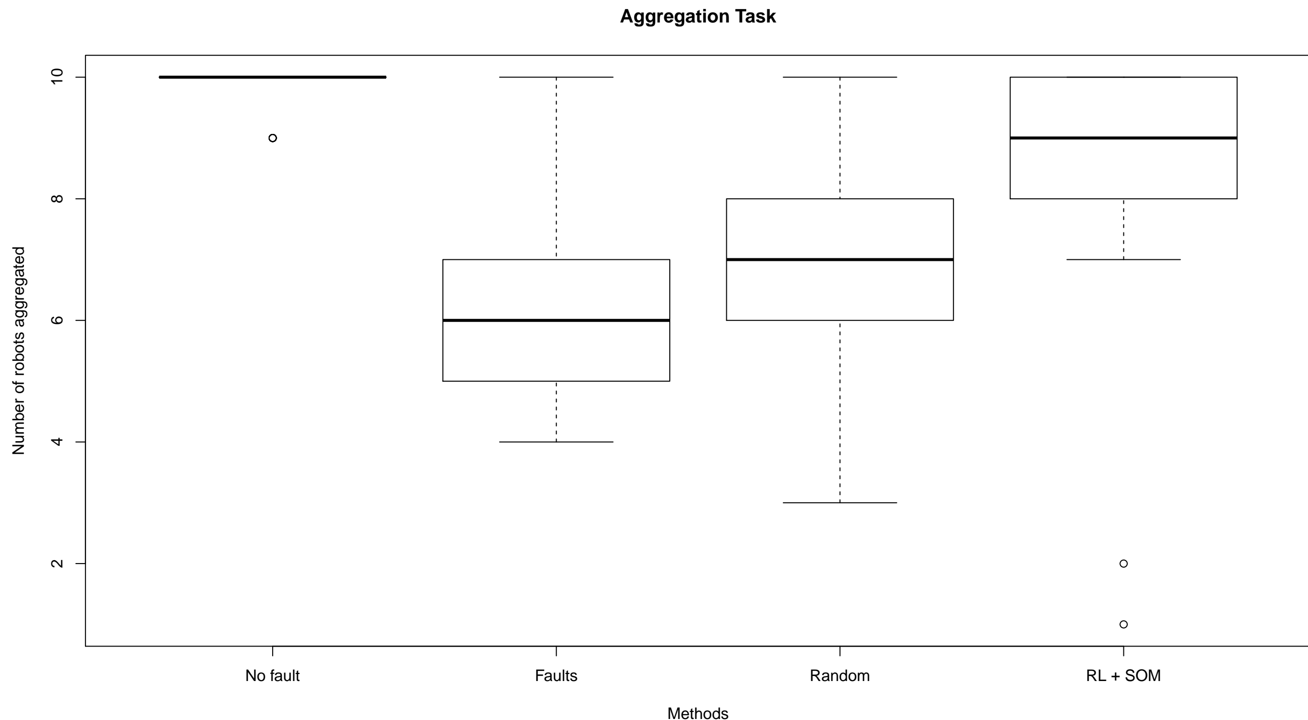


Figure 5.15: Results for aggregation when the range and bearing sensors (Communication sensors) on the robots are faulty.

Figure 5.15 describes the box-plot that displays the number of robots that are either in maximum number of aggregates or at the repair station which depends on what recovery mechanism is chosen. Again, as we are in a cluttered environment, how the aggregates are formed are different than in an uncluttered environment. Depending on the locations of the robots and obstacles at the time a fault has occurred, one or more aggregates may be formed especially if for some reason, the robots cannot make it pass certain obstacles. When there are no faults in the system, all robots are able to aggregate together to form just one aggregate. The discrepancy with the number of robots in one aggregate occurs on how tightly compact the robots are in the aggregate as the robots are consistently moving and keep being repelled and pulled in towards the centre of the swarm. When multiple robots have faults, the robots are not able to sense the nearby robots to create one aggregate; the robots break up and multiple aggregates would be formed where robots would be divided between the faulty robots that move haphazardly around the environment. When

it comes to recovering from this fault, random selection sometimes chooses a good strategy, but in a lot of cases sub-optimal or bad strategies are chosen. However, when using the learnt recovery strategy to select the optimal behaviour, we observe that most of the time, the robots are able to recover from the fault successfully and are able to form one cohesive aggregate.

Foraging

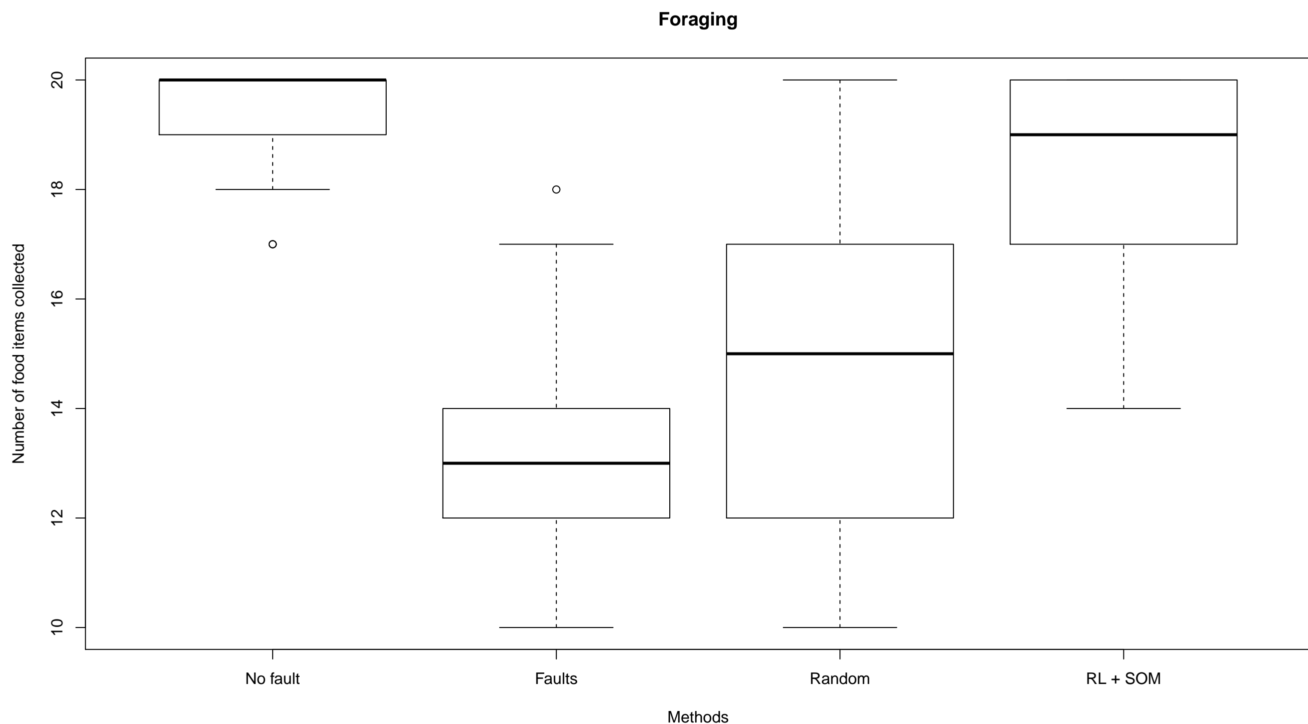


Figure 5.16: Results for foraging when light sensor failures are injected: Number of robots that reached the light source.

Figure 5.17 shows the performance of foraging, when light sensor faults are injected. This figure describes how many robots are able to return back to the home nest after 'foraging' for food. The robots use the lights to know find the nest, therefore when there are no faults present, the robots are able to return back to the nest after they are done searching for food. When faults are present in the swarm, the faulty robots cannot sense the light and therefore cannot make it back to the nest. The faulty robots wander aimlessly around the environment and reduces the probability that other robots will find and collect the rest of the items. The results

show that sometimes the robots find their way back to the nest, and the remaining robots are able to collect the rest of the items, but that does not happen frequently. Random selection sometimes chooses an okay strategy, but usually sub-optimal or bad strategies are chosen. With the learnt fault recovery process, we can see an improvement in how many robots make it back to the nest when compared to no fault recovery process or random selection.

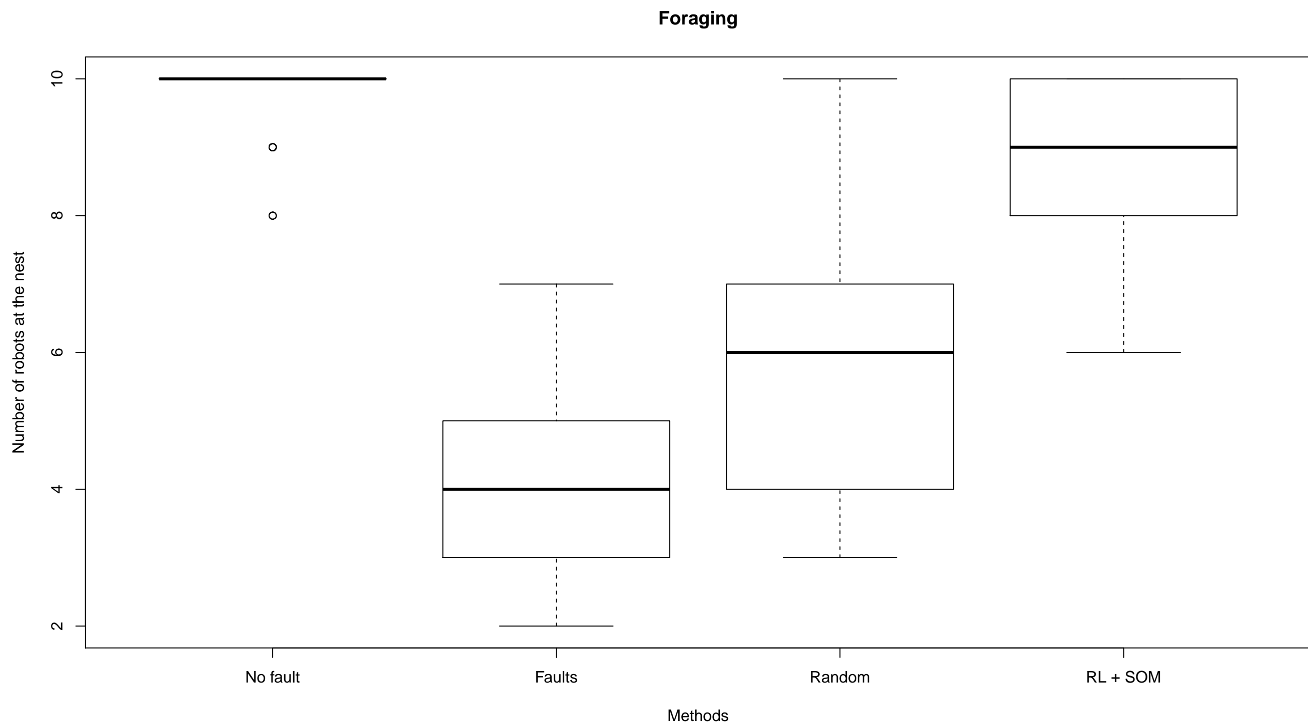


Figure 5.17: Results for foraging when light sensor failures are injected: Number of items collected.

In the figure 5.16, the box-plots describe the number of ‘food’ items collected fin the duration of the foraging run. The main point of foraging is to search and gather ‘food’ items and return them back to the nest. In this situation, when robots cannot use their light sensors to return back to the nest, they pick up the food but cannot return the items back to the nest thereby reducing the efficiency of how items would be collected and returned back to the nest. The results also show that by using the learnt fault recovery strategy, the swarm is able to collect most items and most robots make it back to the nest, returning the items back to the nest. The reason why more items are being picked up after the fault has been ‘recovered’ is because there are more robots to collect them. More robots are able to return to the nest,

drop the items and return back to the environment to collect more 'food' items in the environment compared to random selection. The reason why this result differs from phototaxis is because the swarm does not suffer from anchoring effect here when faults injected; rather the swarm is unable to effectively search the environment for items to be transported back to the nest allowing items and robots to be left behind.

We have shown how the proposed learning algorithm performs in the fault recovery process and how well it is able to perform when compared to randomly selecting a recovery behaviour. The method shows validity in learning what predefined behaviours would work best for certain states and it is able to generalise a suitable solution based on its present 'knowledge'. As stated previously, this chapter describes the learning algorithm being implemented in a more centralised manner where a central system is doing the learning and has a global view of the entire swarm and their environment. However, if a true fault recovery learning algorithm that is to be used in a swarm robotics manner is wanted, a distributed approach is preferred because swarm robots are distributed in nature. The next chapter describes distributed learning approach that better suits a swarm robotics context.

Chapter 6

Distributed Learning Approach

We have discussed Q-Learning in a centralised fashion where a central computer has a global view of the swarm of robots and the environment. The central system is able to use this information to implement the learning process and also to select the best predefined behaviour based on the information available to the central computer that is performing the learning. Additionally, during the testing after the learning process is done, the central computer is able to select the appropriate predefined behaviour when it is tested at the point a fault occurs in a real life situation. As discussed in the previous chapter, as the primary application for the learning approach discussed in this thesis finds its application in swarm robotics. Therefore, in keeping the distributed manner that swarm robotics is developed on, we are extending the centralised architecture for the learning process into a distributed learning architecture which would be better suit a swarm robotics system model. In this chapter, distributed learning is introduced, while the advantages and limitations and how they have been applied in a robotics context is also discussed. The implementation of distributed learning in the learning approach used in this thesis and the results of this implementation is also described and developed.

6.1 Distributed Learning

Collaborative distributed learning is an aspect of distributed learning that involves learning through interaction with oneself and other agents. The learning agents monitor each other's learning process and 'thinking' while obtaining and providing feedback for further clarification and enhancing each other's comprehension. This process could also affect the agent's learning however it further motivates the learning process with the agent [100]. Therefore, we can say that collaborative distributed

learning involves interaction and communication with other learning agents and possibly adapting or changing its own learning to better suit the collective.

Outside the context of robotics, distributed learning is an interdisciplinary domain that usually involves statistics, optimisation, learning theory and core aspects of machine learning algorithms. Distributed learning is widely adopted and used in machine learning applications because of how it deals with using massive data. There are four research problems , which are described below, that distributed machine learning can be broken into but it should be noted that these problems are not mutually exclusive [101]. It should be noted that we are compare the distributed approach to the centralised approach to determine if the fault recovery approach can be be developed and also successful in the implementation of the recovery process.

How to use Statistics, Optimisation Theory and Algorithms

The aim of most machine learning algorithms is to minimise the loss of a set of training data and due to this, some issues arise due to this aim:

- How long does it take the learning process to converge towards a solution?
- How optimal is this solution?
- How large should the data sets be to produce an useful solution?

To study these issues, researchers typically use theoretical analysis tools such as optimisation theory and statistics. Nevertheless, when dealing with machine learning algorithms that use large data sets and we have access to more computational power and resources and additionally, the aim is to reduce the learning or training time by using additional resources through distributed or parallel computing techniques, a set of different problems arise:

- By using distributed or parallel learning methods, are the training models able to converge faster?
- If this does not occur, how far are we from both the optimal solution from the distributed method and the original optimal solution (centralised learning method)
- What would be the new conditions or assumptions that are needed to realise new solutions to achieve convergence?
- Can we compare the speed of the distributed versus non-distributed learning (that is, the scalability) and can this be evaluated?

- Can the learning process be designed appropriately to ensure good scalability and convergence?

Developing Machine Learning models or learning algorithms suitable for distributed learning

This deals with scaling existing non-distributed machine learning algorithms or creating new machine learning algorithm to handle large data-sets

Building large-scale distributed machine learning applications

There are some specific application domains, for example, image classification, which requires research for scaling these specific machine learning algorithms. These solutions are deployed into the production line.

Developing parallel or distributed computer systems that scale up machine learning

If the learning algorithm cannot handle the computational work on one agent, distributed systems can be used to extend to other agents (increasing the computational resources that can be utilised). To do this however, other problems might be faced:

- Consistency: If the agents involved in the learning process each own different parts of the training data, how can it be ensured that all agents are working towards solving the same problem?
- Fault Tolerance: In distributed learning, the learning is spread across multiple agents. What happens if one agent develops a fault or error? Can this be fixed without restarting the process from the beginning?
- Communication: Machine learning involves moving parts such as I/O and data processing procedure? How do we ensure faster I/O and non-blocking data processing procedures that can be used for different environments.
- Resource Management: Computer clusters are expensive to develop and construct, therefore many users generally tend to share them. There is a need to manage and allocate the cluster resources properly whilst maximising cluster usage for each user.
- Programming Model: Finally, when considering the training model/algorithm, are distributed learning structured similarly as non-distributed learning algorithms? Is a more efficient algorithm one that uses less code? Can the learning algorithm be written considering just a single agent and thereafter amplifying it with distributed computing techniques?

These are issues and questions that we need to ask and consider when thinking about utilising distributed learning techniques. These questions show that there are limitations, presently, when it comes to applying these distributed machine learning techniques though the answer to these questions and the overall limits of distributed machine learning would differ depending on how the algorithm would be applied.

6.1.1 Distributed Deep Learning

Deep Learning is becoming a prominent part of machine learning because of how effective it is in various applications so it is plausible to assume that Distributed Deep Learning is also becoming prominent in the field of machine learning. There are two terms to be discussed when delving into distributed deep learning: Data Parallelism and Model Parallelism [101].

6.1.1.1 Data Parallelism

This is a technique that is utilised by partitioning the training data. In training data, using a parallel distributed techniques, the data is divided into parts where the number of learning agents (the computational nodes) equal the number of training data parts. Each agent works and computes on their own data part. As the entire training data set is being worked on in parallel using the learning agents, we are able to compute more data compared to a non-distributed learning model (using one agent in the learning process). One of the advantages of implementing distributed machine learning is how fast the learning goes and converges towards a solution due to using a number of learning agents. The learning model using data parallelism is straightforward and generally follows these steps [102], [101]:

- Each agent performs its own training. In traditional machine learning methods, stochastic gradient descent is an important method for optimising a fitness function and in this step, the agent generates its set of gradients.
- All agents synchronise their gradients via network communication to reach a consensus

The idea is that the synchronisation should not take much time and there is an improvement over the single agent learning process.

The distributed learning model employed in this thesis follows similar steps to data parallelism. Although we are not implementing deep learning in our algorithm, extending the knowledge and modifying the steps of data parallelism to better fit our application seems plausible.

6.1.1.2 Model Parallelism

This technique is a more complex learning technique compared to data parallelism. In data parallelism, we divide the training set into parts while in model parallelism, we divide the machine learning model/algorithm into parts to distribute the workload between the learning agents thereby sharing the computational load. The main question is how does one partition machine learning models? There are many machine learning models/algorithms and they are all written and represented differently. Therefore, there cannot be a defined guideline on how to implement model parallelism [101].

6.1.2 General Distributed Machine Learning Limitations

We have seen the advantages of implementing distributed machine learning and how it generally compares to non-distributed machine learning. However there are some limitations to utilising distributed learning. From the text above, we can summarise that data parallelism works best when we have a large training data set that we can distribute between the learning agents involved in the learning process. Model parallelism works best when we have a heavy computational learning model or algorithm that might be too expensive computationally (uses a lot of computational resources); there is a need to break down the learning model and distribute it across the learning agents involved in the learning process to allocate the computational resources required to allow the learning algorithm to function properly [101]. Ideally, we want a scalable system when dealing with distributed machine learning; that is, if we have N number of learning agents, we can train data N times more than a single agent per unit time. Therefore, our learning model or algorithm has N scalability (linear scalability) which is ideal [101]. Synchronisation, an important part in distributed machine learning, has a large overhead and it could take longer to finish one training run on a distributed learning cluster than on one single agent. More time is used to synchronise across multiple agents to make sure that the learning model converges at the end of the learning process. In some practices, synchronisation could take much longer than the actual learning process. One of the main reasons why this is a concern is that in some clusters, some nodes could be run on older hardware which would make the faster agents wait for these older agents to finish their computation before synchronising. In such practices, we would have a negative scalability which is not ideal as it is a waste of money, resources and time [101], [103].

6.1.3 Multi-Agent Distributed Learning in Robotics

We have discussed the general distributed learning approach. However as we want to implement the distributed approach in a robotics context, we need to discuss how to approach this. Robotics domain faces challenges that also plague inherent fault tolerant, adaptive and efficient distributed multi-agent systems that can generally cope with noisy and incomplete information, delayed feedback, random environments that are in response to actions, opponents and competition for resources. These challenges can be described below [104], [105], [106], [107]:

- Environments can be inconsistent and dynamic
- As this is multi-agent learning, it is not a singular robot that is involved but rather multiple robots which could cause some disruptions and interference
- Computational and other resources such as battery, time needed for learning may be limited
- Communication between the robots may not be accurate and could be low-bandwidth
- The rewards or feedback following the actions may be delayed or lost during the task
- Robot sensors can be noisy; they could provide inaccurate and incomplete information
- Robot grippers (that are used during the foraging task) could slip and cause errors

These are general problems that single agents could be experiencing but multi-agent robot systems share the same issues and could also compound the challenges that are described in the above list. However, it has been argued that, instead of thinking about the multi-agent robot learning issues as an extension and a compound result of the single robot learning issues, we think of multi-agent learning as a separate entity with its own specific challenges rather than as an extension of the single robot case [104].

Communication has been mentioned earlier as a possible challenge and solving this issue can help solve these two main problems [103]:

Hidden State/View This problem is due to the fact that the learning agents have limited view of the environment and might not have all the information to complete the learning task and for the learning process to be done efficiently.

Reward Assignment This problem is due to the fact that the reward in a distributed system is given at a global level (refer to the previous part where centralised learning occurs) and in a distributed learning approach, the reward has to be 'distributed' amongst all the learning agents.

We use a good communication system, for sensing and reward assignment, to make it less distributed for a brief period of time which would help reduce the impact of the hidden state and reward assignment problem that is inherently a distributed learning attribute. Communication is important in multi-agents including swarm robots. Behaviours and their consequences can be interpreted as some form of communication and the agents' cooperation. We can define communication in two forms: direct and indirect communication [103].

Direct Communication This form of communication involves sending direct messages to other agents via radio or bluetooth (there is a transmitter and receiver; the message is directed towards a receiver). The message to be transmitted could be one-to-one, one-to-many transmission.

Indirect Communication This form of communication involves observed behaviour of other agents and their consequences in the environment.

These forms of communication can be seen in nature; various animal species such as bees and ants can be seen displaying these kinds of communication when performing different tasks. We have discussed these forms of this communication in the literature review when discussing swarm intelligence and this application of this intelligence in the field of swarm robotics.

Cooperation is also another form of communication that is based on interacting with the other agents and the environment to perform tasks. There are two forms of cooperation to be discussed [103]:

Explicit Cooperation This form of cooperation is defined as a set of interactions that involve exchanging information or performing actions that would benefit other agents in the environment.

Implicit Cooperation This form of cooperation involves interactions of the own agents' goal however, in the process of achieving its own goal, the agent affects and aids the other agents in the environment to achieve their own goal.

When using communication systems for sensing, there have been comments on how unreliable and inaccurate sensors could be especially in robotics where sensors are very important when the agents are attempting to complete tasks. However, in

multi-agent learning, sensing from communication systems is very important, especially when it comes to sensing other agents, obstacles and objects in the environment. When using communication systems in solving the hidden state issue, ideally, we would like an 'image' view of the environment from each agent however, taking images and analysing these images has a huge computational overhead that is not suitable for a small, simple robot. Therefore, depending on the sensors that are utilised, one can get a good idea on the structure of the environment. By passing information and necessary messages between robots that communicates positions and other necessary information that is pertinent to your task, the robots can have a broader view of the environment and not just be limited to the local view. When using communication in reward assignments, the same idea used in the hidden state problem where you send the necessary information within a limited area to solve how to distribute reward amongst the robots doing the learning. The robots communicate locations and rewards with each other which would aid sharing reward and also sensing a larger area of the environment [103].

In summary, with multi-agent learning, there are challenges when implementing a distributed learning approach such as: limited communication, limited world view, rewards, sensor errors etc. However, there are steps that we can be taken lessen the impact of these challenges when implementing distributed learning; distributed learning is still not a perfect learning approach and there is still work that needs to be done, however, we can mitigate the effects of the challenges when implemented distributed learning. It has been discussed that when communication channels and systems are properly utilised, they can be used to solve the limited world view and delayed rewards problem. Sensors are hardware devices and would be always prone to error however, with the technology available now, the capability to process image without it being computationally expensive (using the software on a simple, small robot) is not available but it should be possible at some point in the future.

6.1.4 Distributed Reinforcement Learning

The idea of distributed learning is not to have the same features or possible learnt behaviours and effectiveness as the centralised method but rather to create a more effective learning process utilising all agents in the learning process. In the previous section, we have discussed distributed learning in a general robotics context. However, some work has done when referring to using reinforcement learning in a distributed setting [105], [106], [107], [108]. Traditionally, Reinforcement Learning (Q-learning, as applied in this thesis) is used in a centralised manner where one agent (where in this thesis, a central computer) does the learning process. One of

the main reasons why there is a lack of multi-agent q-learning processes is because as the number of agents performing the learning process increases, the state-action space increases exponentially therefore increases the learning time exponentially as well. This is usually not a desired effect especially in our application, as we would like to maximise the learning potential and have a more effective learning process [105], [106], [107].

As discussed earlier, distributed reinforcement learning has been done in other literature however, the approach used to implement distribution for RL in this thesis is a novel approach.

A distributed architecture has intrinsic properties that reinforcement learning could benefit from such as parallelism, robustness and scalability. As we are dealing with swarm robotics context, there are issues that need to be addressed when discussing distributed learning:

- The agents involved in the learning process (the robots in the swarm) only have a partial view of the environment; they have a view of just their immediate environment as they are limited by their sensing capabilities
- Each agent can only move in specific ways; forward, left and right especially in swarm robotics context as each robot is designed to be as simple as possible. This also limits the complexity of the actions and additionally, the learning
- Each agent might arrive at a different action (solution) from the other agents in the environment. This could cause disruptions within the robots and prevent the appropriate action to be taken

One of the major problems with distributed reinforcement learning is the question of how they can coordinate their actions (finding the right sequence of groups of well-matched actions) so that they can collectively perform the task that they are trying to accomplish. In this situation, the task is the fault recovery approach which is selecting the appropriate predefined behaviour for different environment states. Another issue that been has stated above is the increased state-action which would increase learning time. In the next section, we discuss how we solve these issues to implement an optimal learning process for our fault recovery approach.

6.2 Learning Approach

We have discussed how learning in a distributed setting is approached in the previous chapter. Communication and cooperation is an important aspect of distributed

learning in helping to solve these issues. As in the previous chapter (centralised approach), we discuss how we are trying to accomplish a pre-fault learning strategy where most of the work is done in simulation. Therefore, we have to determine the level of importance of communication and cooperation. There are two questions that we need to ask when moving forward with implementing distributed architecture to our present centralised learning architecture:

- How much do the robots share with each other to allow for an efficient learning process?
- How do all the robots arrive at an optimal solution to the problem?

We have mentioned previously how we are using 'model' scenarios during the learning process; the robots are learning imaginary but plausible states that could be experienced when the robots are actually online and running tasks. Therefore, during the learning, the state features are modelled and no communication between the robots would be needed. Each robot can be seen as its own global observer, observing the internal learning simulation being done in its internal simulator. It has information about the nearby, non-faulty robot(s) that would be involved in the fault recovery. However, after the learning is done and the robots are being used in the real world (albeit still in simulation), this is where the bullet points become important. The first concern is what information, and how much of this information, is permitted amongst the robots to implement an efficient learning process. The robots have to communicate their states to each other to allow for the robots to select the appropriate predefined strategy. The reason for this is that the optimal predefined behaviour is dependent on the state and some of the state features is not locally available to each robot, therefore, if the robots want to have all the information needed to select the optimal predefined strategy, there needs to be direct communication between the robots.

The second point deals with the issue of consensus, which is also an issue during the learning process. This becomes apparent when learning is preferred and the robots are being used in the real word (still in simulation), running set tasks and a fault occurs. Each robot has had a different learning experience, therefore may not arrive at the same recovery solution for the same fault that is being assessed by the other robots that would be involved in the fault recovery. We do not expect to implement different predefined recovery strategies in one instance, but rather it is necessary that all robots are able to arrive at a common solution when it is time to select a recovery solution. One way to solve this would be to do some sort of distributed consensus. This idea is not completely new as we can see in general swarm

intelligence literature (as discussed in the literature review), swarms of ants and bees are able to arrive at similar solutions without having to directly communicate with each other. By relying on their interactions with each other and their environment, they are able to synchronise effectively to arrive at similar solutions when solving tasks and problems. Ideally, this is the approach that we would like to implement but it is non-trivial. For this thesis, we implement a standard consensus that functions similarly to majority voting. It is understood that this might not be technically classified as a completely distributed technique, however, direct communication is required for sharing information between the robots, as explained in the previous paragraph and therefore this process can be also utilised robots to allow the robots to come to an agreed solution. The robots directly communicating with each other does not necessarily take away the distributed aspect of this approach [103].

As with the previous work done in the centralised learning, this thesis still uses reinforcement learning (RL) and self organising maps (SOMs) when implementing our fault recovery approach.

6.3 Empty Environment

In this scenario, we discuss an approach to fault recovery that involves selecting predefined recovery strategies intelligently when this is learnt in an empty environment. Our previous architecture for the fault recovery process uses a global observer to calculate the system state and select the most appropriate recovery action. In contrast, this section extends this architecture to a decentralised system, where each robot in the swarm makes decisions based only on local information that they receive from the environments and other robots in its immediate vicinity. I assume that the swarm robotic system is capable of detecting and diagnosing faults already, so that focus here will mainly be on recovering from faults. These recovery strategies cover faults enumerated by [5] that commonly occur in swarm robots.

From the point at which a fault has been detected and diagnosed, the swarm must decide upon an appropriate recovery strategy. We assume that each robot has the ability to repair other robots in the swarm, therefore the problem reduces to choosing which non-faulty robot(s) should be recruited to repair the faulty robot, and which predefined behaviour is most appropriate given the current scenario. The most appropriate recovery strategy will depend on a number of factors, such as proximity to the faulty robot or remaining battery power, thus some method of assessing the quality of a strategy and its future effect on the swarm is required. I present a solution to this problem that uses machine learning techniques to inform decisions

at run-time based on the results of offline training.

Just as it is represented in the centralised architecture, the q-table is represented as more of a three-dimensional table where the states, action set 1 and action set 2 are stored which can be seen in Figure 5.1.

The swarm's state is defined by the distances of the nearest three robots closest to the faulty robot, their energy levels, the level of importance of the faulty robot, how busy the nearest three robots are, and the distance of the faulty robot to a repair station [d_1, d_2, d_3, b_1, b_2, b_3, p_1, p_2, p_3, I, d_{rs}]. These values represent the input of the learning algorithms that is used to learn the best action for any possible state.

We propose a method that allows robots to learn how to select the most appropriate recovery strategy for any given system state or task. This approach is still 'pre-fault learning' where learning the predefined recovery mechanisms for different possible swarm states before a fault occurs. To test the proposed solution towards fault recovery, we use Autonomous Robots Go Swarming (ARGoS) [99], a widely used swarm robotics simulator. A swarm of 10 foot-bots (a particular configuration of modules based on the marXbot robotic platform [82]) is simulated in a 10x10m arena free from obstacles, undertaking case study behaviours of collective phototaxis, aggregation and foraging.

For the distributed architecture, the importance of the on-board simulator is obvious as we need to run the fault recovery learning process on each robot where they would run the simulations and are also able to store the results and the Q-table. The reason for this is because once the learning process is finished, each robot has to be able to access their 'memory' to recall learnt experiences. As all the experiments in this thesis are run completely in simulation, we can treat the ARGoS simulator as the on-board simulator where we assume that the ARGoS simulator is installed on each robot in the swarm. However, it should be noted that when porting the architecture and learning into the physical robots, that is foot-bots as used in the experiments, ARGoS would not be recommended for use as an internal simulator, as it is computationally too heavy to be run onboard the swarm robots as they are currently designed to be as simple as possible.

One of the properties that is required for the internal simulator is that it has to be able to represent the robots accurately with all the sensors and actuators. E-pucks [3] are more basic robots compared to foot-bots as they were developed primarily for education. Figure 6.1 and figure 6.2 show the features of both e-pucks and foot-bots and it can be observed that foot-bots have more features, meaning it would require more computational power to simulate and run. It has been explained in the previous chapter why it was decided to use foot-bots instead of e-pucks in this thesis to run the experiments; foot-bots have the features that are required for performing

the planned pre-defined recovery strategies.

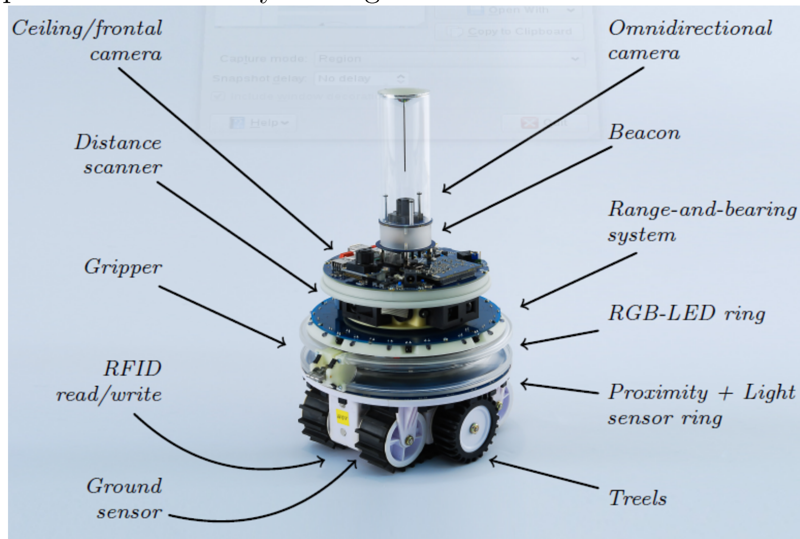


Figure 6.1: This figure represents the properties and features of a foot-bot. Taken from [2]

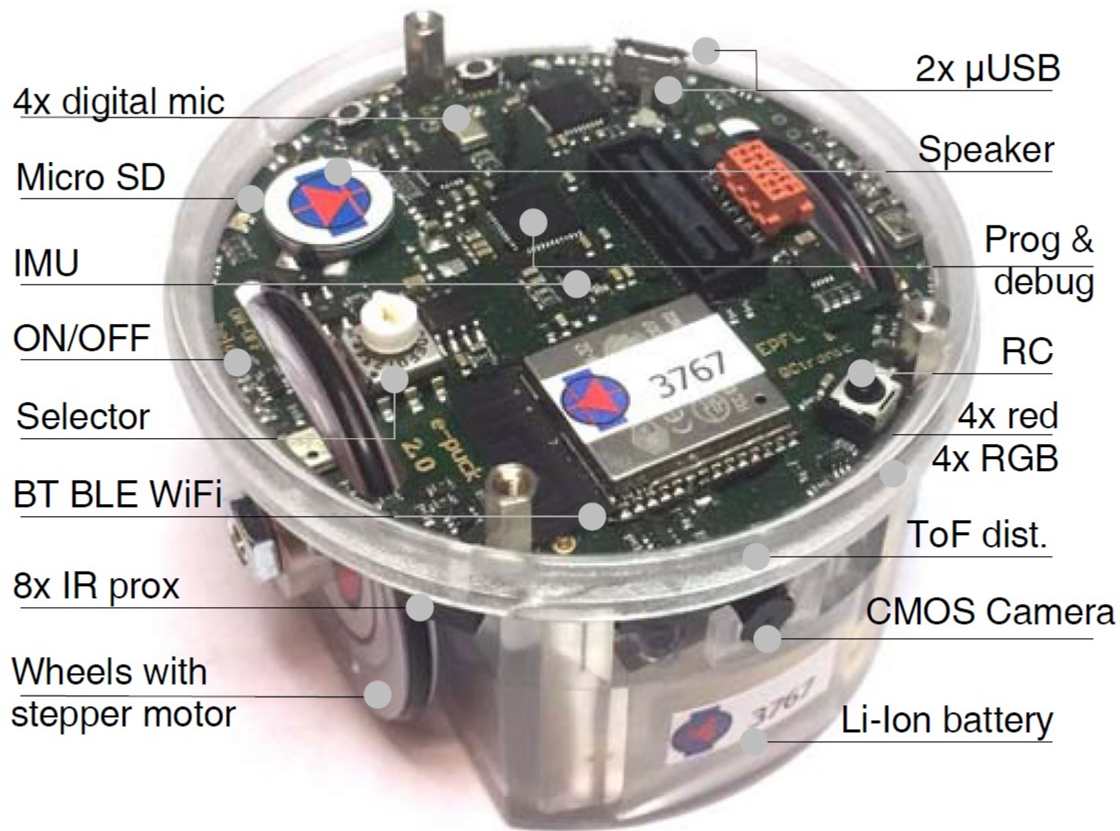


Figure 6.2: This figure represents the properties and features of an e-puck. Taken from [3]

6.3.1 States and Actions

Each robot, this includes the fault robot, is running the simulation on its own internal simulator where it takes the role of the faulty robot in the simulation and generates imagined states which has been discussed previously. To re-iterate, the states that are fed into the learning architecture are:

Distance from the faulty robot d_1 ... d_3 describes how far the nearest robot(s) is from the faulty robot (the robot running the learning simulation on-board the internal simulator).

'How busy' nearby robots are b_1 ... b_3 describes how busy the nearest robot(s) is. The designer can decide how the swarm assigns the 'busy' rating. This is rated on a discrete scale from 0 to 5, where 0 means not busy and 5 signifies

that the robot is very busy. If the robot is not busy, then it can tend to the faulty robot immediately, but as the robot 'busyness' increases, the longer it takes for the robot to be deployed. Although swarm are homogeneous in nature, there are some tasks that different robots have different capabilities and also different sub-tasks. This property is especially useful in these areas; for example in foraging, where some robots are searching for food, some robots have found food items and are carrying them back to the base. During the learning process, the robot performing the learning allocates the 'busy' rating for each of the nearby, non-faulty robots. During the testing phase, the robots communicate with each other about this rating so that they are able to use this in their individual input state to return an optimal solution.

Power left p_1 ... p_3 describes the amount of power left in the robot at the time that the fault is detected. This is in percentage, so as to make calculating the reward easier. Again, the power left in the robots is not globally available where there is no overall global observer overlooking the swarm of robots and the environment. For the robots that are part of the fault recovery process, they need this information from the other robots and therefore would require direct communication to get the necessary information.

Importance of the faulty robot I describes how important a faulty robot is. For example, if it is actively busy with a task, e.g. transporting an object in a foraging or search-and-rescue task, it will be considered more important. The designer can decide how the swarm assigns the importance rating. This is rated on a discrete scale from 0 to 5, where 0 means not important and 5 signifies that the robot is very important. If a faulty robot is not important, we do not necessarily care about how fast the repair is, but at the same time, we want to reduce the overall cost of the repair. This information is also directly communicated with the other 'nearby' robots to allow for the robots to select optimal solutions.

Distance to repair station d_{rb} describes how far the faulty robot is from a repair station (in meters). At the beginning of the task, all robots are given the location of the repair station and the robots keep track of their position from their start point and through put their journey in the environment. Due to this, there is no need to share or directly communicate this information amongst the robots.

Also, our actions are similar to the centralised approach but to re-iterate, there are two action sets:

- Select any combination of the three robots chosen closest to the faulty robot, which will be involved in the fault recovery. There are seven possible robot combinations: (A, B, C, AB, AC, BC, ABC).
- Select one of the four predefined recovery mechanisms below:

  Transport to repair station: There is a repair station where faulty robots can be taken to be repaired. This behaviour involves the assisting robots gripping the faulty robot and dragging it to the repair station. The chosen robot(s) returns to the task, leaving behind the faulty robot to be fixed.

  Repair on the spot: Following [25], we assume that each robot has the ability to repair other robots in the swarm, and that the robots have access to a repertoire of recovery mechanisms which can fix common faults. This behaviour could be especially useful if a faulty robot is very important and needs to resume its task immediately. However, this takes a significant amount of time and energy. Each fault takes a different amount of time to fix, it takes less time to fix the faulty robot if more assisting robots are recruited. However, there is a limit to how many robots make a difference for the 'cost' of the repair.

  Drag Along: This behaviour requires only one robot to drag a faulty robot along. When the 'helper' robot gets to the faulty robot, it grips it and continues on with its task. It should be noted that it takes energy to drag a robot along; therefore it needs to be included when calculating the reward.

  Leader-Follower: This behaviour also requires only one robot and does not work for specific faults: complete/partial motor failure and power failure. The faulty robot copies behaviour of helper robot.

  If multiple robots are chosen as the first action, drag along and leader-follower behaviours are not allowed. The swarm must first select from action set 1 (choosing robots) and then from the action set 2 (repair mechanism).

6.3.2 Rewards

The calculation of the reward during the learning process is similar to how it is calculated in the centralised approach. When quantifying the expense when selecting an action, we use a cost function, defined as a weighted sum of objective measures that can be evaluated in simulation. The reward of the reinforcement learning in this

experiment is based on this cost. The cost is to be minimised is because we want the fault recovery to be done as efficiently as possible.

The cost is calculated based on the time it takes to get to the faulty robot, time it takes to finish the predefined recovery mechanism, the energy it takes to get to the faulty robot, the energy it takes to complete the predefined behaviour. There are also punishment/reward clauses:

- If *all* robots complete the task, reward by reducing the cost (- 1000)
- If only some robots complete the task, punish by increasing the cost (+ 1000)
- Punish robots that take a long time to fix an 'important' robot (+ 1000)
- Punish recruited robot(s) that run out of power before completing the task (+ 1000)

6.3.3 Algorithm

As we had stated earlier, each robot simulates the fault recovery process on its own internal on-board simulator. For this experiment, ARGoS is used as the internal simulator and the algorithm is structured to allow for this. ARGoS is also used to simulate the 'real world' scenario and the results of the learning process is tested with using this simulator. During the learning process, the robots are individually coming up with their own action plans. Each robot generates a set of imagined scenarios to learn from where it is assumed that each robot has different learning experiences and there would be variation across the swarm for the 'imagined' scenarios learnt by each robot.

The general pseudo-code for the reinforcement learning and self-organising map procedure for the learning procedure is described below:

6.3.3.1 Learning Set-Up

We are implementing that ARGoS as the internal simulator and are modifying the centralised learning algorithm. The centralised works where a central computer is the global observer that oversees the learning and has access to all the robots in the swarms' information. The central computer processes the input vector states, performs the training on the self organising maps and populates the general q-table that contains all the information and learning states, parameters and results. Moving forward with distributed learning, we are simulating distribution. On each simulated robot, the existing method is run 10 separate times on the central computer; this is

Algorithm 5 Reinforcement Learning and Self-Organising Maps Procedure

```
1: procedure For Training/Learning
2:     for <All input vectors in training data> do
3:         Send an input vector state X(S_t), to the SOM
4:         Identify the winning unit in the input map
5:         Select possible action using π_t(S_t,A_t)
6:         Choose action, A_t to calculate reward
7:         Receive reward based on performance in environment
8:         Calculate Q-value and update Q-table using update rule in equation (2)
9:         Update the winner unit in SOM using the update rule in equation (4)
10:        Update neighbouring units in the SOM
```

the number of robots in the swarm that initially participates in the learning. Each run or separate time represents running on one robot internal simulator. However, each training of SOM and reinforcement learning results in its own individual SOM and Q-table; meaning that each robot has its own map and q-table that would contain its own set of results. To make this clearer, definitions of the notations used to describe this distributed approach is explained below:

Run This represents each robot internal simulations. For example, in the learning process done in this thesis, 10 robots are used in the learning phase so therefore there are 10 distinct internal simulations, 10 self organising maps and Q-tables which are linked to each robot.

Episodes This represents one learning cycle for one independent input state vector.

Iterations This represents the number of repetitions for one episode.

Training period This represents how many episodes that the learning is trained over per robot or run.

Each run is done on different seed values fed in to the simulator which translates to different starting points in the simulation. These runs are all done independently and they do not affect each other's results in the map and the Q-table. Each run goes through multiple iterations to get mean results. The training period is over 150 episodes; 150 different input state vectors). The learning process per robot consists of a lower training period compared to the centralised approach. In actuality, the distributed approach reduced the learning time overall due to the fewer runs per robot that is done. Conceptually, having the distributed algorithm structured in this

Parameter	Value
Arena size	10x10 metres
Size of SOM	3x3 units
Distributed Agents involved in learning	10
Number of states in training data	150 per robot
Simulation time	1300 seconds
SOM Input map learning rate, β	0.6
Number of state-action repetitions	30
Q-learning learning rate, α	0.2
Exploration factor, ϵ	0.3
Neighbouring function	$\sigma = \sigma_o exp(\frac{-t}{\lambda})$

Table 6.1: The table describes the parameters used for the learning process. All Q-values are set to zero and the weights if the input SOM are initialised randomly in the range [0,1]. The set of parameters described above were empirically derived. The value σ represents the width of the neighbourhood which shrinks over time, σ_o represents the initial width of the neighbourhood, t represents the current iteration loop, λ represents the time constant.

form equates to an ideal, general distributed learning scenarios where each robot is performing the learning on-board its own internal simulator.

It should be noted that the distributed system has a smaller area and the comparison with the centralised approach is less critical; this is because the centralised approach can be considered in a developmental context.

The table 6.1 describes the parameters of the learning process.

Algorithm 6 describes the complete learning process.

The next sub-section describes the experimental set-up for testing the fault recovery learnt process. Additionally, we describe the algorithm of how the testing for the different tasks is implemented.

6.3.4 Experimental Setup

The work done is for 10, 20, 40 robots in a swarm with a certain number of faults introduced at a specific time during the experiment. For the learning, the system was setup with a swarm of 10 robots and one fault type is injected at one time during the learning. For each state, the simulation is run 100 times for the selected

Algorithm 6 Learning process

1: **procedure** For Training/Learning per repetition
2: Randomise seed value to be used in simulation
3: Initialise ARGoS simulation using state from training data.
4: Choose action from action set (1): move to faulty robot
5: Calculate cost for taking action
6: Choose action from action set (2): run recovery mechanism and calculate cost
7: Calculate total reward
8: Update the *Q-values* in the Q-table
9: Terminate the episode after a fixed duration

actions and the mean of the rewards are thereafter calculated and used in the Q-value calculations.

The system was tested on 50 randomly generated scenarios where each scenario is run with 100 different random seeds and the average performance is reported. We consider four different treatments that describes the steps taken by the swarm when getting results and is described in the list below. This is done for three possible failure modes: motor failure, communication failure, light sensor failure.

It should also be noted that the robots are not allowed to leave any robot behind; they have to select from any of the predefined behaviours. To iterate, from the beginning of the task, the swarm generates imagined scenarios, using a simulator, and learns the best recovery strategy from them before a fault actually occurs and is detected. When a fault is detected, the swarm selects the best recovery strategy based on what has been learnt during the imagined scenarios.

This section describes how direct communication between the robots in the swarm can be implemented. As discussed earlier, direct communication is to be utilised in the context of sharing information between the robots to allow for the selection of the optimal solution and also for generating a general consensus amongst the robots involved in the fault recovery.

When using direct communication to share information, the information shared is dependent on what robots would need the information, faulty or recruited nearby robots; not all robots need the same information. We use radios to broadcast the messages across the swarm; this is available on the ARGoS simulator platform. Other possible means of achieving direct communication with the swarm could be wireless communication or bluetooth. There are some extension boards that can be installed on-board the robot that has some of these features. These additional features can be

Robot Type	Information Broadcasted	Information Needed
Faulty Robot	Importance Rating	Importance Rating from Faulty Robot, Distance from Faulty Robot for R_a, R_b and R_c, 'How busy' rating for R_a, R_b and R_c, Power Left for R_a, R_b and R_c
Recruited Robot [R_a]	Distance from Faulty Robot, 'How busy' rating, Power Left	Importance Rating from Faulty Robot, Distance from Faulty Robot for R_b and R_c, 'How busy' rating for R_b and R_c, Power Left for R_b and R_c
Recruited Robot [R_b]	Distance from Faulty Robot, 'How busy' rating, Power Left	Importance Rating from Faulty Robot, Distance from Faulty Robot for R_a and R_c, 'How busy' rating for R_a and R_c, Power Left for R_a and R_c
Recruited Robot [R_c]	Distance from Faulty Robot, 'How busy' rating, Power Left	Importance Rating from Faulty Robot, Distance from Faulty Robot for R_b and R_c, 'How busy' rating for R_b and R_c, Power Left for R_b and R_c

Table 6.2: This table describes the information on what in transmitted and broadcasted between the robots during the recovery process

implemented as a fail-safe incase of radio failure so that the messages can be broadcasted across the swarm. The table below describes the information broadcasted on the radios.

All robots within the range of any information being transmitted has the ability to receive the information if the radio is enabled. To ensure that the robots involved in the fault recovery make use of just necessary information, each information is prefixed with the type of information is being sent; with this, when a robot receives a transmitted message, it looks up the prefix to check if this is the information it requires. If it is not, it ignores this information and waits until the information needed is transmitted.

Each robot labels itself, A, B, C and their respective information matches each individual robot. This is also broadcasted so that when using the state information to select the appropriate fault recovery strategy, the correct sequence of information is used.

Finally, when the robots have come to a solution, they broadcast their solution (how many robots, what combination of robots and what predefined behaviour would be used in the fault recovery). This information is represented in the Q-table so

Algorithm 7 Testing process

1: **procedure** FOR TESTING
2: Fault is detected and diagnosed
3: Input Vector State is collected (Information is transmitted and received from surrounding robots), $X(S_t)$
4: Unit of SOM with smallest distance from input vector state is winner unit
5: Winner unit is identified in Q-table; action with smallest Q-value is selected
6: Action with smallest Q-value is selected; best learnt recovery strategy selected for that particular state
7: Broadcast solution to other robots; receive other solutions from other robots
8: Based on other solutions, select action with highest number of votes and implement

they compare what positions on their individual Q-tables. As we follow the major consensus rule, after receiving the information, the number of recovery strategies that is chosen more is selected as the winner. If there are equal number of possible strategies selected, then either solution is chosen. If all robots involved with fault recovery robots come up with different solutions, again any of the solutions are chosen though it is unlikely for this specific scenario to occur. The reason is because the input map is trained similarly across all the robots so the states are structured similarly; we expect similar states to be chosen across all the maps.

The following list describes the steps taken by the swarm from the beginning of the task to the end during the learning and testing phases.

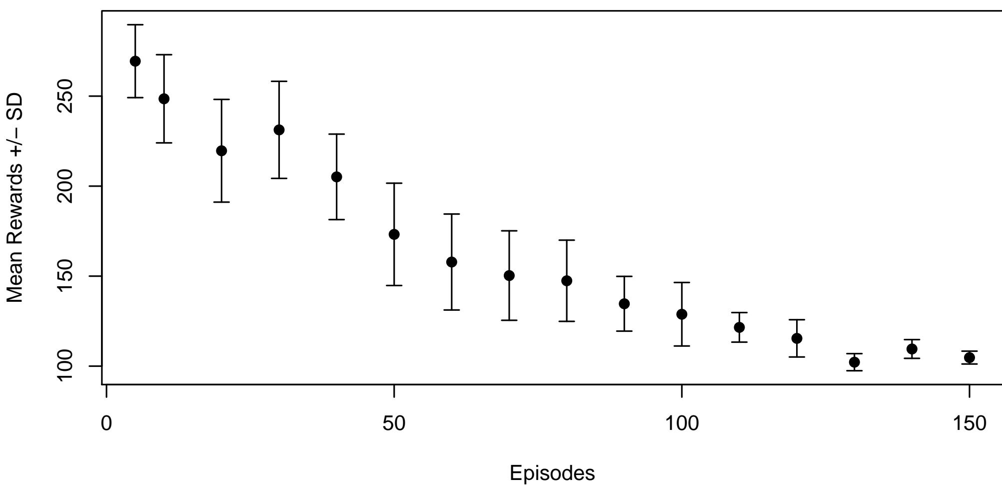
- Run experiments for collective phototaxis, aggregation and foraging with no faults injected to give baseline performance
- Then test performance with two faults injected with faults; this is enough to break behaviour.
- Test the performance random selection of actions. This might help to recover swarm but could be suboptimal.
- Then test performance on the SOM and RL solution.

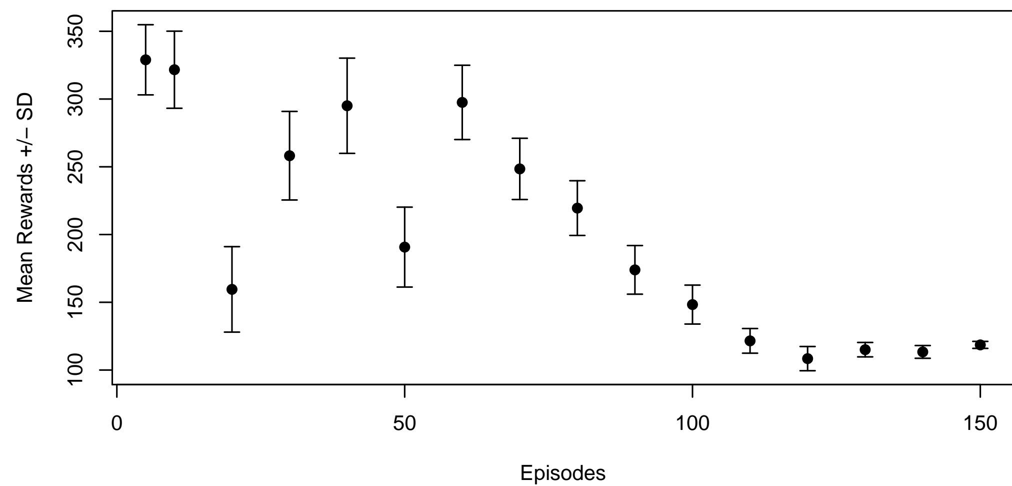
The algorithm to test the SOM + RL infrastructure is as follows:

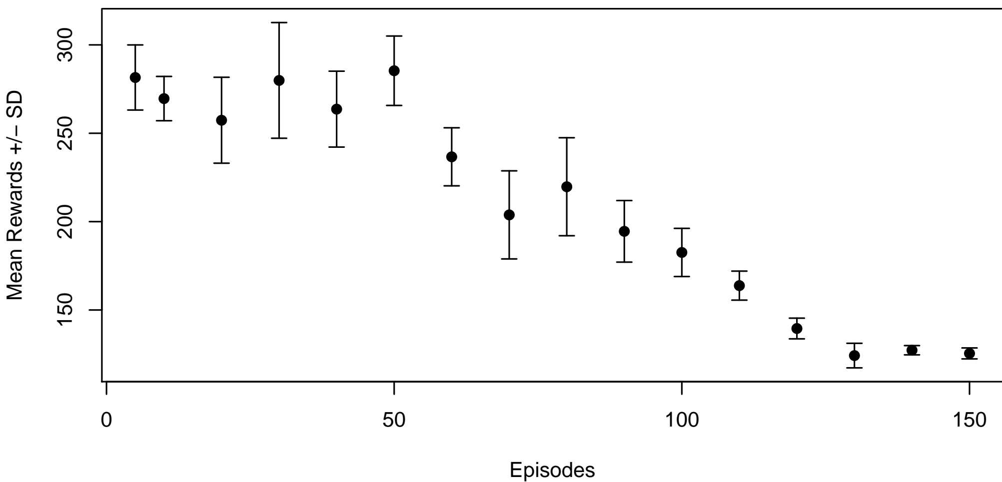
Algorithm 7 describes the testing process for the distributed process as it differs from the centralised approach because the input state information is not readily available to all the robots. As described earlier, the necessary information has to be shared amongst the robots.

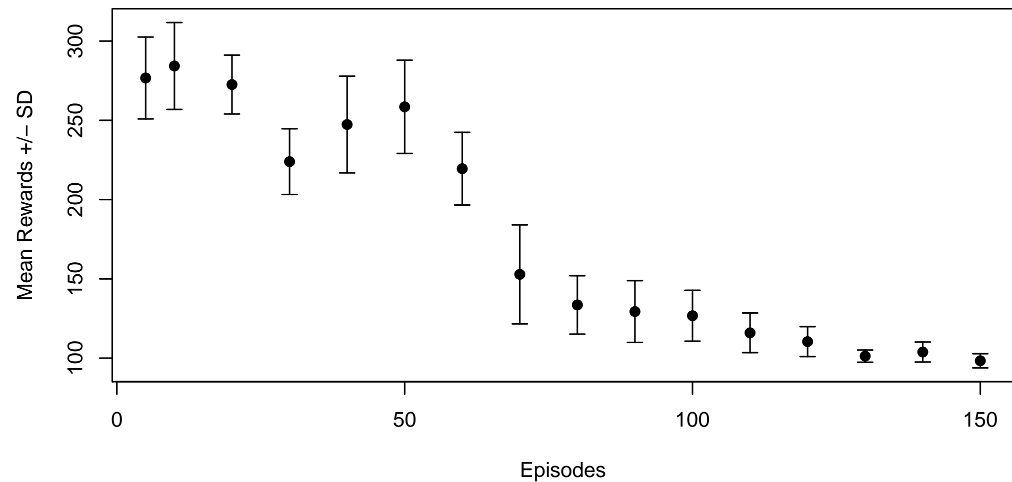
6.3.5 Results

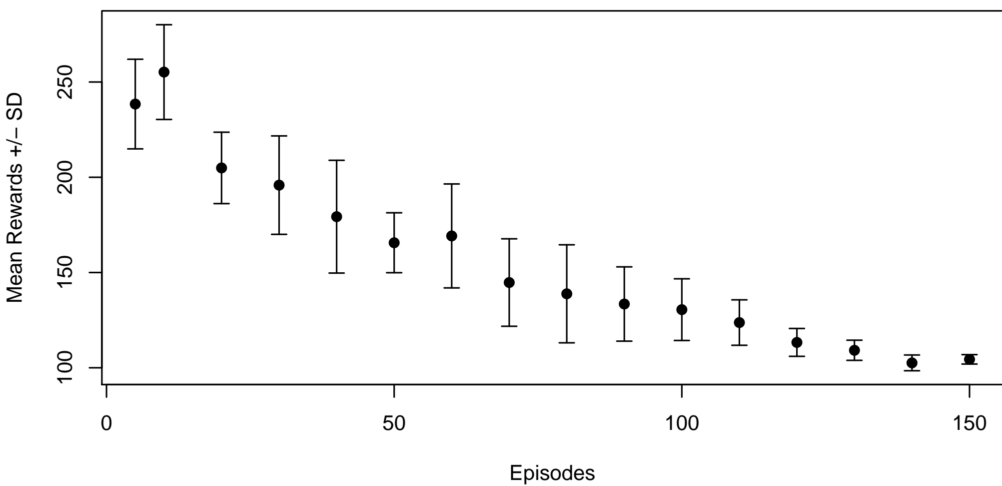
The set of graphs in Figure 6.3 describes the total average rewards over 150 episodes for each robot. During each episode, the swarm generates the imagined scenario which is represented as the state that goes into the SOM and RL. The swarm thereafter selects the actions from both action sets and the cost from selecting these actions are calculated. As can be seen from the graph, at the beginning, the rewards fluctuate heavily as the swarm is exploring different actions because actions are not learnt yet. As the episodes go on, the swarm learns more and starts to select actions more intelligently, less exploratory and more exploiting though it still strikes a balance between exploration and exploitation. Towards the end, we can see that the rewards level out, as the swarm starts to pick more optimal solutions based on previous rewards for the different states. Each point on the graph represents the mean over 30 repetitions for the same state.

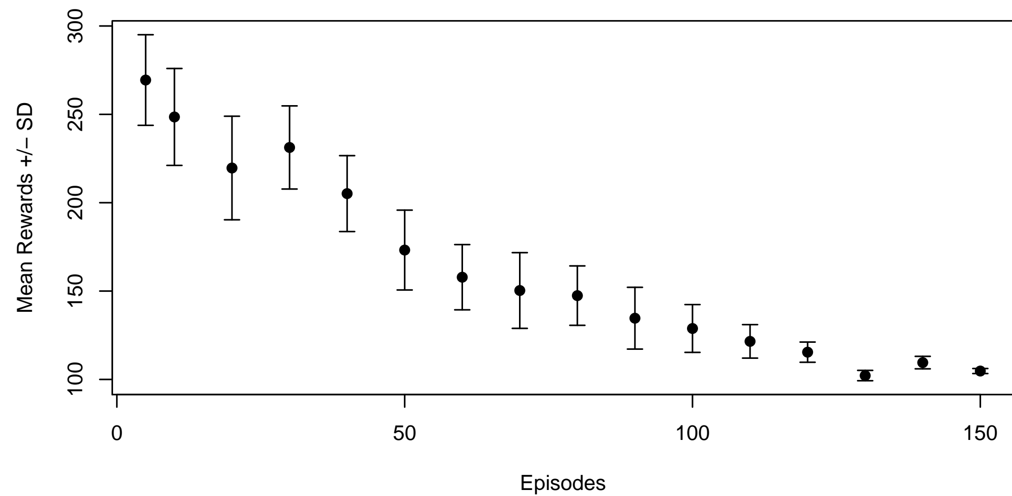

Graph describing mean rewards over episodes (Robot 1)
Mean Rewards +/− SD
Episodes
0
50
100
150
100
150
200
250

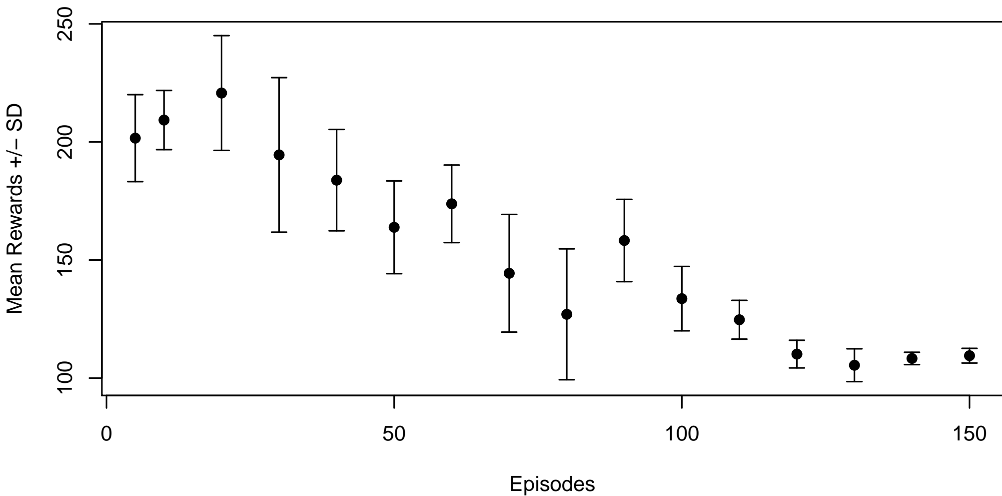

Graph describing mean rewards over episodes (Robot 2)
Mean Rewards +/− SD
Episodes
0
50
100
150
100
150
200
250
300
350

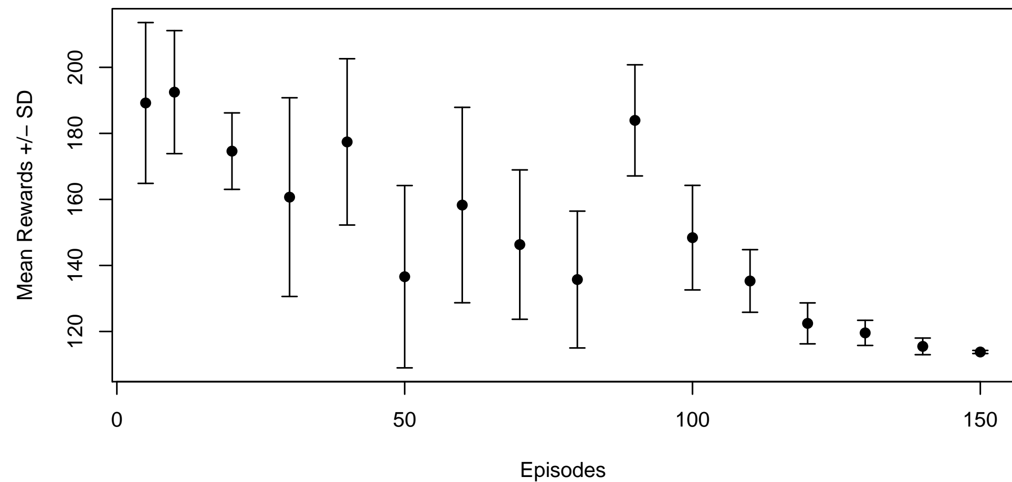

Graph describing mean rewards over episodes (Robot 3)
Mean Rewards +/− SD
Episodes
0
50
100
150
150
200
250
300

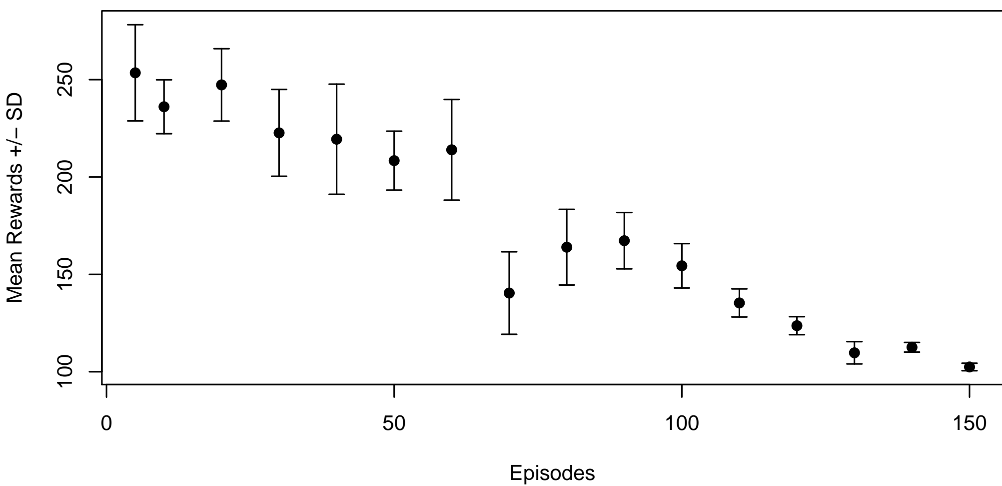

Graph describing mean rewards over episodes (Robot 4)
Mean Rewards +/− SD
Episodes
0
50
100
150
100
150
200
250
300

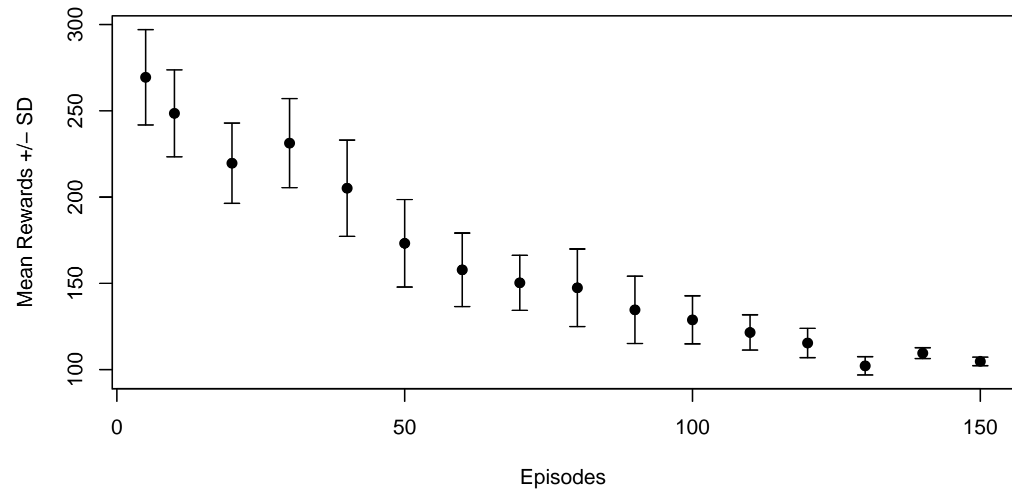

Graph describing mean rewards over episodes (Robot 5)
Mean Rewards +/− SD
Episodes


Graph describing mean rewards over episodes (Robot 6)
Mean Rewards +/− SD
Episodes


Graph describing mean rewards over episodes (Robot 7)
Mean Rewards +/− SD
Episodes


Graph describing mean rewards over episodes (Robot 8)
Mean Rewards +/− SD
Episodes


Graph describing mean rewards over episodes (Robot 9)
Mean Rewards +/− SD
Episodes
100
150
200
250
0
50
100
150


Graph describing mean rewards over episodes (Robot 10)
Mean Rewards +/− SD
Episodes
100
150
200
250
300
0
50
100
150

Recovery Type Chosen	Average Percentage Chosen
Drag to Base	31%
Fix on the Spot	25%
Drag Along	29%
Leader-Follower	15%

Table 6.3: The table describes the percentage of the test input that is selected during the testing phase of the RL + SOM and the recovery type selected. The recovery type column represents what predefined behaviour was chosen for the 50 sample inputs. The percentage column represents what percentage of the test inputs selects the corresponding recovery type.

Figure 6.3: Rewards Convergence for each Robot

This is tested for different sizes in the swarm (10,20 and 40 robots) to test the scalability of the learnt recovery strategy. It is also tested for three tasks aggregation, foraging and collective phototaxis. The results for the robot swarm sizes, 20 and 40 are in the appendixes.

6.3.5.1 10 Robots in the swarm

Table 6.5 shows that for the 50 test states, 31% of the states chose the strategy to drag to base, 25% chooses to fix on the spot, 29% chooses to drag the faulty robot along and 15% chooses to follow the leader. The reason why the Leader-Follower is chosen less is because it is limited for what faults it can be used on while the other predefined behaviours can be used on all the faults. The swarm picks the best recovery strategy based on the fault type and also the input state. The swarm never leaves a robot behind, they select the best strategy that has been learnt by the swarm. These results are collected across collective phototaxis, aggregation and foraging.

Collective Phototaxis

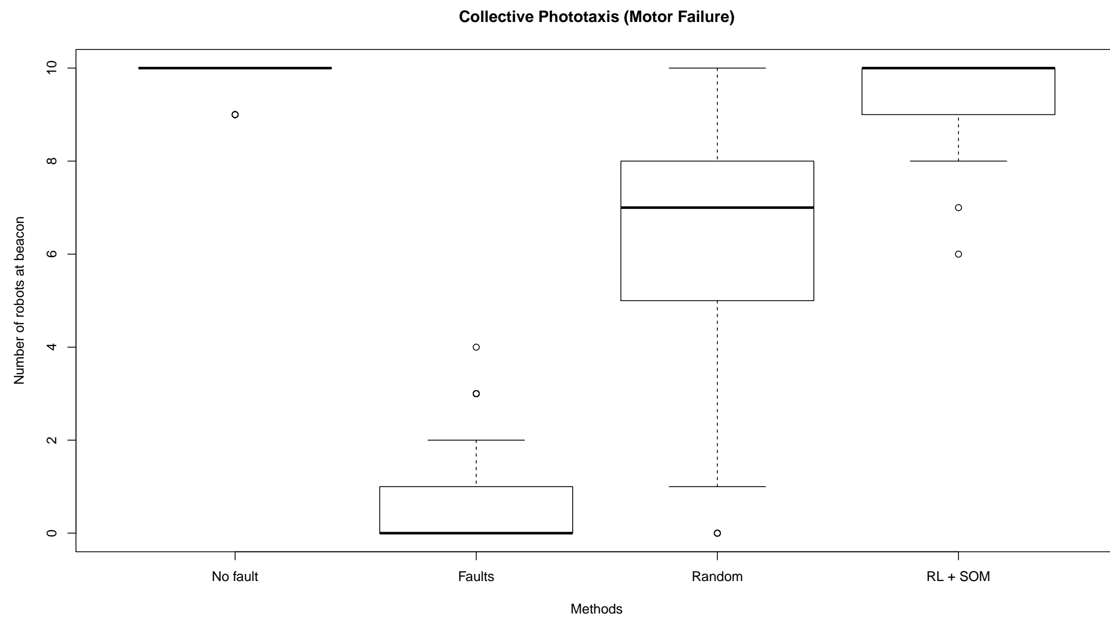


Figure 6.4: Collective Phototaxis: Motor failures

Figure 6.4 describes a box plot showing the number of robots that arrive at the beacon or are at the repair station (this depends on what recovery mechanism is used). As can be seen, when there are no faults in the swarm, all of the robots are able to make it to the beacon. When there are faults, the faulty robots anchors the healthy robots as the sensors on the robot are still functioning. We can observe that some robots make it to the light source; they escape the anchoring effect of the faulty robots. When a random behaviour is chosen, the robots make it to the light source or repair station a times while other times, the worst behaviour is chosen which is why we see a range of results. Finally, when the learnt recovery strategy is chosen, more effective strategies are chosen and most of the robots are able to make it to the beacon or repair station.

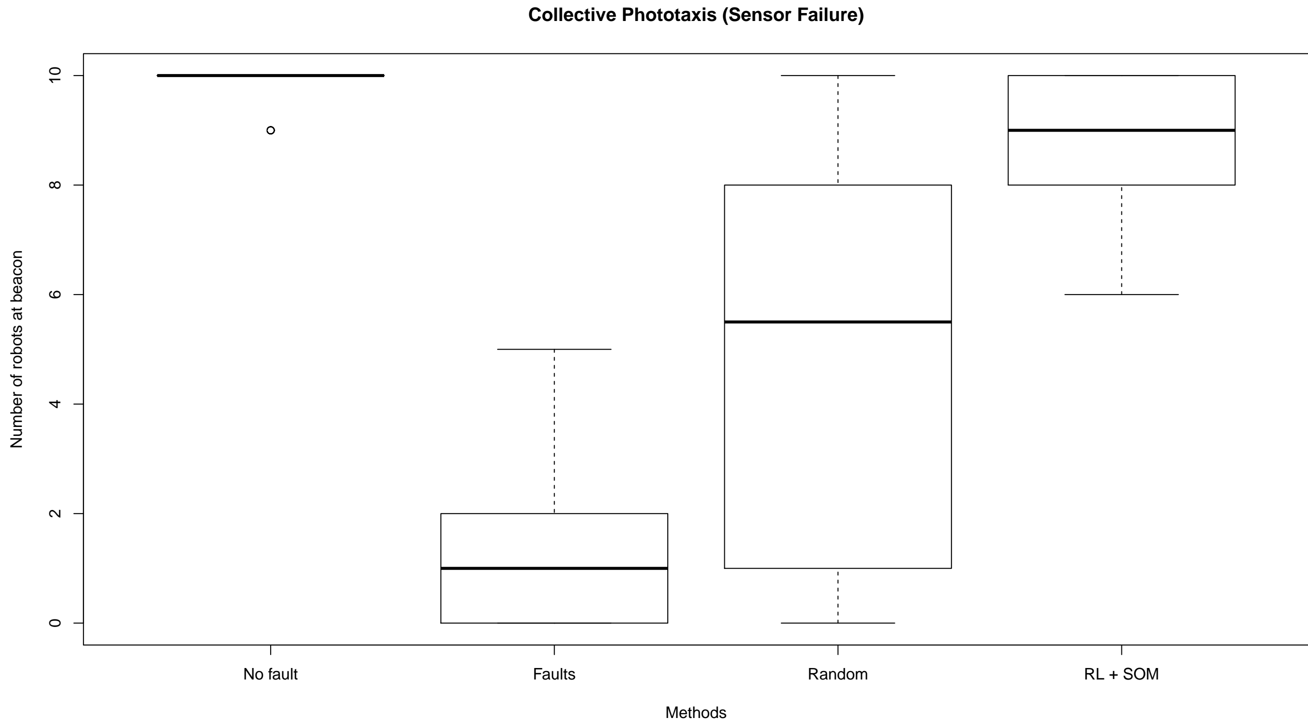


Figure 6.5: Collective Phototaxis: Communication sensor failures

From Figure 6.5, communication sensor failure can be observed; the communication sensors being described are the range and bearing sensors that are used in the robots to locate other robots and the robots use these sensors for cohesion and avoidance. Again, the results show that when no faults are injected, the robots are able to make it to the light source. When the communication fault is injected, the faulty robots move randomly around the environment and the 'healthy' robots are not able to complete task as they anchor and follow around the faulty robots. Choosing a random behaviour shows fault recovery solutions that work sometimes while other times, they do not work. Finally, when the learnt recovery strategy is chosen, more effective strategies are observed to be chosen and majority of the robots are able to complete the task and no robot is left behind.

Aggregation

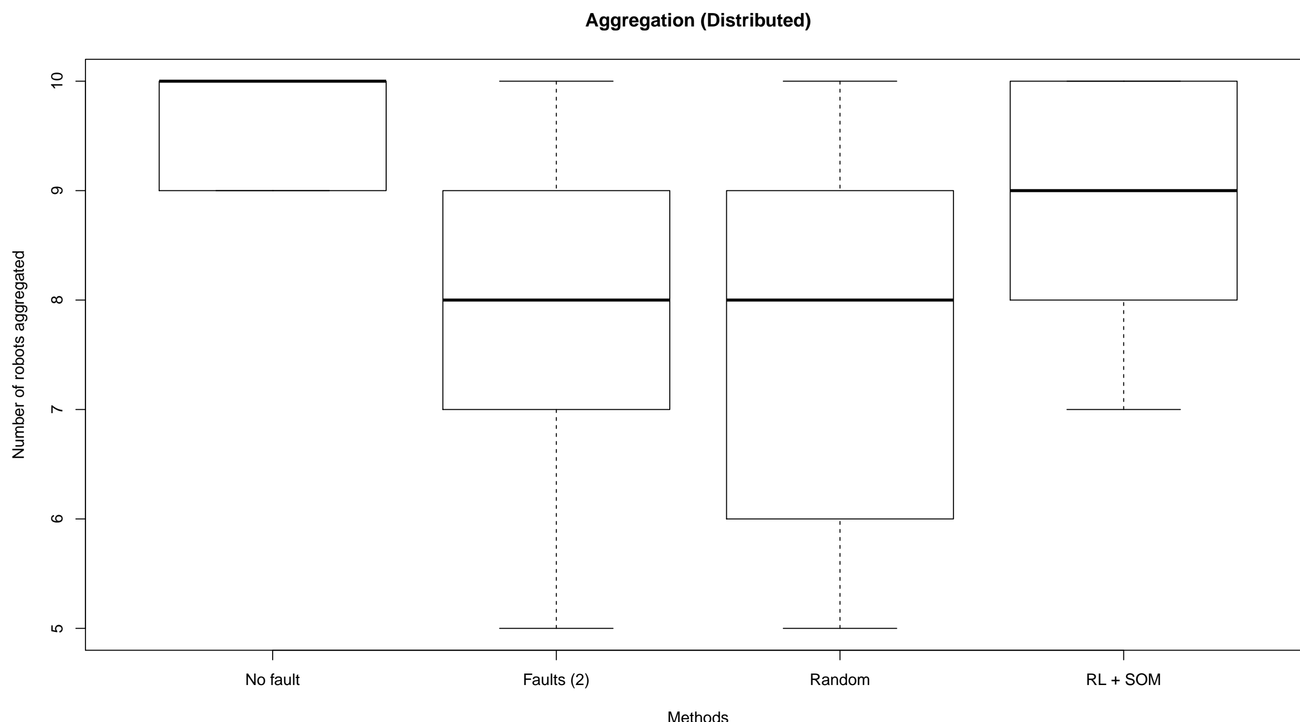


Figure 6.6: Results for aggregation when the range and bearing sensors (Communication sensors) on the robots are faulty.

Figure 6.6 describes the number of robots in aggregates in the environment or at the repair station; depending on the learnt recovery mechanism used. When no faults are present in the system, all robots are able to aggregate together to form one aggregate. As it is discussed in the centralised approach, the discrepancy with the number of robots in an aggregate is due to how compact the robots are when forming the aggregate. The robots are continuously moving even after forming complete aggregates. When faults are present, the robots are not able to make use of their sensors to perform cohesion to create one aggregate; the robots break and form multiple aggregates. This happens because robots move randomly around the environment and the robots become divided between the faulty robots forming smaller collections of the robots. Again, randomly selecting a behaviour shows that it selects a good strategy sometimes but in most cases, it chooses a bad strategy. When the learnt predefined behaviour strategy is implemented as the chosen recovery method, we can see that the swarm recovers successfully which allows the robots form one cohesive

aggregate.

6.3.5.1.1 Foraging

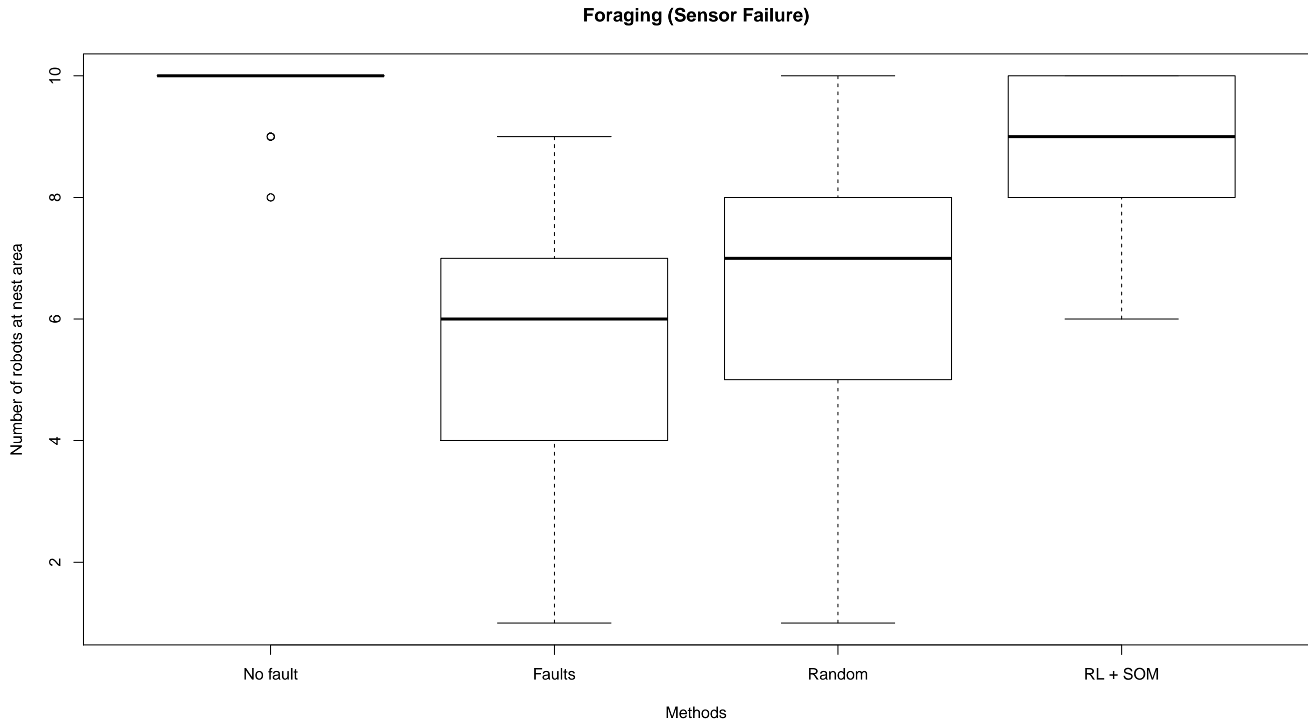


Figure 6.7: Results for foraging when light sensor failures are injected: Number of robots that reach the nest at the end of the task.

Figure 6.7 describes the performance of the foraging task when the light sensor faults are injected. The robots in the swarm, when performing foraging, use lights to determine the location of the nest. Therefore when there is a fault, the faulty robot is unable to sense the light and therefore cannot make it to the nest. The faulty robots wander aimlessly around the environment and reduces the probability that other robots would find and collect the rest of the items. It is observed that between randomly selecting a predefined behaviour and using the learnt recovery strategy, the learnt strategy performs better overall because when randomly selecting a solution, sub-optimal or bad strategies are chosen.

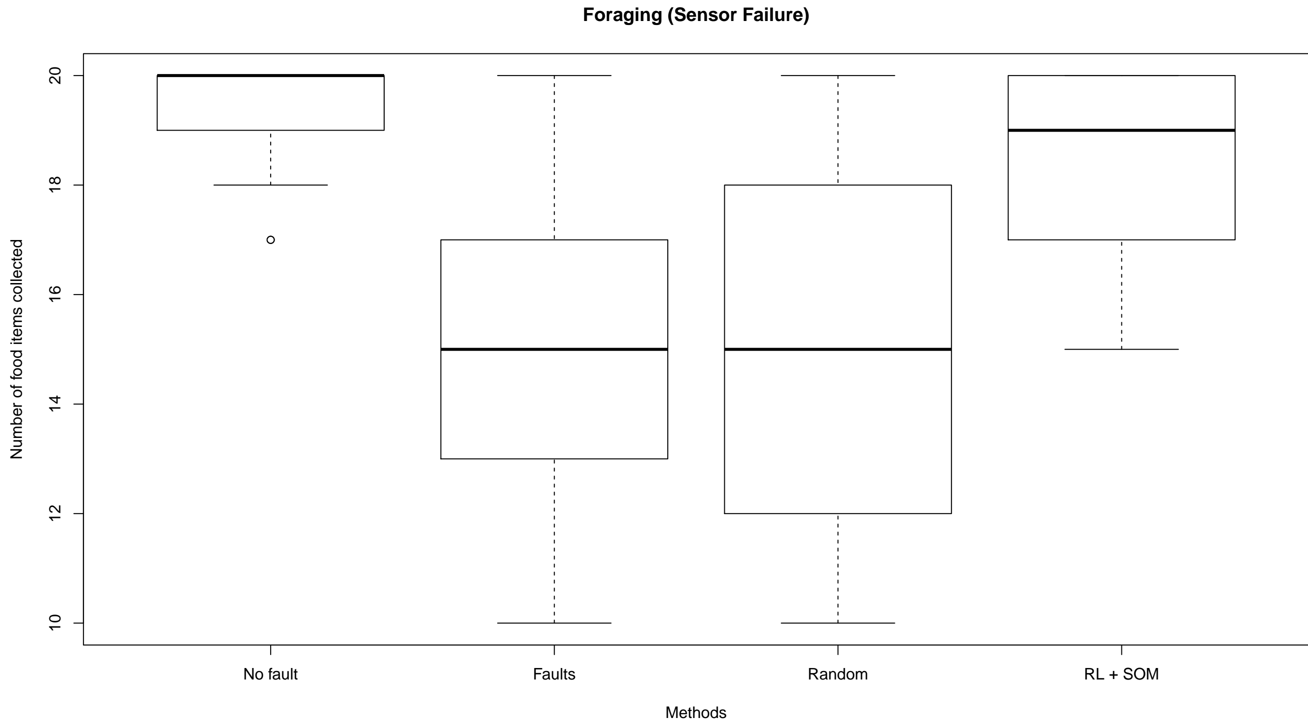


Figure 6.8: Results for foraging when light sensor failures are injected: Number of items collected in the environment

Figure 6.8 shows the results for when amount of 'food items' collected in the environment. In the no-fault scenario, the swarm is able to collect most, if not all of the items in the environment and most of the robots make it back to the nest. When the is fault injected, it can be seen from the results that few 'food items' are collected. When randomly selecting a predefined recovery strategy, we can see that there is a range of results where sometimes some items are picked; the reason is because by randomly selecting a behaviour, sub-optimal or bad strategies are chosen. Finally, when using the learnt recovery strategy solutions, it can be observed that most of the items in the environment are collected because most of the robots are functioning and are able to complete the task as can be observed in Figure 6.7.

6.3.6 Inclusion of Obstacles

In the previous section, we discussed the work done in an uncluttered environment where we extended the centralised architecture to a more distributed approach. The main reason for this is to test the architecture without any obstructions to check the validity of the distributed approach and how well it would perform. The learning procedure is tested with the same fault types that are shown in the above results. It should be noted that the object inclusion for the distributed learning environment would be different because there is no global overseer that has a global view of the environment therefore the obstacle inclusion would be restricted.
There are additional aspects that need to be considered when looking at including obstacles into the environment:

State Space Increase As the local environment is cluttered, there are new states that the robots would need to consider when selecting the appropriate recovery strategy. The new states include: a blockage obstructing the robots from getting to the repair station and a fault diagnosis state that describes what type of fault is being learnt from. It should be noted that we are assuming that the swarm has the capacity to detect and diagnose faults. In this distributed approach, the larger the state space, the more episodes would be needed to have the rewards converge.

Reinforcement Learning Rule The selection of action and state space has been straightforward so far, however, with the introduction of new states, there is a need to increase the complexity of the reinforcement learning rule. In this new scenario, actions can now change before the episode finishes, only if there is something actively interrupting the 'helper' robots from completing the recovery process. The robots monitor themselves continuously, and if they meander around an obstacle for longer than 10 seconds, change the action. Again, in this situation, the robots would directly communicate with each other about the 'meandering'. When this occurs, only the robots already involved in the recovery would select new behaviour to enable the swarm to complete the task. This prevents the 'helper' robot from meandering till it runs out of energy. This would allow the 'helper' robot to change action to something more suitable for the present scenario. From the action selected, you have your new state, get the reward which then takes you to a new action and you get your new reward.

6.3.7 States and Updated Rewards

For this new learning scenario, the robots are initialised based on the input vector state $X(S_t)$ chosen. Some of the input vector state features remain the same as in an uncluttered environment with some additional features included in the state space. The input state $X(S_t)$ is defined as: $[d_1, d_2, d_3, b_1, b_2, b_3, p_1, p_2, p_3, I, d_{rs}, f_d, n_o, p_{o1}, p_{o2}, p_{o3}, p_{o4}, p_{o5}, p_{o6}, p_{o7}, p_{o8}, p_{o9}, p_{o10}, b_{y/n}, p_b]$.

Distance from the faulty robot d_1 ... d_3 describes how far the nearest robot(s) is from the faulty robot.

'How busy' nearby robots are b_1 ... b_3 describes how busy the nearest robot(s) is. The designer can decide how the swarm assigns the 'busy' rating. This is rated on a discrete scale from 0 to 5, where 0 means not busy and 5 signifies that the robot is very busy. If the robot is not busy, then it can tend to the faulty robot immediately, but as the robot 'busyness' increases, the longer it takes for the robot to be deployed.

Power left p_1 ... p_3 describes the amount of power left in the robot at the time that the fault is detected. This is in percentage, so as to make calculating the reward easier.

Importance of the faulty robot I describes how important a faulty robot is. For example, if it is actively busy with a task, e.g. transporting an object in a foraging or search-and-rescue task, it will be considered more important. The designer can decide how the swarm assigns the importance rating. This is rated on a discrete scale from 0 to 5, where 0 means not important and 5 signifies that the robot is very important. If a faulty robot is not important, we do not necessarily care about how fast the repair is, but at the same time, we want to reduce the overall cost of the repair.

Distance to repair station d_{rs} describes how far the faulty robot is from a repair station (in meters).

Fault diagnosis f_d describes the type of fault that has occurred in the swarm. In this scenario, we use numbers to represent the type of fault. 0 - Complete Motor failure, 1 - Communication Failure, 2 - Partial Sensor Failure, 3 - Complete Sensor Failure, 4 - Complete Power Failure. Although we are using simple integers here to represent the fault types, ideally, features of the robot behaviour after the fault has been detected would be used here. As we are assuming that fault detection and diagnosis are available, if a fault that has been identified

is not part of the 'commonly known faults', it should choose the closest features that best represents the fault from the fault diagnosis. This is randomly generated.

Blockage $b_{y/n}$ describes whether a blockage would initialised in the simulation. It's a 0 or 1 option. The size of the box is fixed. During learning, this state is always known however during testing, as this is a distributed learning environment, the learning agents (robots) do not have a global view of the environment. During testing, the two states that involve the blockage are set as NULL until a blockage is detected and this is thereafter fed into the input state vector to get the appropriate fault recovery solution.

Position of blockage p_b describes where in the section of the arena illustrated below, the blockage is initialised.

The predefined fault recovery behaviours remains the same in this cluttered environment.
The calculation of the updated reward is similar to the previous rewards calculation where there are no obstacles. To iterate, the reward of the reinforcement learning in this experiment is based on this cost. The cost is to be minimised is because we want the fault recovery to be done as efficiently as possible. The cost is calculated linearly based on the time it takes to get to the faulty robot, time it takes to finish the predefined recovery mechanism, the energy it takes to get to the faulty robot, the energy it takes to complete the predefined behaviour. As it has been stated earlier, the actions can change mid-execution depending on if they are obstructed by any obstacles and are unable to finish the recovery strategy. However, we do not want the actions to change mid-execution as this contributes to power used during the recovery process. In addition to the present punishment/reward clauses, more reward clauses are added due to the added complexity of this experimental setup.

- If *all* robots complete the task, reward by reducing the cost (- 1000)
- If only some robots complete the task, punish by increasing the cost (+ 1000)
- Punish robots that take a long time to fix an 'important' robot (+ 1000)
- Punish recruited robot(s) that run out of power before completing the task (+ 1000)
- Punish recruited robot(s) that change action
- Reward recruited robot(s) that finish recovery strategy (action set 1 and action set 2) without changing any acton mid-execution.

Parameter	Value
Arena size	10x10 metres
Size of SOM	5x4 units
Distributed Agents involved in learning	10
Number of states in training data	550 per robot
Simulation time	1300 seconds
SOM Input map learning rate, β	0.6
Number of state-action repetitions	30
Q-learning learning rate, α	0.2
Exploration factor, ϵ	0.3
Neighbouring function	$\sigma = \sigma_o exp(\frac{-t}{\lambda})$

Table 6.4: The table describes the parameters used for the learning process. All Q-values are set to zero and the weights if the input SOM are initialised randomly in the range [0,1]. The set of parameters described above were empirically derived. The value σ represents the width of the neighbourhood which shrinks over time, σ_o represents the initial width of the neighbourhood, t represents the current iteration loop, λ represents the time constant.

6.3.8 Learning Setup

The learning setup is similar to the setup where there are no obstacles. The learning is done with 10 robots where the states are set and initialised in the swarm. The learning process is run on the phototaxis task. In the previous chapter, the distributed terminology was explained therefore it is going to be used here in describing the learning and testing process. For each state, the simulation is run 30 times for the selected actions and the mean of the rewards are thereafter calculated and used in the Q-value calculations. Additionally, each run (each robot) generates 400 states; this is more than the previous chapter because of the added complexity of the state space.

Table 6.4 describes the updated learning process setup where it can be observed that most of the present learning setup is the same except for the SOM size and the number of states in the training data. These properties needed to be increased due to the new complexity of the states.

The learning algorithm for this is similar to the algorithm where there are no obstacles however, there is an added component where the action can change mid-execution. It

Algorithm 8 Learning process

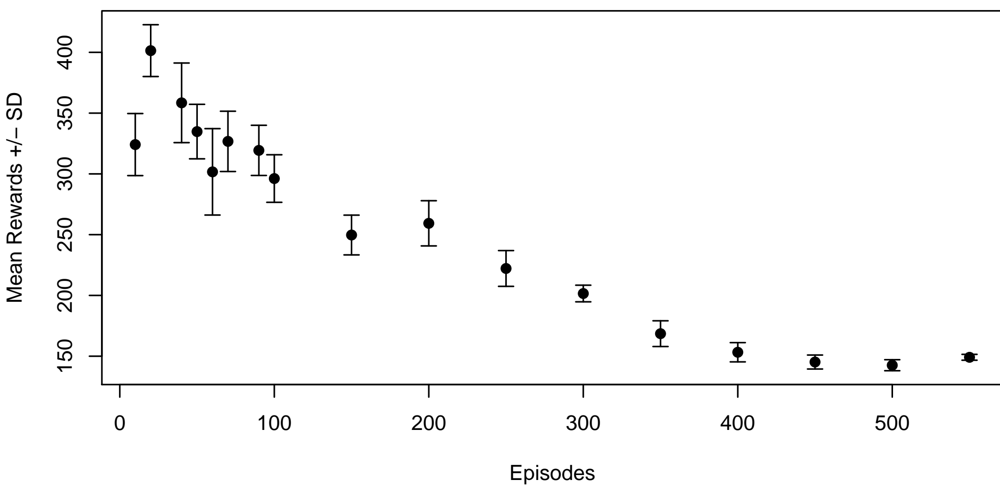
1: **procedure** For Training/Learning
2: Initialise ARGoS simulation using state from training data.
3: Choose action from action set (1): move towards faulty robot
4: If obstructed by obstacle (status does not change in 10 secs), choose another action from action set (1)
5: Calculate cost for taking action
6: Choose action from action set (2):
7: If obstructed by obstacle (status does not change in 10 secs), choose another action from action set (2)
8: Run recovery mechanism and calculate cost
9: Calculate total reward
10: Update the *Q-values* in the Q-table
11: Terminate the episode after a fixed duration

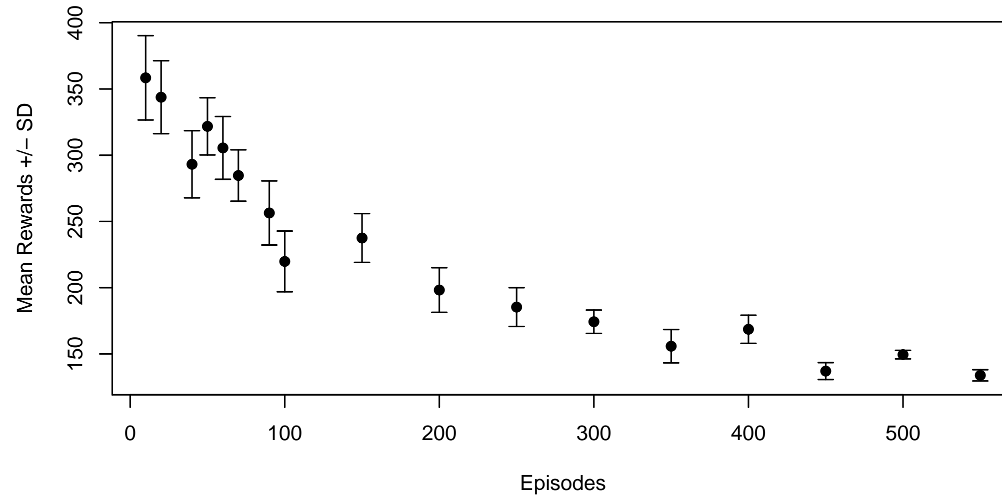
should be noted that the learning process is done on each robot's internal simulator and each robot is a global observer within its own internal simulation and can set the state features within the learning environment. Algorithm 8 describes the updated learning algorithm for this second experimental setup.

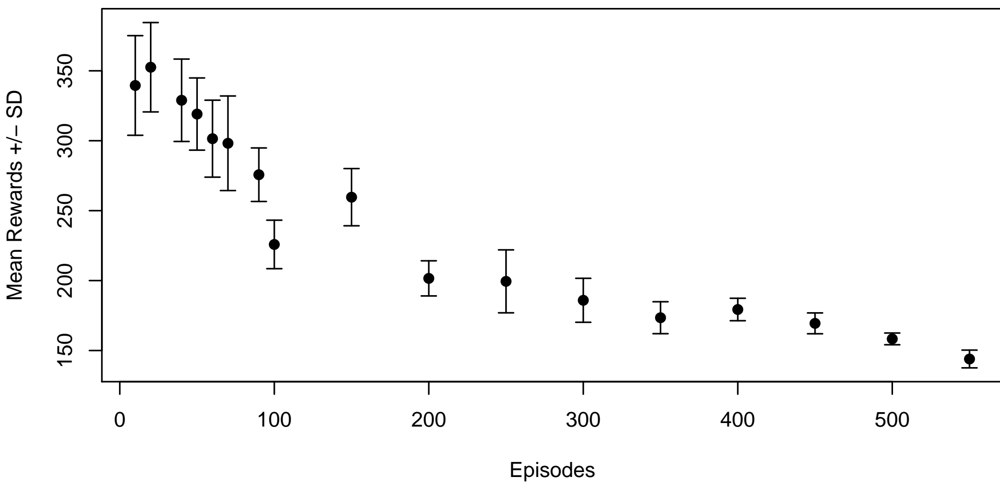
When testing the learning process, the experimental setup is similar to the setup discussed in the experimental setup. One of the most important aspects needed in the testing process is for the robots involved in the recovery process to directly communicate with each other to share the information that is needed to allow for a more efficient learnt recovery strategy selection. The additional information that is transmitted and received amongst the robots is the location of the obstacles within each robot's view in the environment. During the learning process, the sequence of the obstacle positions are put into the learning maps. During the testing phase, the obstacle positions must also be put in the same sequence therefore all the obstacle information is required before the best strategy is chosen. The input map is not flexible therefore the number of obstacles is set at a maximum of 10. However, if less than 10 obstacles are present in the environment, the default value is null. The faulty robot also needs to transmit the type of fault that has occurred during the task.

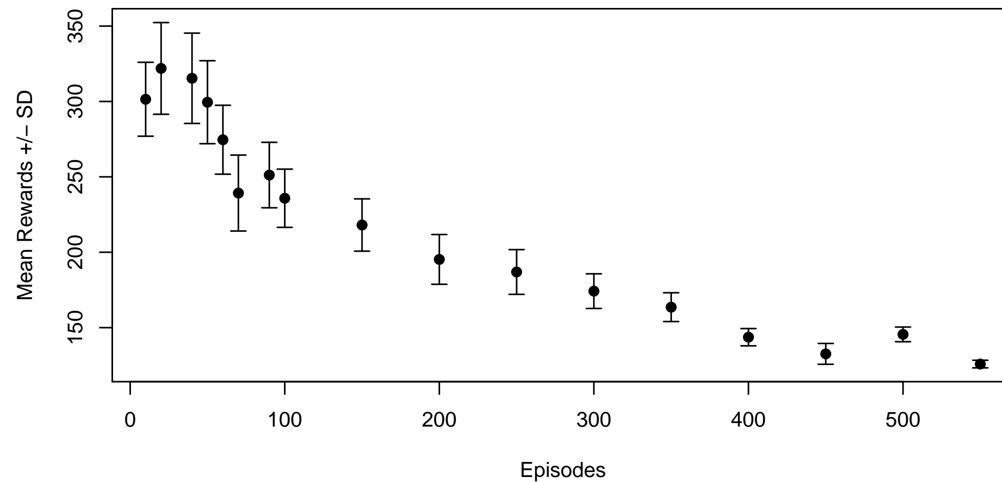
6.3.9 Results

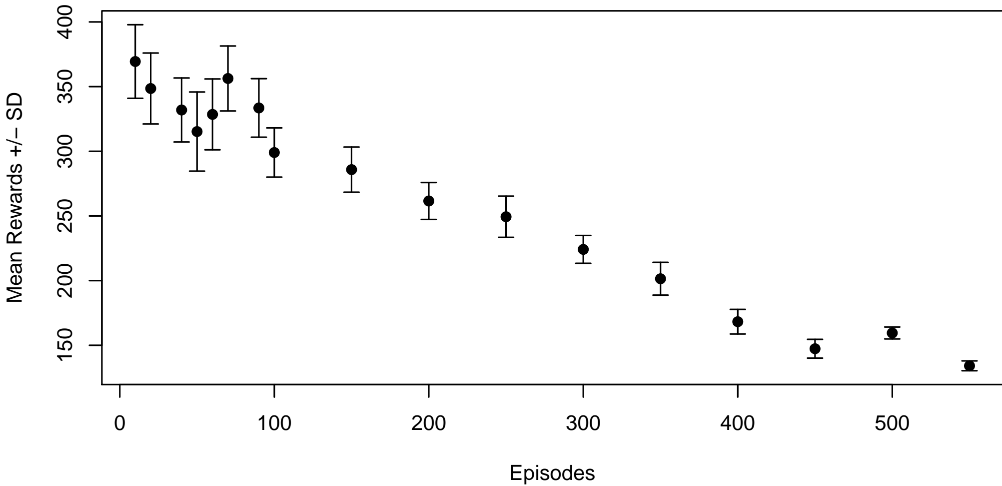
Figure 6.9 describes the average rewards over 550 episodes. It can be noticed that the number of episodes it takes to reach convergence in this experimental setup is more than the previous setup with no environment. We can assume that due to the increase in the state space; it is logical to assume that it would take longer for the learning to converge. As can be seen from the graph, the rewards fluctuate at the beginning of the learning process which is what we expect because the algorithm is in the exploratory stage, trying different actions for the different states before it finally levels out. As the episodes increase, the algorithm starts to select actions more intelligently. Towards the end, we can see that the rewards level out, as the swarm starts to pick more optimal solutions based on previous rewards for the different states.

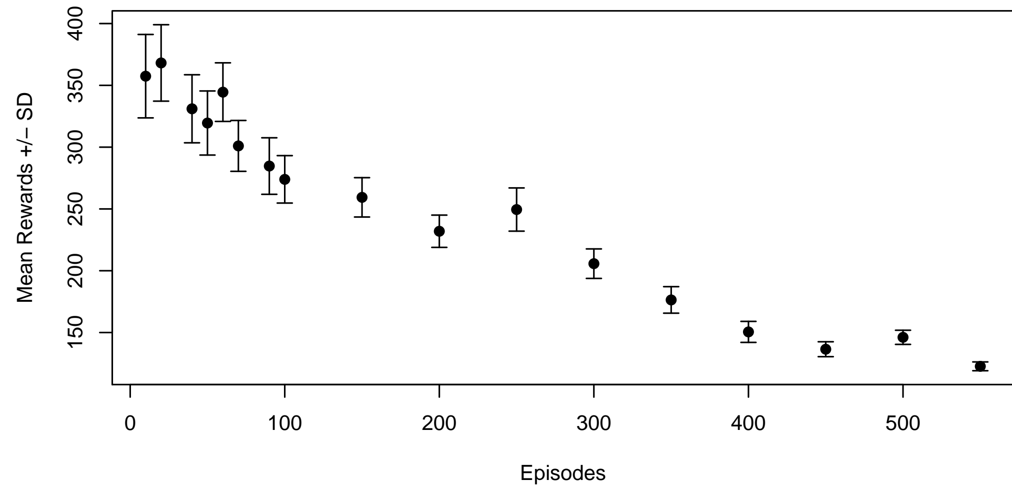

Graph describing mean rewards over episodes (Robot 1)
Mean Rewards +/− SD
Episodes

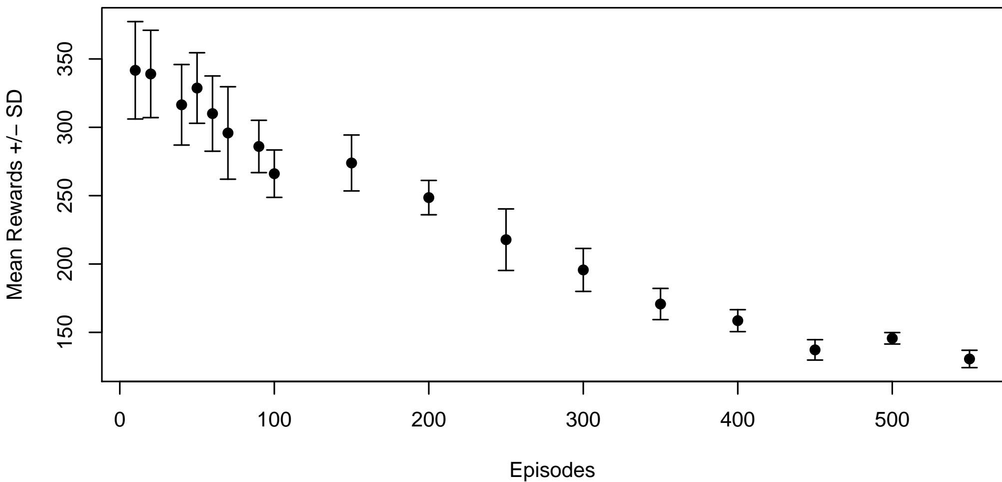

Graph describing mean rewards over episodes (Robot 2)
Mean Rewards +/− SD
Episodes

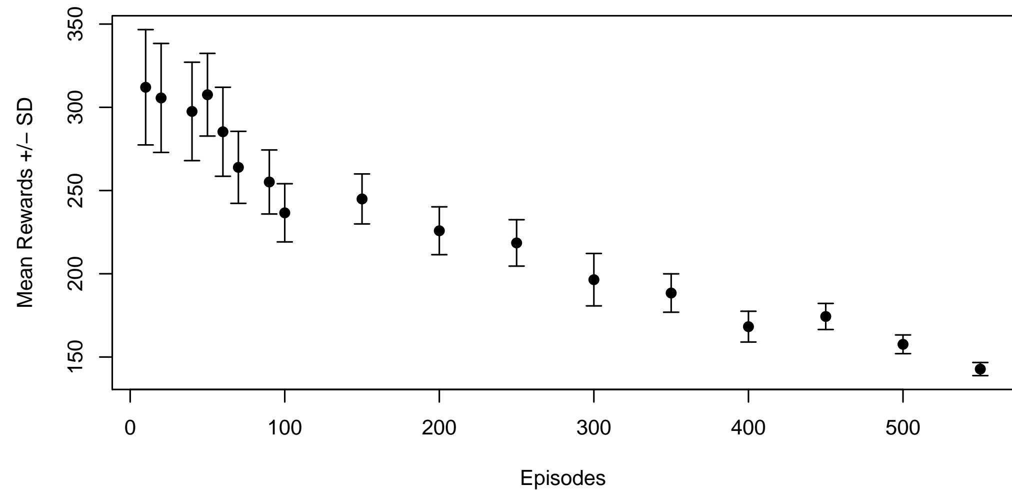

Graph describing mean rewards over episodes (Robot 3)
Mean Rewards +/− SD
Episodes

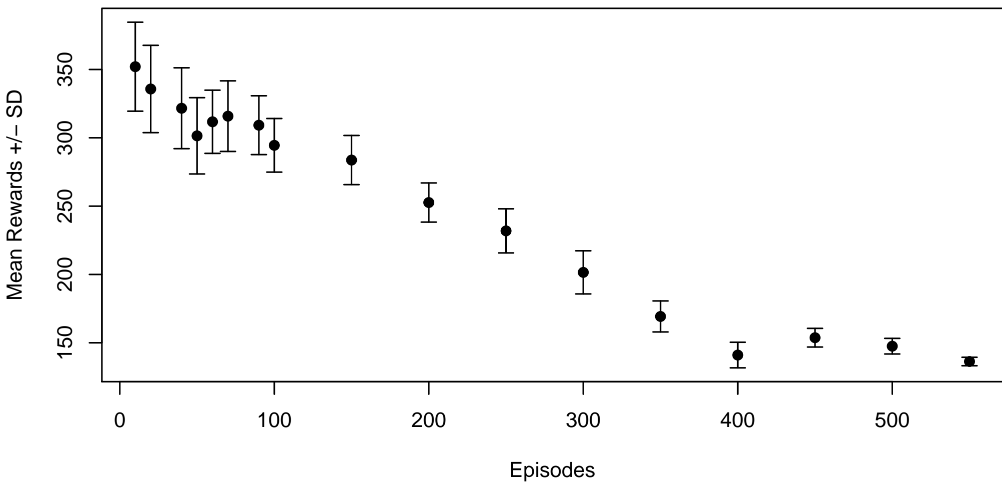

Graph describing mean rewards over episodes (Robot 4)
Mean Rewards +/− SD
Episodes

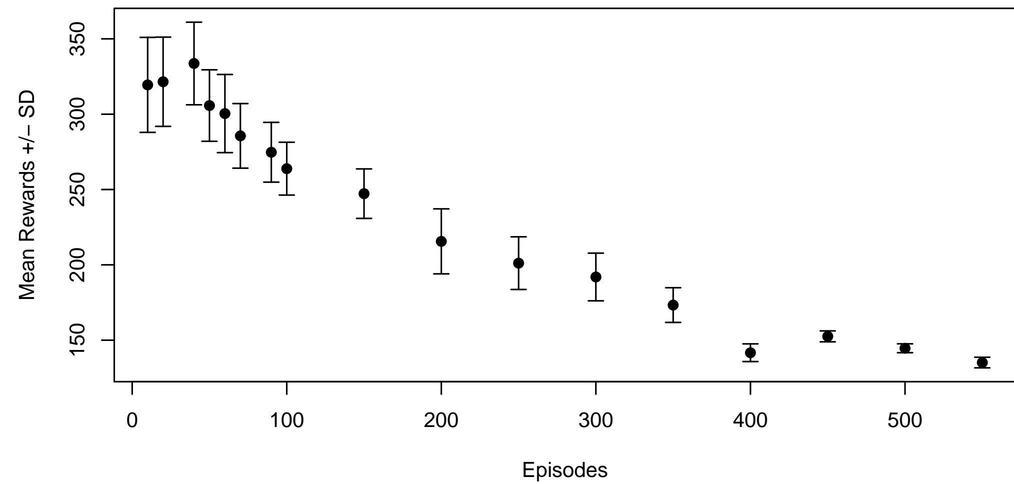

Graph describing mean rewards over episodes (Robot 5)
Mean Rewards +/− SD
Episodes


Graph describing mean rewards over episodes (Robot 6)
Mean Rewards +/− SD
Episodes


Graph describing mean rewards over episodes (Robot 7)
Mean Rewards +/− SD
Episodes


Graph describing mean rewards over episodes (Robot 8)
Mean Rewards +/− SD
Episodes


Graph describing mean rewards over episodes (Robot 9)
Mean Rewards +/− SD
Episodes
150
200
250
300
350
0
100
200
300
400
500


Graph describing mean rewards over episodes (Robot 10)
Mean Rewards +/− SD
Episodes
150
200
250
300
350
0
100
200
300
400
500

Recovery Type Chosen	Average Percentage Chosen
Drag to Base	24%
Fix on the Spot	26%
Drag Along	33%
Leader-Follower	17%

Table 6.5: The table describes the percentage of the test input that is selected during the testing phase of the RL + SOM and the recovery type selected. The recovery type column represents what predefined behaviour was chosen for the 50 sample inputs. The percentage column represents what percentage of the test inputs selects the corresponding recovery type.

Figure 6.9: Rewards Convergence for each Robot

This is tested for different sizes in the swarm (10,20 and 40 robots) to test the scalability of the learnt recovery strategy. It is also tested for three tasks aggregation, foraging and collective phototaxis. The results for the robot swarm sizes, 20 and 40 are in the appendixes.

6.3.9.1 10 Robots in the swarm

Table 6.5 shows that for the 50 test states, 24% of the states chose the strategy to drag to base, 26% chooses to fix on the spot, 33% chooses to drag the faulty robot along and 17% chooses to follow the leader. The reason why the Leader-Follower is chosen less is because it is limited for what faults it can be used on while the other predefined behaviours can be used on all the faults. The swarm picks the best recovery strategy based on the fault type and also the input state. The swarm never leaves a robot behind, they select the best strategy that has been learnt by the swarm. These results are collected across collective phototaxis, aggregation and foraging.

Collective Phototaxis

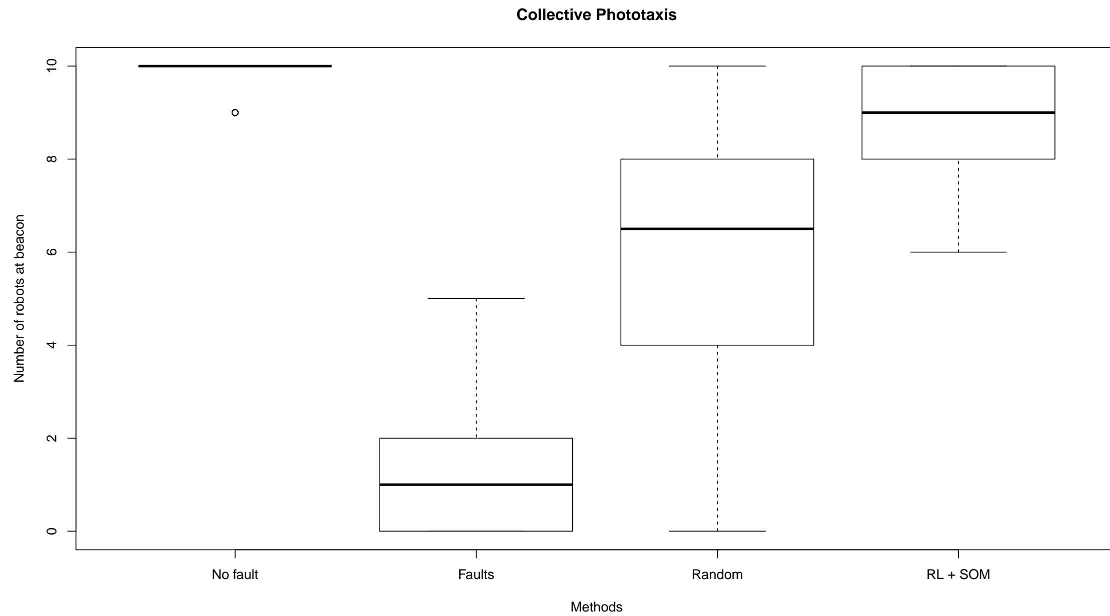


Figure 6.10: Collective Phototaxis: Motor failures

Figure 6.10 describes a box plot showing the number of robots that arrive at the beacon or are at the repair station with obstacles present in the environment. As it can be observed, when there are no faults in the swarm, all of the robots are able to make it to the beacon. When there are faults, the faulty robots anchors the healthy robots as the sensors on the robot are still functioning. We can observe that some robots make it to the light source; they escape the anchoring effect of the faulty robots. When a random behaviour is chosen, the robots make it to the light source or repair station a times while other times, the worst behaviour is chosen which is why we see a range of results. Finally, when the learnt recovery strategy is chosen, more effective strategies are chosen and most of the robots are able to make it to the beacon or repair station.

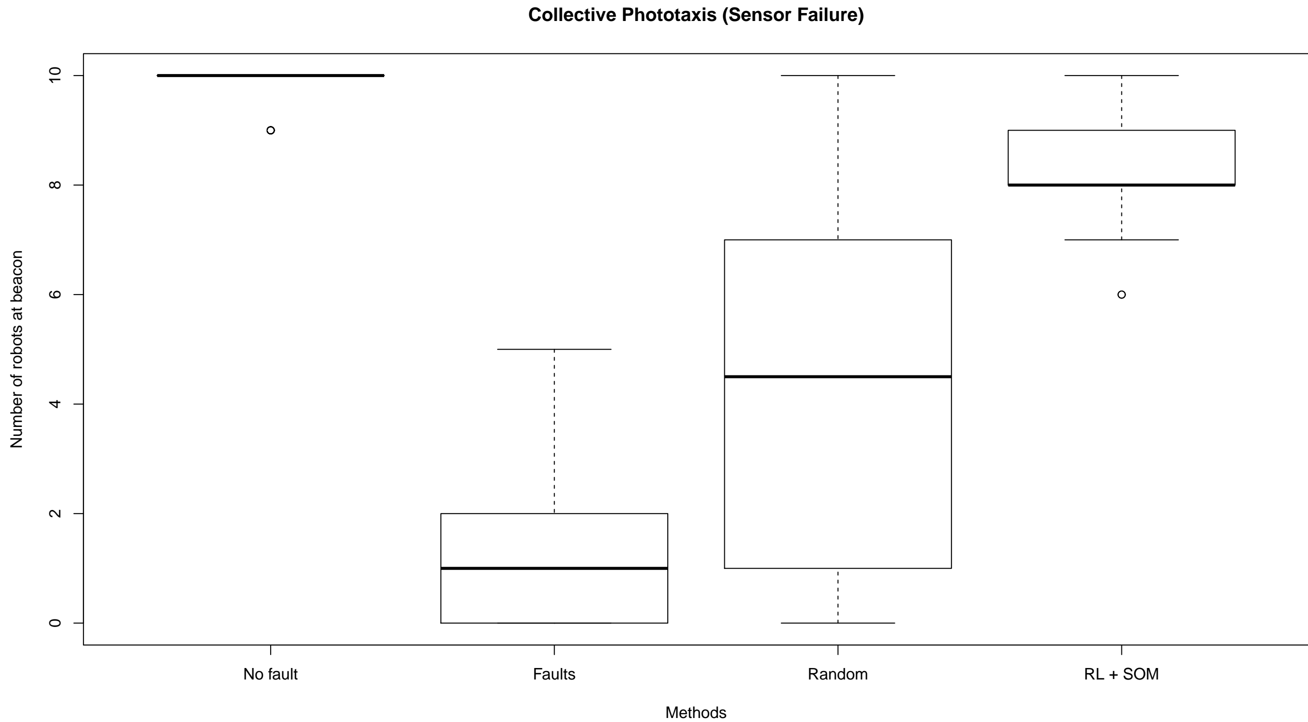


Figure 6.11: Collective Phototaxis: Communication sensor failures

From Figure 6.11, communication sensor failure can be observed; the communication sensors being described are the range and bearing sensors that are used in the robots to locate other robots and the robots use these sensors for cohesion and avoidance. As to be expected, the results show that when no faults are injected, the robots are able to make it to the light source. When the communication fault is injected, the faulty robots move randomly around the environment and the 'healthy' robots are not able to complete task as they anchor and follow around the faulty robots. Choosing a random behaviour shows fault recovery solutions that work sometimes while other times, they do not work. Finally, when the learnt recovery strategy is chosen, more effective strategies are observed to be chosen and majority of the robots are able to complete the task and no robot is left behind.

Aggregation

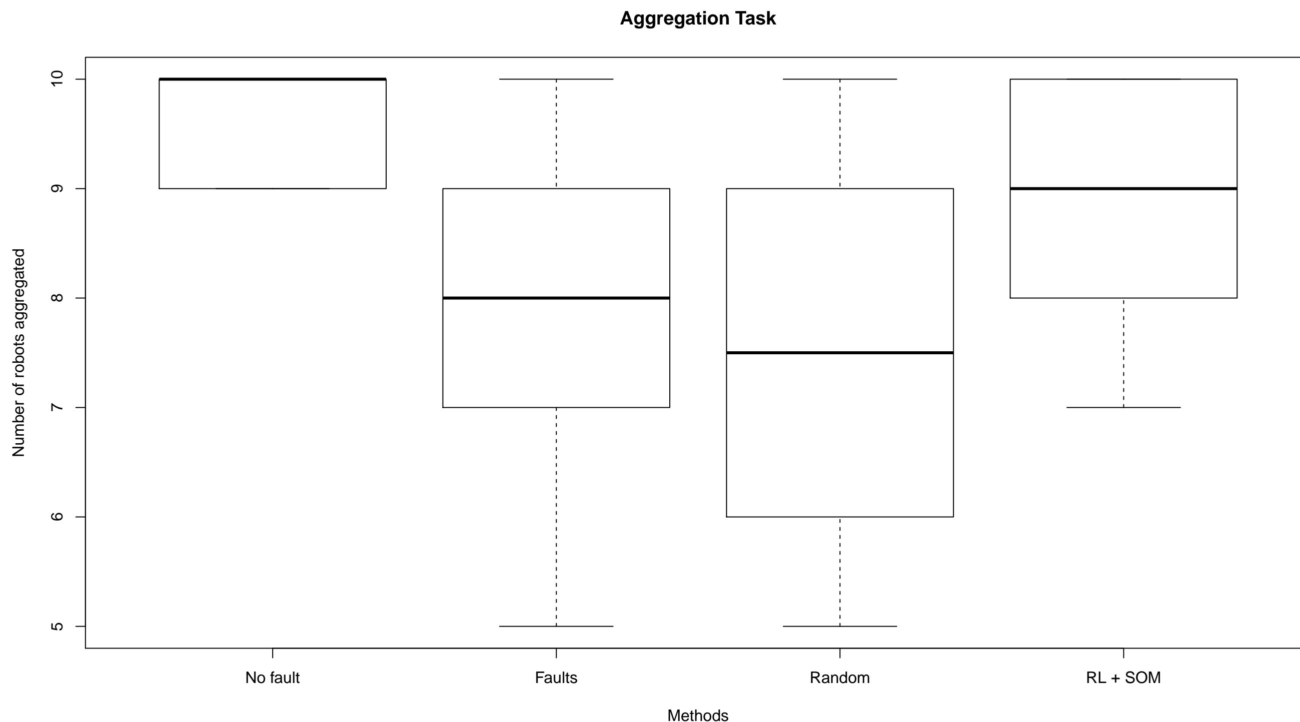


Figure 6.12: Results for aggregation when the range and bearing sensors (Communication sensors) on the robots are faulty.

Figure 6.12 describes the number of robots in aggregates in the environment or at the repair station; depending on the learnt recovery mechanism used. When no faults are present in the system, all robots are able to aggregate together to form one aggregate. As it is discussed in the centralised approach, the discrepancy with the number of robots in an aggregate is due to how compact the robots are when forming the aggregate. The robots are continuously moving even after forming complete aggregates. When faults are present, the robots are not able to make use of their sensors to perform cohesion to create one aggregate; the robots break and form multiple aggregates. This happens because robots move randomly around the environment and the robots become divided between the faulty robots forming smaller collections of the robots. Again, randomly selecting a behaviour shows that it selects a good strategy sometimes but in most cases, it chooses a bad strategy. When the learnt predefined behaviour strategy is implemented as the chosen recovery method, we can see that the swarm recovers successfully which allows the robots form

one cohesive aggregate.

6.3.9.1.1 Foraging

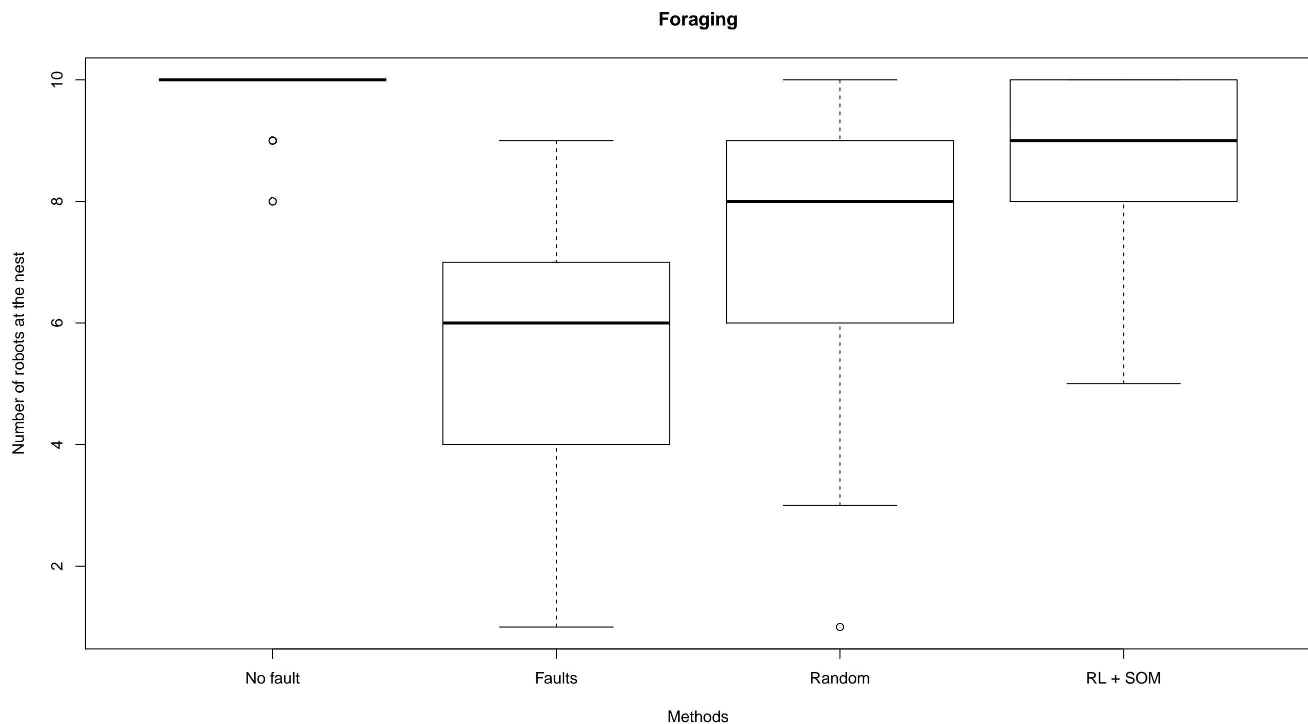


Figure 6.13: Results for foraging when light sensor failures are injected: Number of robots that reach the nest at the end of the task.

Figure 6.13 describes the performance of the foraging task when the light sensor faults are injected. The robots in the swarm, when performing foraging, use lights to determine the location of the nest. Therefore when there is a fault, the faulty robot is unable to sense the light and therefore cannot make it to the nest. The faulty robots wander aimlessly around the environment and reduces the probability that other robots would find and collect the rest of the items. It is observed that between randomly selecting a predefined behaviour and using the learnt recovery strategy, the learnt strategy performs better overall because when randomly selecting a solution, sub-optimal or bad strategies are chosen.

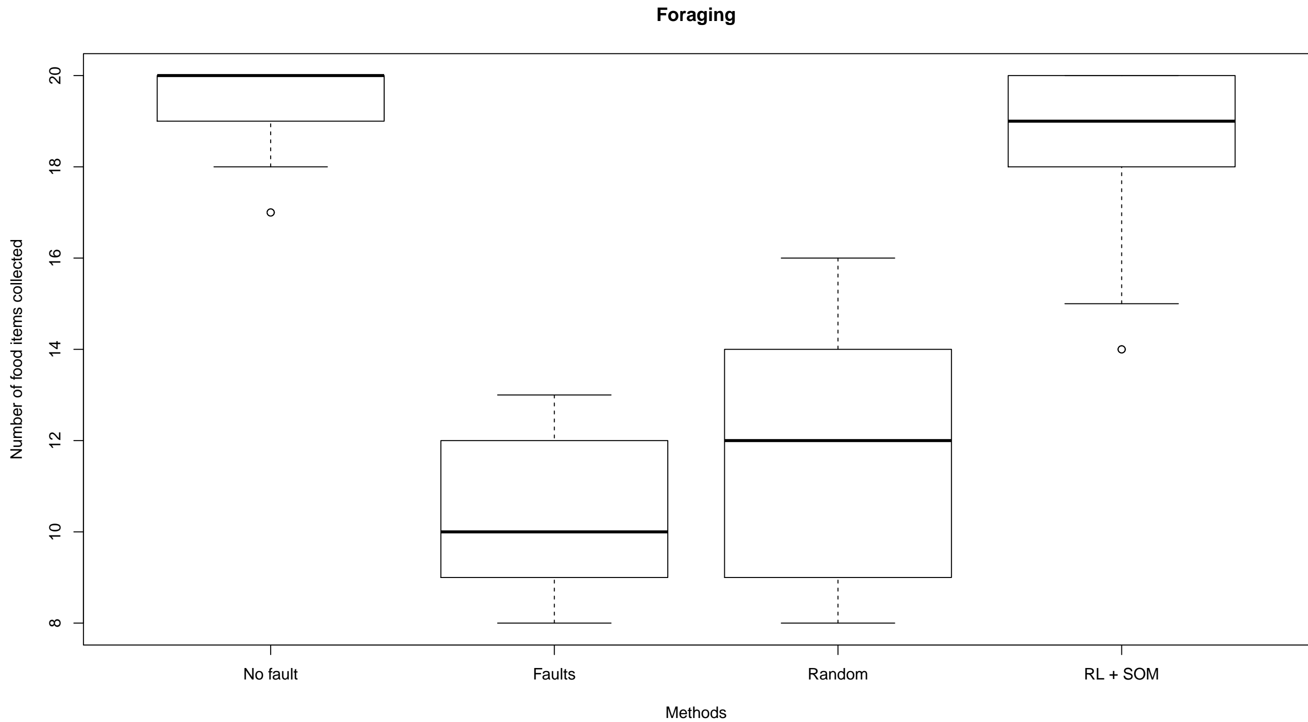


Figure 6.14: Results for foraging when light sensor failures are injected: Number of items collected in the environment

Figure 6.14 shows the results for when amount of 'food items' collected in the environment. In the no-fault scenario, the swarm is able to collect most, if not all of the items in the environment and most of the robots make it back to the nest. When the is fault injected, it can be seen from the results that few 'food items' are collected. When randomly selecting a predefined recovery strategy, we can see that there is a range of results where sometimes some items are picked; the reason is because by randomly selecting a behaviour, sub-optimal or bad strategies are chosen. Finally, when using the learnt recovery strategy solutions, it can be observed that most of the items in the environment are collected because most of the robots are functioning and are able to complete the task as can be observed in Figure 6.13.

In general, we can see that the results are similar to the centralised solution which is what we would expect from these set of experiments. We expect that these experiments would show that learning a fault recovery strategy before a fault occurs rather than randomly select solutions when a fault has been detected and diagnosed.

In this thesis chapter, we have discussed our approach to distributed learning when it is applied to the fault recovery architecture. We have demonstrated with results and graphs how successful the recovery strategy works at selecting the optimal solution for the fault recovery process. We discussed achieving distribution to a certain degree and having a centralised standard consensus where the robots communicate their chosen best recovery strategy with each other. In the future, we would like to have a completely distributed approach where the consensus is also distributed in nature. In both chapters, centralised and distributed, we tested the fault recovery strategy based one type of fault occurring in two robots simultaneously. It is understood that this situation rarely occurs and therefore, the next part would show results for various scenarios by stress testing the present architecture to discover in what conditions does the fault recovery architecture work and why.

Chapter 7

Extension of Experiments

7.1 Introduction

In this section, we extend the experiments that were done in previous sections; in previous sections, each test run was done for two same faults, on two robots at the same time. While it has been observed that in both the centralised and distributed approach the results are positive and the faults are able to recover efficiently and complete the task, it is not realistic to expect that two of the same type of faults would occur at the same time on multiple robots. In the real world, even if the same type of fault occurs in multiple robots, it is not expected for them to occur at the same time. Therefore, in this section, we explore different combinations of faults occurring within a test run and also increasing the number of faults that occurs within a test run. Effectively, we stress-test the system to observe the behaviour of the algorithm when it is pushed to recover from some 'stressed' scenarios. It should be noted that we do not expect the algorithm to work in all of these scenarios.
The reasoning behind stress-testing the system is that robots are made of movable parts which are liable to faults, sometimes multiple faults during a task run. Sometimes, fault(s) could occur at the same time or they could at different times during the task run and the learnt recovery strategy has to be tested to observe its performance in various fault scenarios. In this chapter, we test for different faults that occur randomly which conceptually could occur at the same time or different times. For this, we expect the learnt recovery strategy to cope very well with few faults but as the number of faults injected within a task run increases, there is a degradation in the recovery process. This is due to one main reason:
As the number of faults increases, more robots are actively involved in the recovery process. In the beginning of the recovery process, when there are fewer faults, fewer

robots are involved in the recovery process, therefore, overall less power is being used to 'recover' the robots from their faults. However when more robots are involved in the recovery process (when there are more faults), the 'cost' of the repair becomes more expensive as more power is being used and therefore more robots run out of power faster. There are more robots with low power and if this is what is available during a fault recovery process, the robot breaks down alongside the already faulty robot. Another fault recovery process would have to be done with probably another set of 'low power' robot(s) and the cycle continues. It should reminded that when robots fail, other 'healthy' robots anchor around the robots preventing the swarm from completing the task and this is what occurs. Random selection of predefined behaviour for the recovery process does not cope well with these sort of scenarios whereas the learnt recovery strategy copes better as it has learnt various possible states but as can be seen in the results later on in this chapter, this scenario still affects the learnt recovery strategy.
We have defined the possible faults (that can have varying effect on the swarm) that can occur in a swarm system as:

- Complete Motor Failure
- Communication (Sensor) Failure
- Power Failure

As discussed in earlier chapters, swarm robotic systems are inherently fault tolerant and because of this, they are able to 'tolerate' some faults without adversely affecting the performance of the swarm and the swarm is able to continue with the task. However, depending on the task, there are some faults that have a negative impact on the ability for the swarm to complete their tasks which is not ideal. In an ideal situation, the swarm should able to complete their task when faults occur whether it is a crippling fault or not. When running tests and testing the learnt recovery strategy, i did not deem it necessary to test for faults that do not have an adverse effect on the swarm as the swarm completes the task regardless. Rather, the testing is done for faults that has an adverse effect on the swarm and this is used for all task scenarios in this chapter. We are not saying that the learnt recovery strategy would not work in fault scenarios that do not have a negative effect on the swarm, rather, this thesis is reviewing faults that have the negative effect on the swarm because we can clearly see how well the learnt recovery strategy handles the faults that have occurred.

7.2 Experimental Setup

All experiments in this chapter are set up similarly with a swarm of 10 robots where the faults are injected in the swarm during the swarm. The learning architecture is tested on 50 randomly generated scenarios where each scenario is run with 100 different random seeds and the average performance is reported. Each test run's duration is 1300 seconds. We consider four different treatments that describes the steps taken by the swarm when getting results for the testing phase and is described in the list below. This is done for three possible failure modes: motor failure, communication failure, light sensor failure.

Additionally, the faults are injected at different times within the task and this is done randomly between 100 and 800 seconds; therefore not right at the beginning of the simulation and not right at the end of the task run. This is tested in the collective phototaxis, aggregation and foraging tasks using both the centralised and distributed learning architectures.

The following list describes the steps taken by the swarm from the beginning of the task to the end during the testing phases.

- Run experiments for collective phototaxis and foraging with no faults injected to give baseline performance
- Then test performance with faults injected with faults; this is enough to break behaviour.
- Test the performance random selection of actions. This might help to recover swarm but could be suboptimal.
- Then test performance on the SOM and RL solution.

Depending on the learning approach, centralised or distributed, the algorithm to test the self-organising map and reinforcement learning algorithm would differ as already discussed in previous chapters. Algorithm 9 describes the centralised approach while algorithm 10 describes the distributed approach.

As explained earlier, all experiments in this chapter are setup the same, experimentally.

7.3 Experiment Overview

The scenarios presented in this chapter have similar results as to what was expected and already explained in the introduction. In this section, we give a brief overview

Algorithm 9 Testing process

1: **procedure** FOR TESTING
2: Fault is detected and diagnosed
3: Input Vector State is observed by global observer, $X(S_t)$
4: Unit of SOM with smallest distance from input vector state is winner unit
5: Winner unit is identified in Q-table; action with smallest Q-value is selected
6: Action with smallest Q-value is selected; best learnt recovery strategy selected for that particular state

Algorithm 10 Testing process

1: **procedure** FOR TESTING
2: Fault is detected and diagnosed
3: Input Vector State is collected (Information is transmitted and received from surrounding robots), $X(S_t)$
4: Unit of SOM with smallest distance from input vector state is winner unit
5: Winner unit is identified in Q-table; action with smallest Q-value is selected
6: Action with smallest Q-value is selected; best learnt recovery strategy selected for that particular state
7: Broadcast solution to other robots; receive other solutions from other robots
8: Based on other solutions, select action with highest number of votes and implement

of what to expect in the results in the experiment.
As explained in the introduction, not all faults have a negative impact on the swarm behaviour as swarms are inherently fault tolerant, therefore, the tests done in this chapter are for specific faults that have an adverse effect on swarm behaviour preventing the swarm from completing the task assigned to them. For collective phototaxis, we observe complete motor failures and sensor failures and run our tests on these faults. For aggregation, we test on sensor failures as this is the only fault that shows such an adverse effect on the performance of the swarm. When observing complete motor or power failures in the robots for aggregation, the robots simply aggregate around the faulty robot whereas for sensor failures, the faulty robots escape from the pull of the swarm and wander haphazardly around the environment with the other 'healthy' robots following. This causes, fewer robots to no robot in an aggregate to be formed as the robots are scattered around the environment. It can be said that they 'anchor' around the wandering faulty robot. For foraging, when any fault occurs in the swarm, the faulty robot(s) gets lost in the environment and is

not able to participate in the foraging task (searching and collecting food items). In this foraging, the robots do not need to perform cohesion and they avoid each other and obstacles in the environment to avoid self-damage. Due to this, it has been observed because there are no anchoring effects rather faulty robot(s) getting lost in the environment when complete motor failure, power failure, sensor failure or light sensor. This reduces the number of robots that are able to explore the environment, collect food items and return back to base and the healthy robots are pushed more than normal to collect the food items and are usually not able to pick all the food items during the task run; collecting food items and returning them to the base is the goal of foraging.
In the table below, we detail the various scenarios and the results that can be observed from the plots that are shown in next few sections:

Learning Approach	Task	Scenarios	Result Overview
Centralised	Collective Phototaxis	Same Fault at Different Times	As seen in figure 7.1 and figure 7.2; the results for both motor and sensor are similar to the results seen in previous centralised approach chapter. This is because we are dealing with the same number of faults but they are occurring at random times within the task run which is what is expected.
		Different Faults (2) at Different Times	In the scenario shown in figure 7.3 which describes when two different faults (motor and sensor failures) are injected, it is still expected that the learnt recovery strategy would perform better than the random strategy which is what can be observed from the plots shown above. The strategy performs similar as the same two faults as, again, we are still dealing with two faults during the task run.
		Multiple Faults (4, 6, 8) at Different Times	In figure 7.4, it can be observed that as more faults are added, the performance of both the random and learnt recovery strategy deteriorates which is to be expected although the learnt recovery strategy still performs better than the random strategy.
	Aggregation	Same Fault at Different Times	As can be seen in figure 7.5, It can be observed that the number of robots in the aggregates, when compared to the results in the centralised approach chapter, is very similar which is expected as we are dealing with the same number of faults.
		Multiple Faults (4, 6, 8) at Different Times	Figure 7.6 describes the highest number of robots in an aggregate in the swarm at the end of the task run for when 4, 6 and 8 sensor faults within a task run. As seen in the collective phototaxis section,we see a degradation of the learnt recovery strategy's performance although it performs better than random selection.

	Foraging	Same Fault at Different Times	For Figure 7.7 and figure 7.8, It can be observed that the performance of the learnt recovery strategy performs better than random selection but performs similarly when the faults are inserted at the same time which is what is expected when compared to the results in the centralised chapter. Most of the robots are able to make it back to base while majority if not all of the food items scattered around the environment is collected which is what is we want and classify as a successful foraging exercise.
		Multiple Faults (4, 6, 8) at Different Times	Figure 7.9 and figure 7.10 describes the results for when multiple faults (4, 6 and 8 light sensor faults) are inserted into the swarm. It has been discussed in previous chapters, the importance of the light sensors in foraging and as expected, there is a deterioration in random selection and also the learnt recovery strategy. However, the learnt recovery recovery strategy performs better in general when compared to randomly selecting a predefined strategy. A good number of robots make it back to the base and a good number of food items are collected.
Distributed	Collective Phototaxis	Same Fault at Different Times	As can be seen in figure 7.11 which describes complete motor failures in the faulty robots, the performance of the fault recovery strategy when compared to the performance in the distributed approach chapter, the results are very similar and this is expected as two faults are still being dealt with even though they are being inserted at different times which is what is expected. In figure 7.12, we observe the collective phototaxis task where complete sensor faults are inserted at different times during the task run. Again, when compared to the distributed approach results, the results are similar, including the random and the learnt fault recovery strategy which is what is expected as the same number of faults are being injected in this scenario as that described in the distributed learning chapter.
		Different Faults (2) at Different Times	Although two different faults are being inserted, the number of faults are still the same and it is expected that the learnt recovery strategy should perform better than random selection, which it does and there is no significant deviation from the performance in the learnt recovery strategy when compared to injecting two similar faults in the swarm at different times. The plot can be seen in figure 7.13

		Multiple Faults (4, 6, 8) at Different Times	The plots shown in figure 7.14 shows when of 4, 6 and 8 faults are inserted into the swarm and the adverse effect it has. In the faults column, it can be observed that no robot is able to make it to the beacon; this can be classified as a task fail. It can be observed that as more faults are added, the performance of both the random and learnt recovery strategy deteriorates which is to be expected although the learnt recovery strategy still performs better than the random strategy.
	Aggregation	Same Fault at Different Times	Figure 7.15 describes the highest number of robots in an aggregate in the swarm at the end of the task run for this scenario. It can be observed that the number of robots in the aggregates, when compared to the centralised approach, is very similar which is to be expected as the same number of faults is still being dealt with.
		Multiple Faults (4, 6, 8) at Different Times	Figure 7.16 describes the highest number of robots in an aggregate in the swarm at the end of the task run for when 4, 6 and 8 sensor faults within a task run. As seen in the collective phototaxis section,we see a degradation of the learnt recovery strategy's performance although it performs better than random selection. These faults are inserted into the swarm at different random times all through the task run. These results are what is expected as more faults are added in the swarm.
	Foraging	Same Fault at Different Times	Figure 7.17 and figure 7.18 describes the number of robots at the base at the end of the task run and also the number of food items collected. Two light sensor faults are inserted randomly at different times within the the task run. It can be observed that the performance of the learnt recovery strategy performs better than random selection but performs similarly when the faults are inserted at the same time which is what is expected. The results observed here are similar to the results in the previous distributed approach chapter, which is what is expected as the same number of faults are being inserted into the swarm.
		Multiple Faults (4, 6, 8) at Different Times	Figure 7.19 and figure 7.20 describes the results for when multiple faults (4, 6 and 8 light sensor faults) are inserted into the swarm. It has been discussed in previous chapters, the importance of the light sensors in foraging and as expected, there is a deterioration of the learnt recovery strategy and random selection as the number of faults increases in the swarm during the task. The faults are inserted at random times within the task run (between 100 and 800 seconds).

Table 7.1: This table gives an overview of all the results that are all the experiments that are in this chapter.

The corresponding graphs for the scenarios detailed in the table above, for both the centralised and distributed approaches, are shown in the following sections.

7.4 Centralised Approach

This section describes the results of the tasks when using the learnt recovery strategy that has been implemented in the centralised approach.

7.4.1 Collective Phototaxis

Collective phototaxis involves a swarm of robots moving towards a light source in the environment. The robots make use of the range and bearing sensors for coherence and avoidance between the robots and objects in the environments while they use the light sensor to sense the beacon. The results from the scenarios described in the previous section is shown below:

7.4.1.1 Two same faults at different times

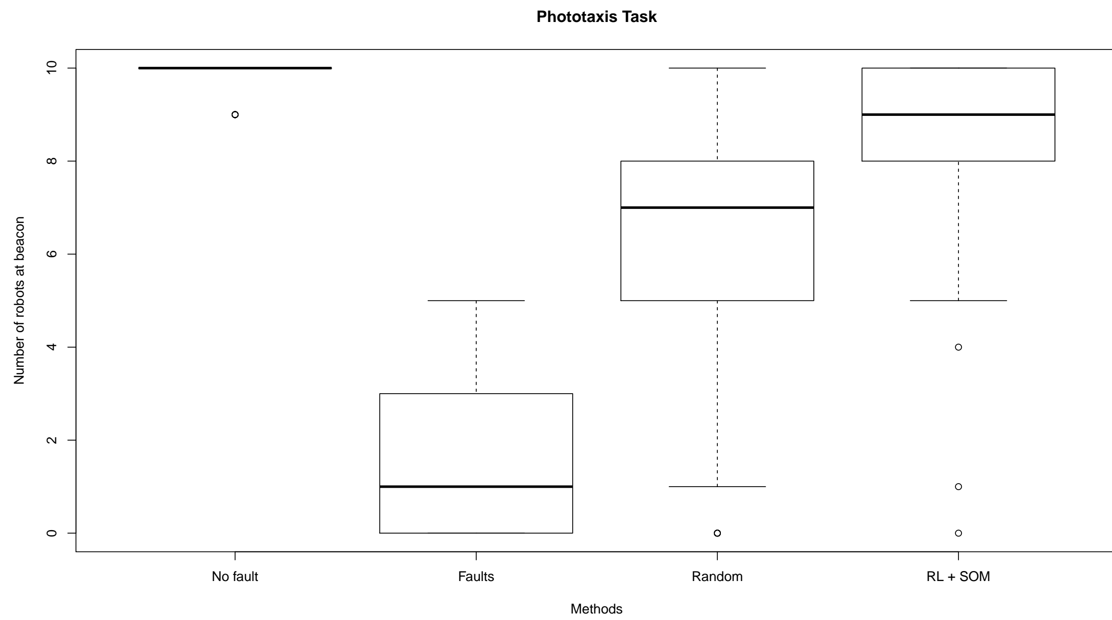


Figure 7.1: Collective Phototaxis: Motor failures

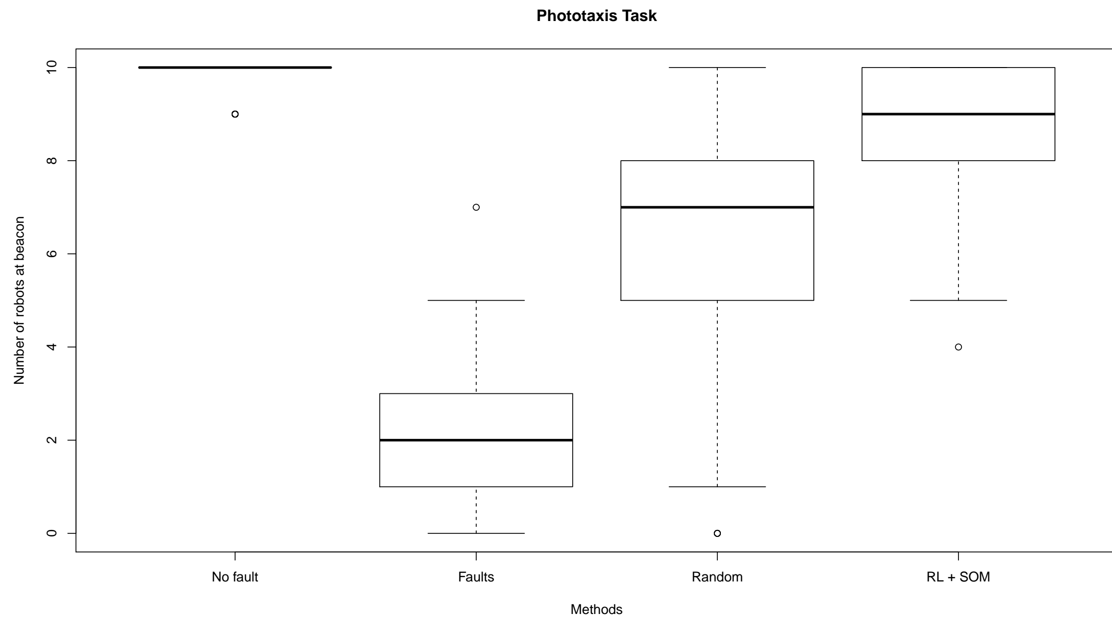


Figure 7.2: Collective Phototaxis: Complete Sensor Failures

7.4.1.2 Two different faults at different times

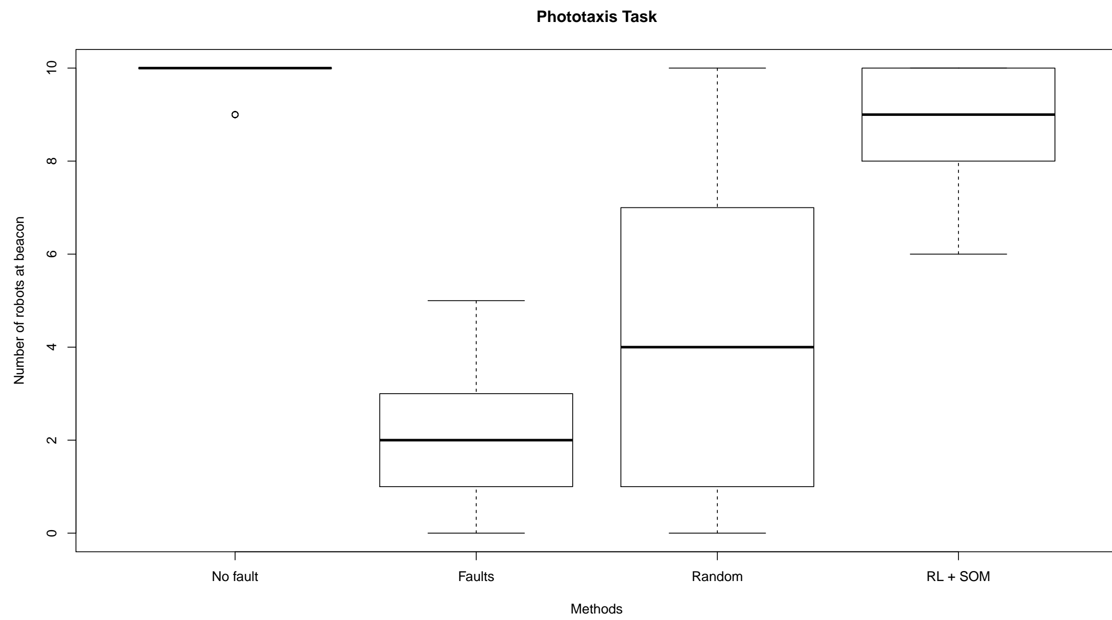
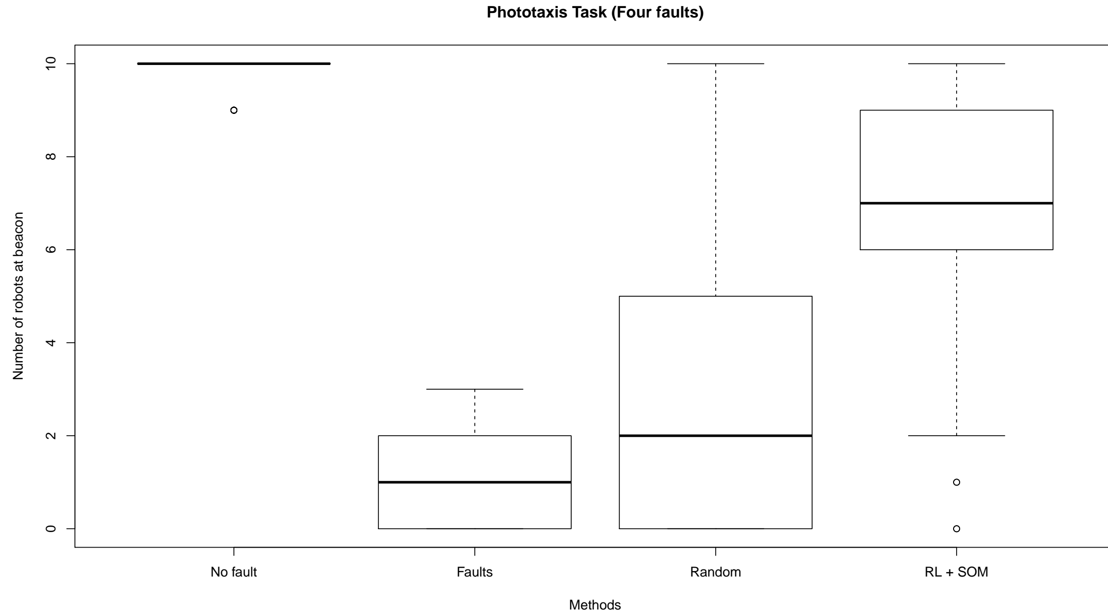
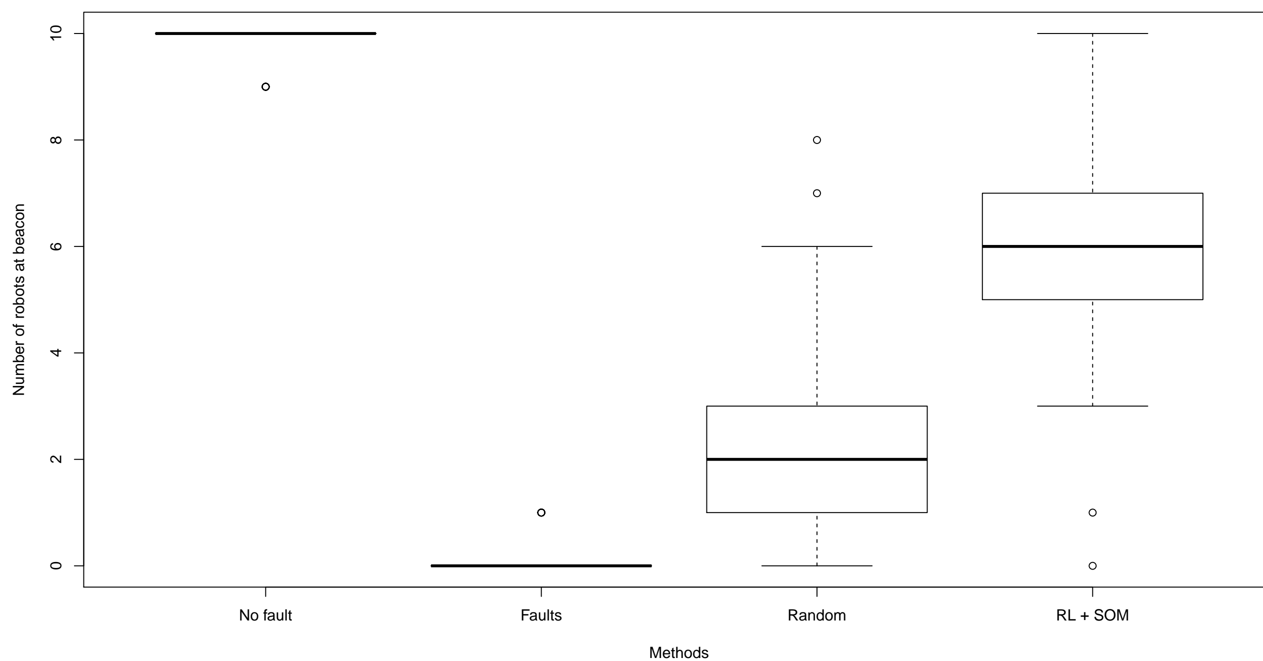


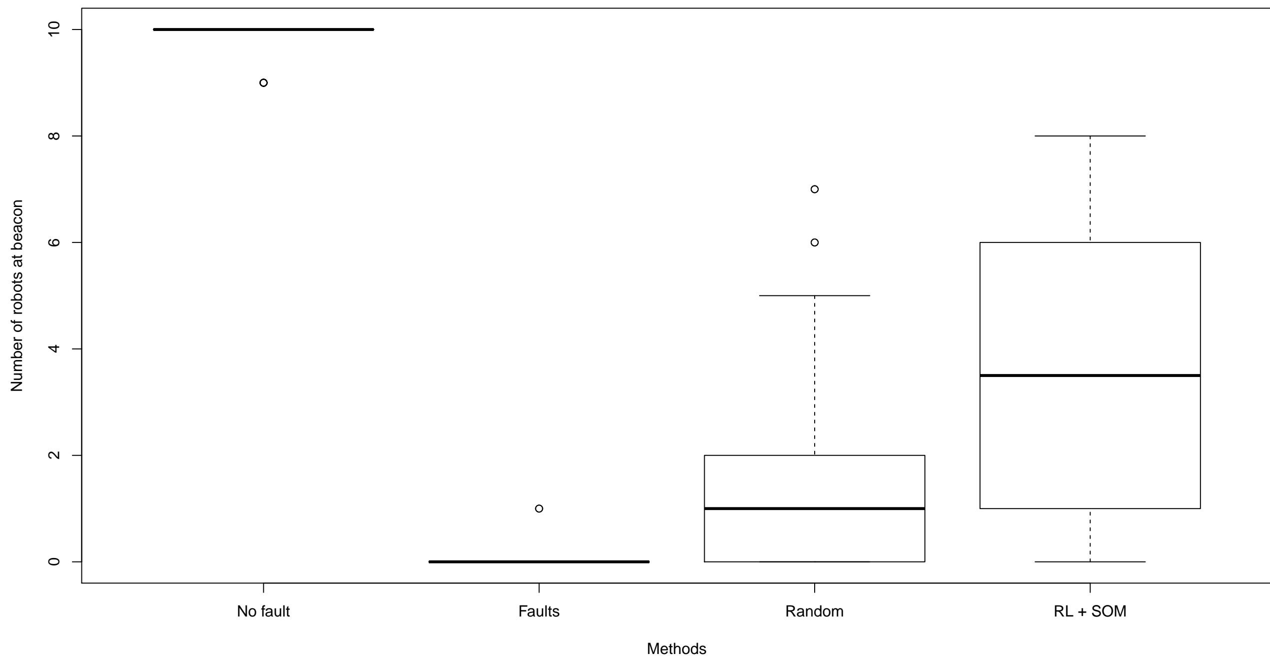
Figure 7.3: Collective Phototaxis: Multiple Failures

7.4.1.3 Multiple combination of faults at different times

Figure 7.4: The plots below describe Collective Phototaxis for Multiple Failures when injected with 4,6 and 8 faults




Phototaxis Task (Six Faults)
Number of robots at beacon
0
2
4
6
8
10
No fault
Faults
Random
RL + SOM
Methods


Phototaxis Task (Eight Faults)
Number of robots at beacon
0
2
4
6
8
10
No fault
Faults
Random
RL + SOM
Methods

7.4.2 Aggregation

Aggregation involves the robots in a swarm coming together from around the environment to form one cohesive aggregate in the environment. The robots in the swarm make use of the range and bearing actuators and sensors onboard to sense the other robots and objects in the environment to move towards each other to form one aggregate. The results from the scenarios described in the previous section is shown below:

7.4.2.1 Fault at different times

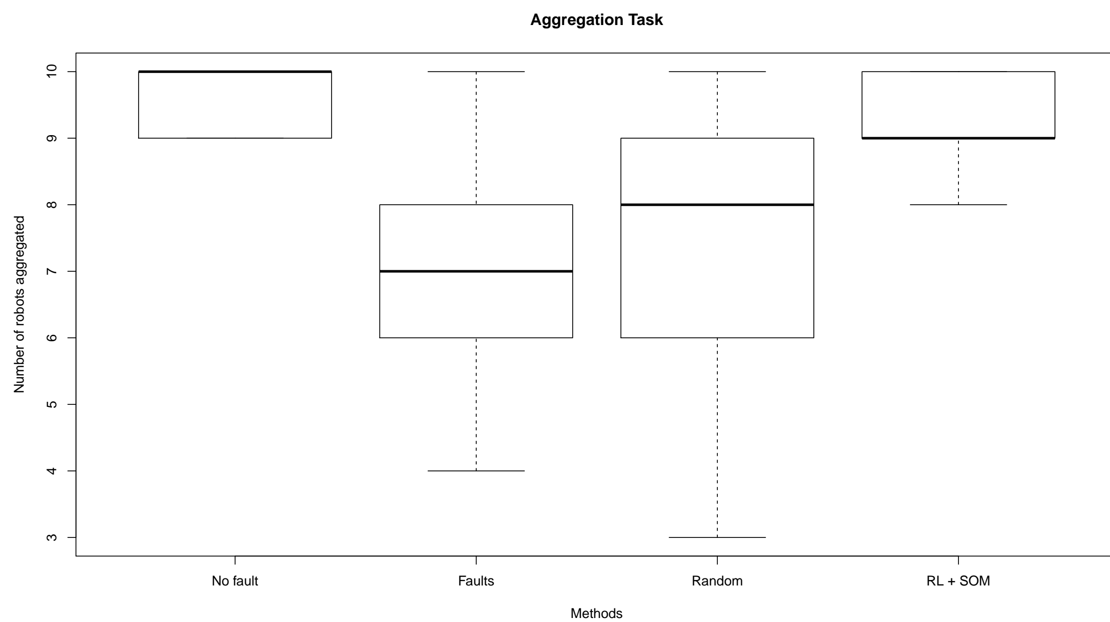
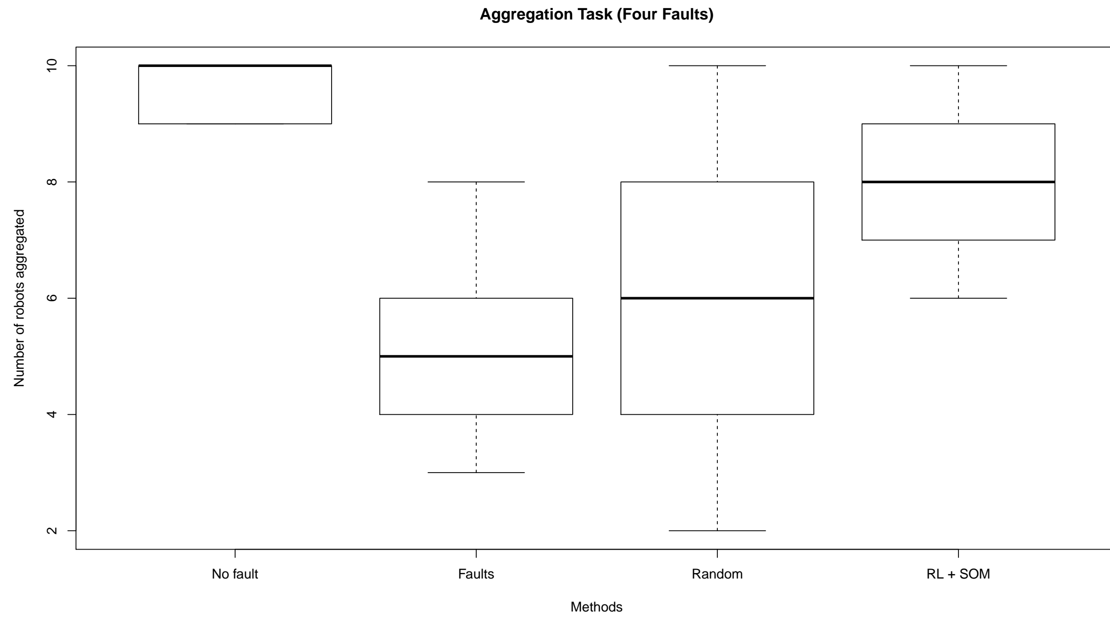
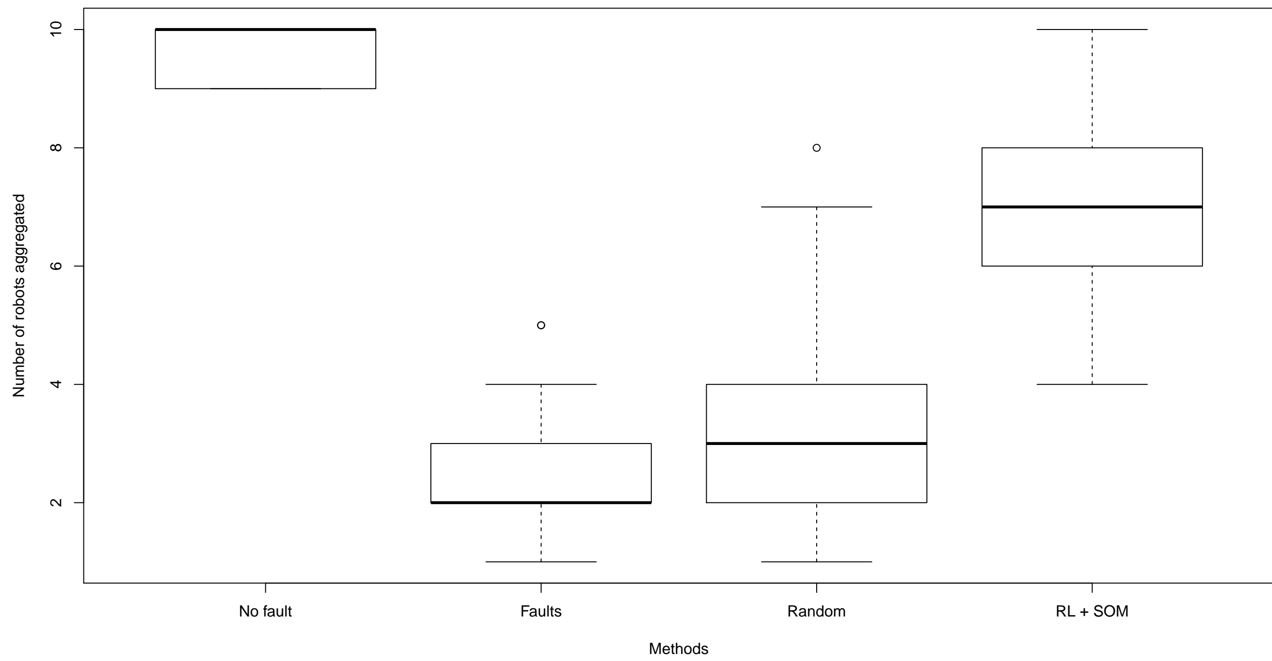


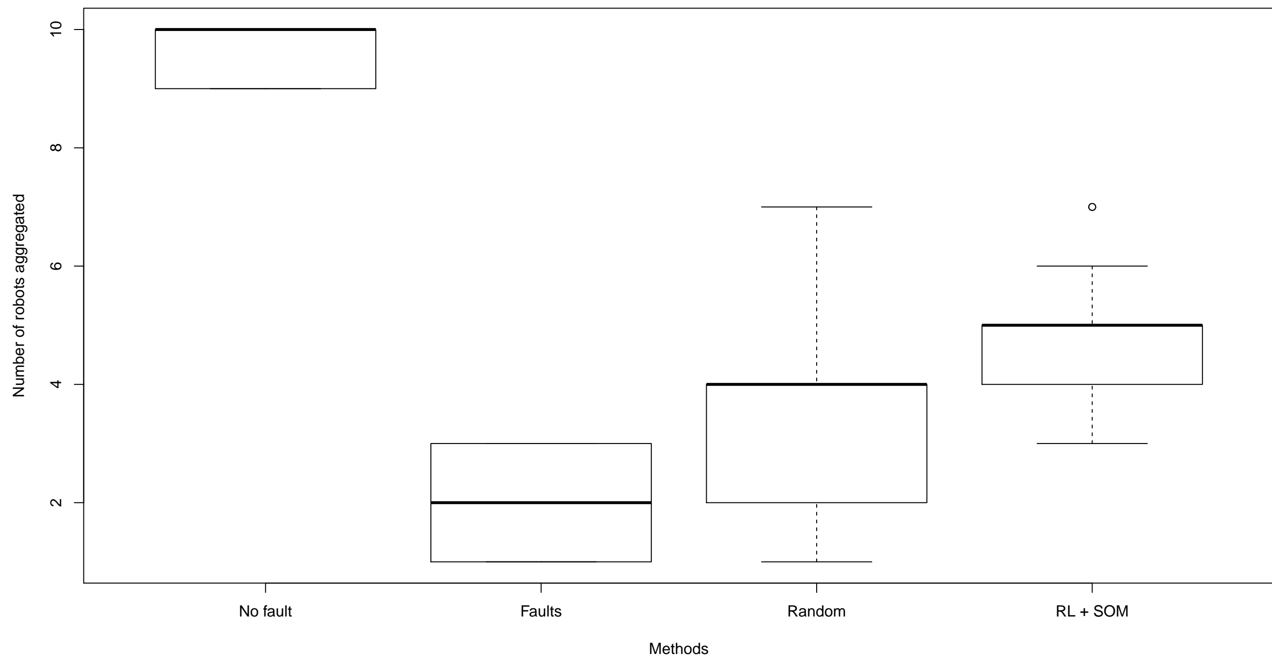
Figure 7.5: Aggregation: Complete Sensor Failures

7.4.2.2 Multiple faults at different times

Figure 7.6: The plots below describe Aggregation for Multiple Failures when injected with 4,6 and 8 faults




Aggregation Task (Six Faults)
Number of robots aggregated
10
8
6
4
2
No fault
Faults
Random
RL + SOM
Methods


Aggregation Task (Eight Faults)
Number of robots aggregated
10
8
6
4
2
No fault
Faults
Random
RL + SOM
Methods

7.4.3 Foraging

Foraging involves robots in the swarm moving from the 'base' to the environment in search for food items. The robots search the environment for food, collect them and bring them back to base. The goal is for the robots in the swarm to search the entire environment and collect all the 'food items' in the items and return back to the base station. There are light sensors present at the base station which allows the robots to return back to the base station safely to drop the food items and continue searching for food items located in the environment.

7.4.3.1 Two faults at different times

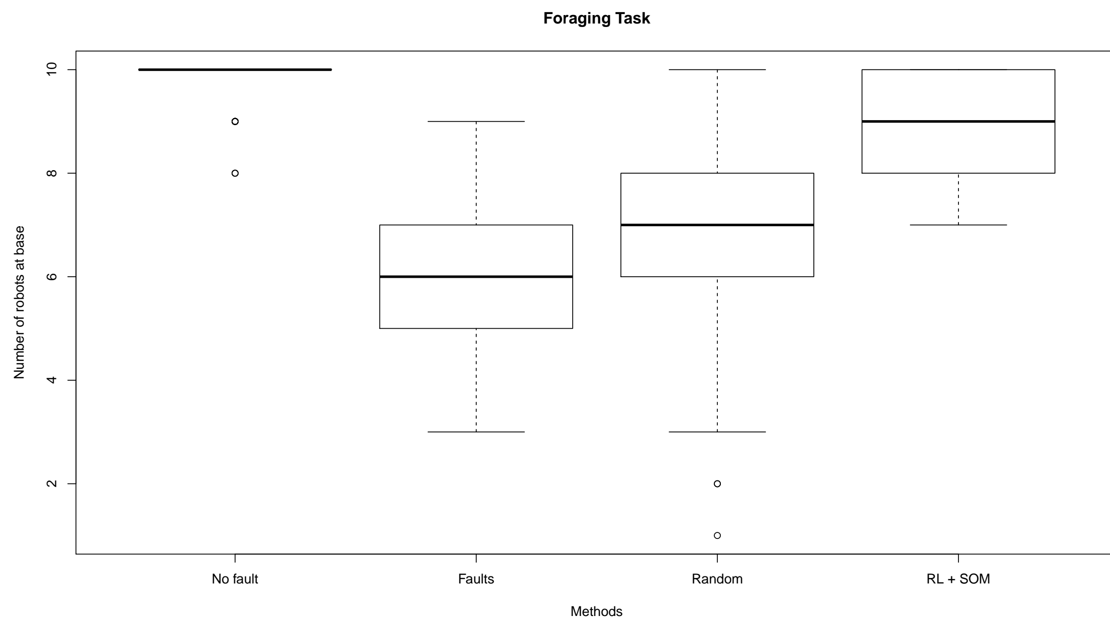


Figure 7.7: Foraging: Light Sensor Failure (Number of Robots at Base)

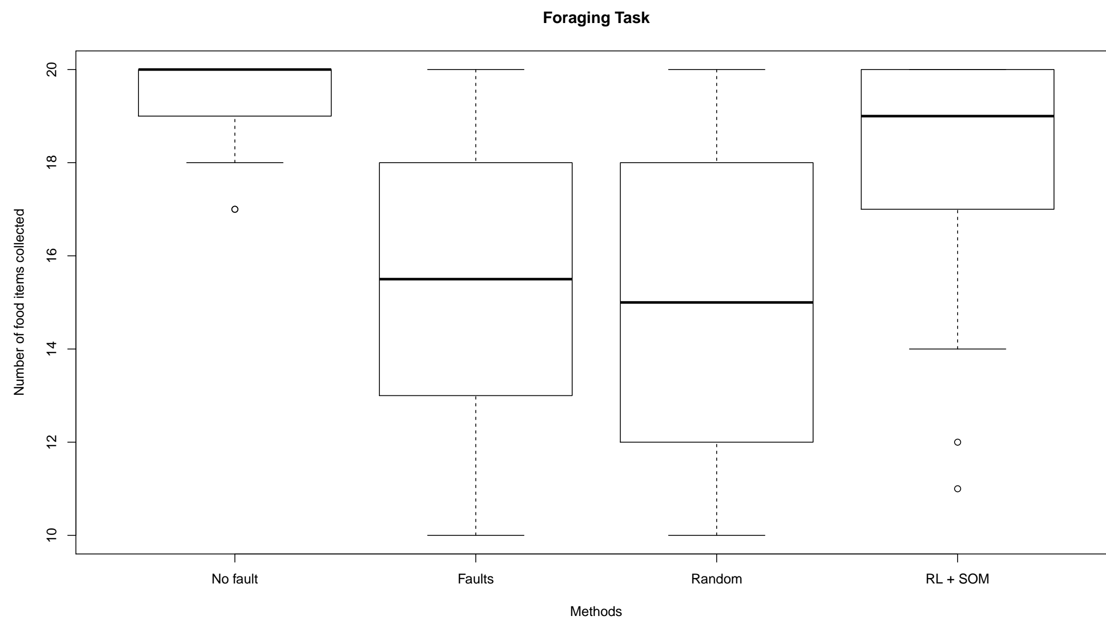
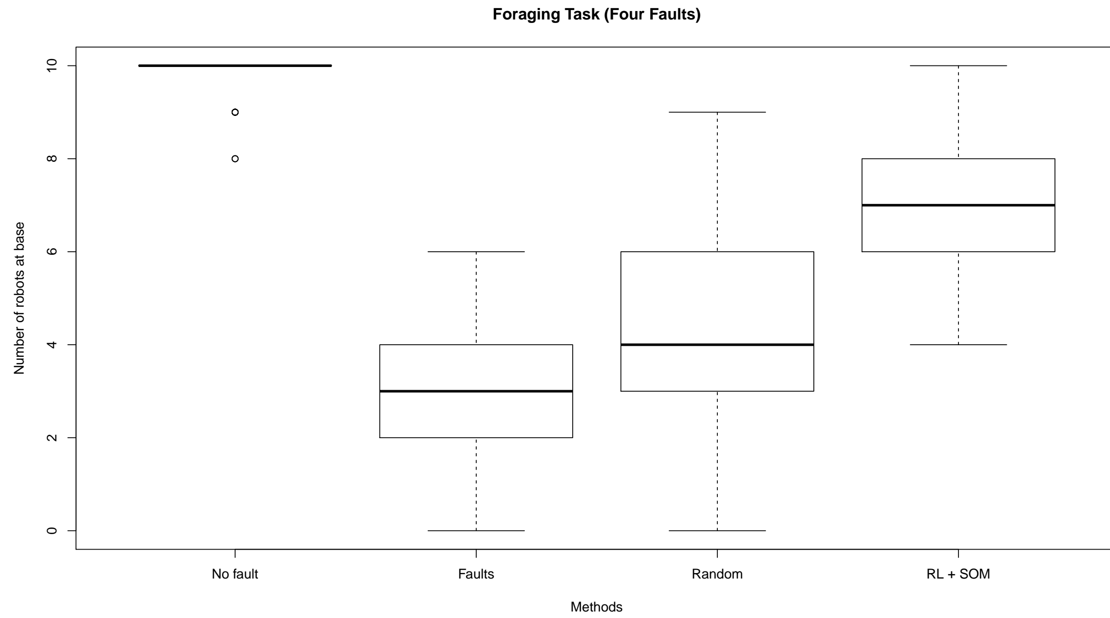
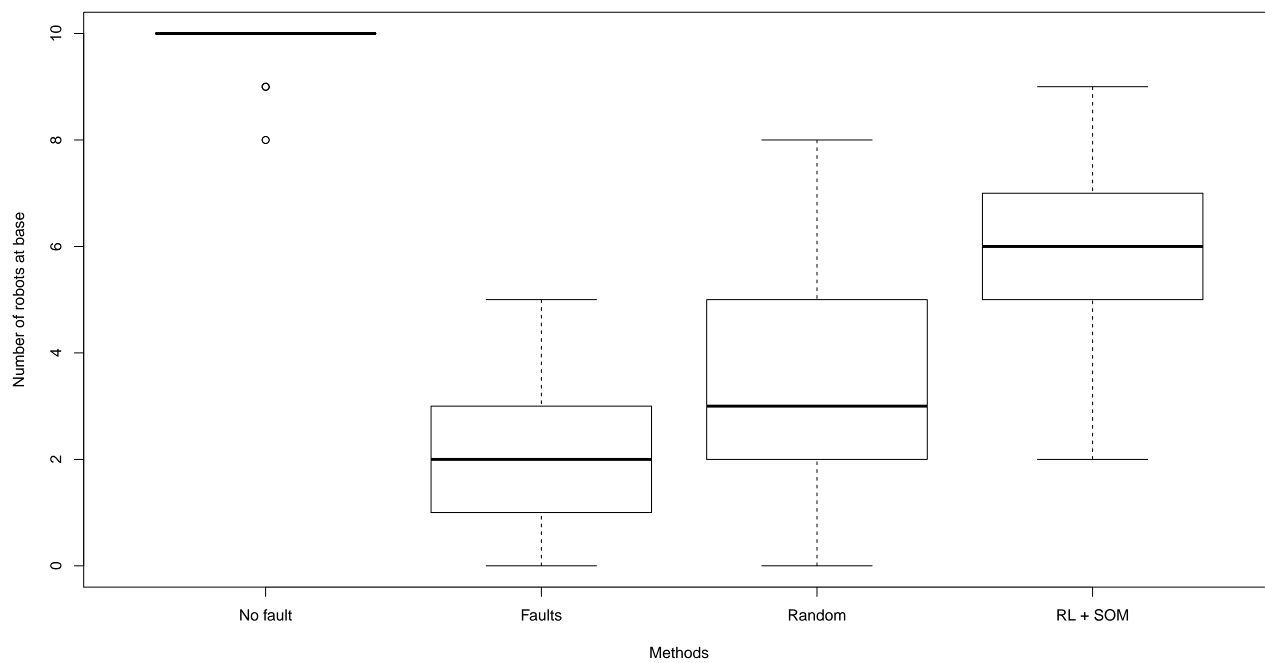


Figure 7.8: Foraging: Light Sensor Failure (Food Items Collected)

7.4.3.2 Multiple faults at different times

Figure 7.9: The plots below describe Foraging for Light Sensor Failure (Number of Robots at Base) when injected with 4,6 and 8 faults




Foraging Task (Six Faults)
Number of robots at base
0
2
4
6
8
10
No fault
Faults
Random
RL + SOM
Methods

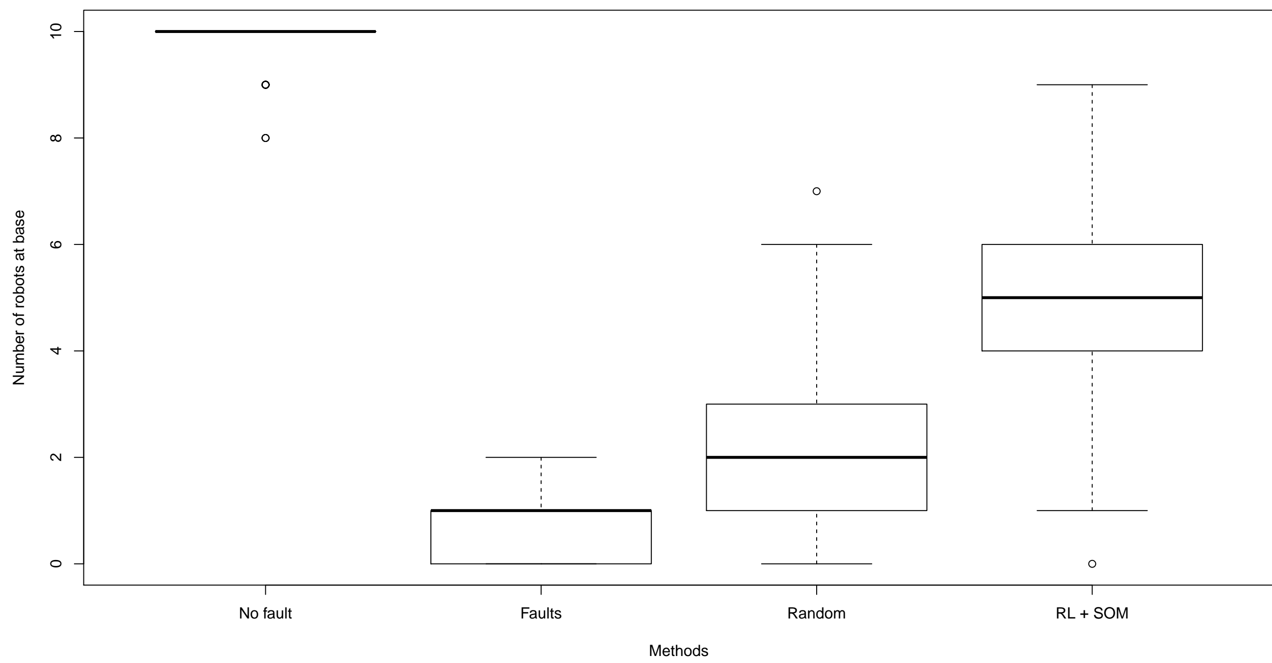
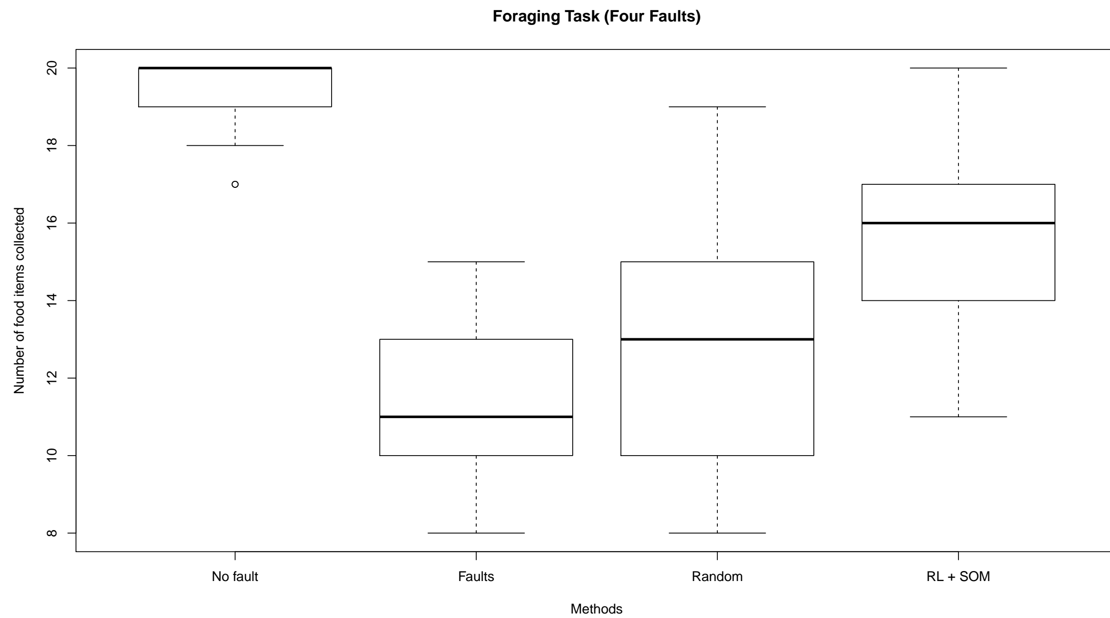
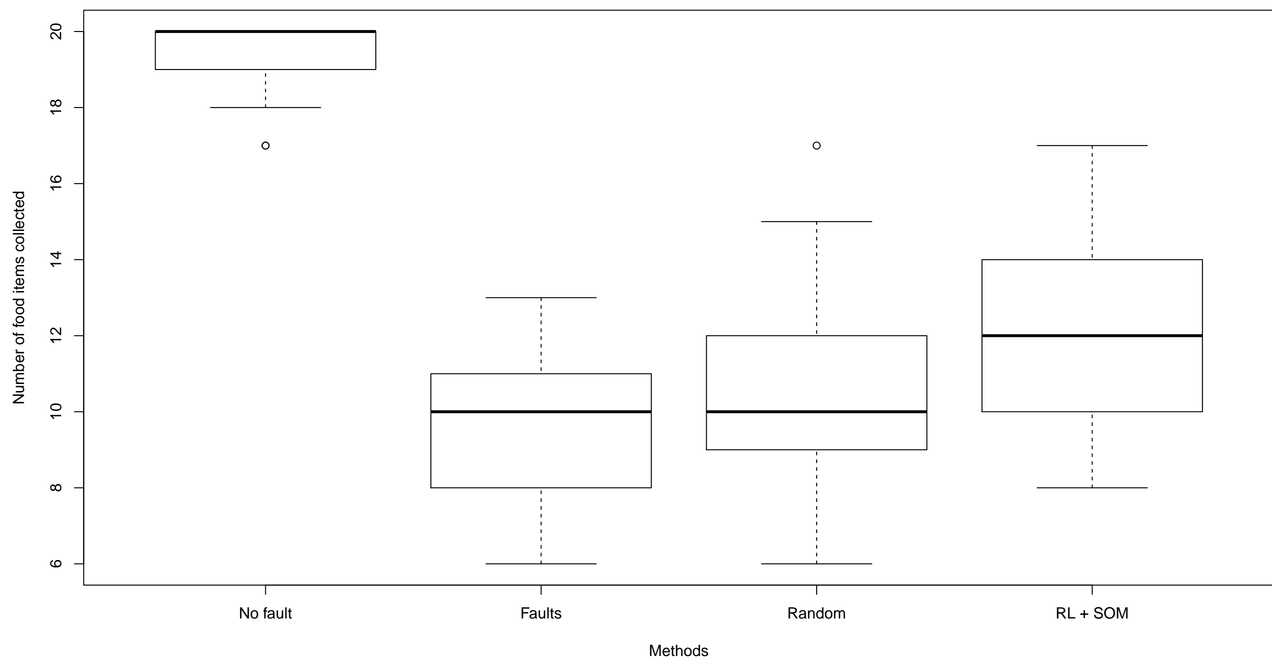
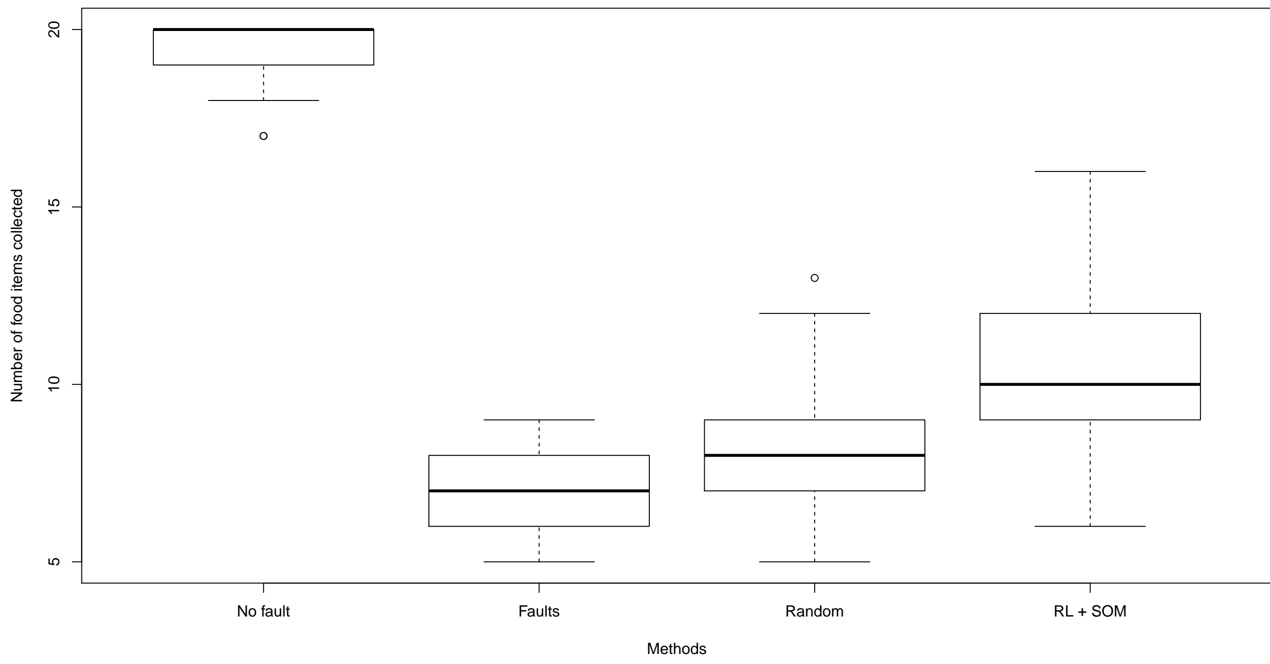

Foraging Task (Eight Faults)
Number of robots at base
0
2
4
6
8
10
No fault
Faults
Random
RL + SOM
Methods

Figure 7.10: The plots below describe Foraging for Light Sensor Failure (Food Items Collected) when injected with 4,6 and 8 faults




Foraging Task (Six Faults)
Number of food items collected
6
8
10
12
14
16
18
20
No fault
Faults
Random
RL + SOM
Methods


Foraging Task (Eight Faults)
Number of food items collected
20
15
10
5
No fault
Faults
Random
RL + SOM
Methods

In this section, we describe the results for various scenarios when using the centralised learnt architecture. The results that are shown are what we expect when these various faults are applied to the swarm. When we have two faults in the swarm, we observe that the learnt strategy copes well with regardless of the type of fault (as seen in collective phototaxis) however, as more faults are added to the swarm, the performance for the learnt recovery strategy falls although it still performs better than randomly selecting a predefined recovery strategy which ideally, is what should be seen. Regardless of the situation, learning your recovery strategy before-hand has a better performance overall when applied to these different tasks.

7.5 Distributed Approach

This section describes the results of the tasks when using the learnt recovery strategy that has been implemented in the distributed approach.

7.5.1 Collective Phototaxis

Collective phototaxis involves a swarm of robots moving towards a light source in the environment. The robots make use of the range and bearing sensors for coherence and avoidance between the robots and objects in the environments while they use the light sensor to sense the beacon. The results from the scenarios described in the previous section is shown below:

7.5.1.1 Two same faults at different times

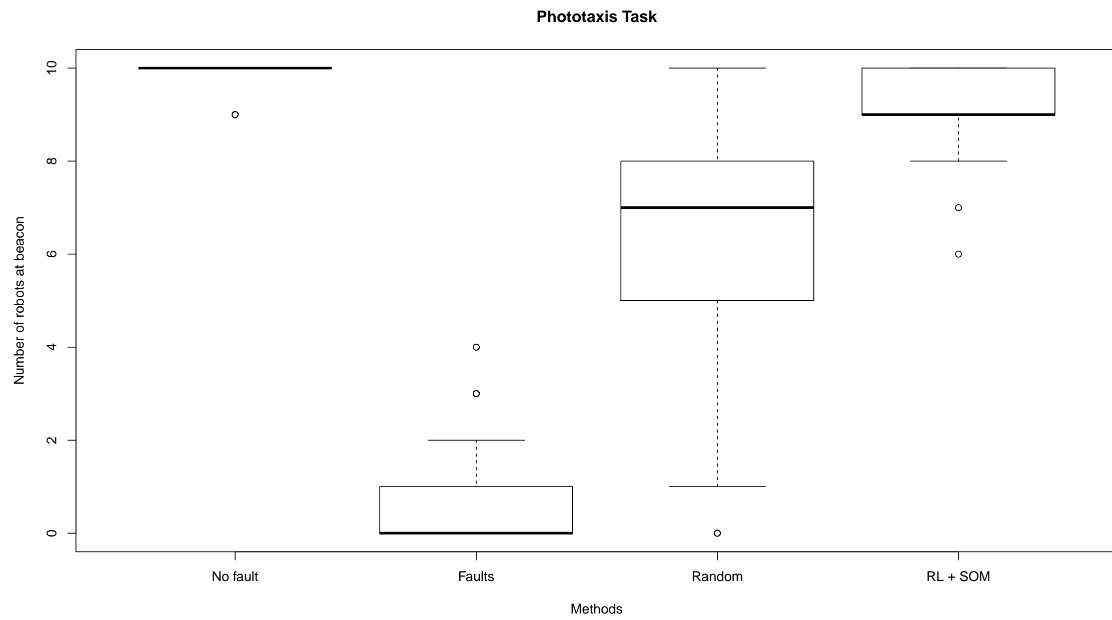


Figure 7.11: Collective Phototaxis: Motor failures

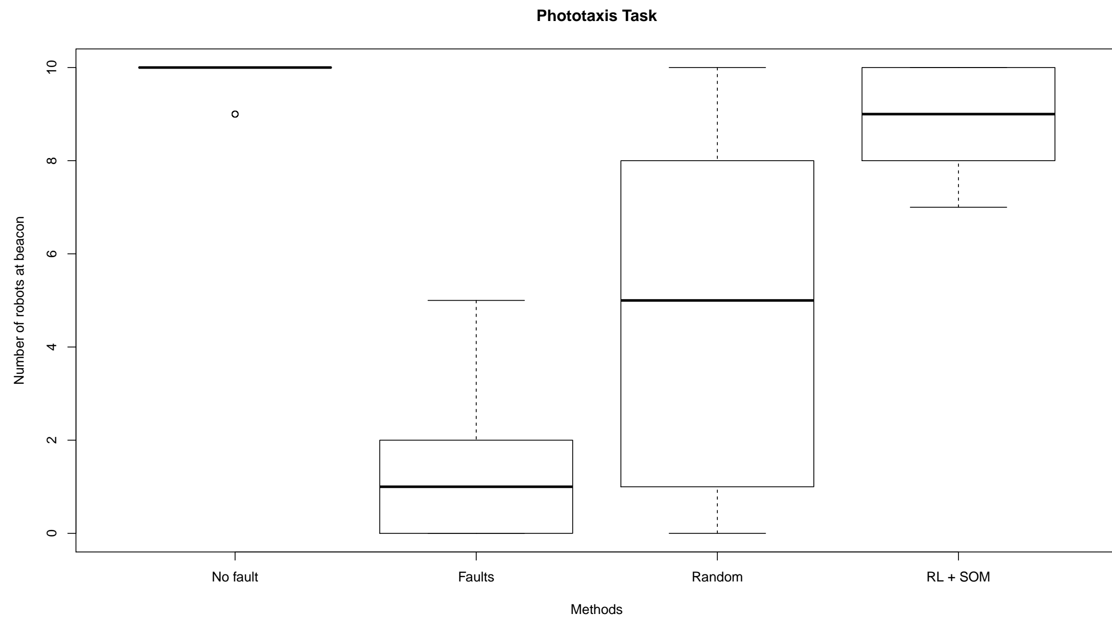


Figure 7.12: Collective Phototaxis: Complete Sensor Failures

7.5.1.2 Two different faults at different times

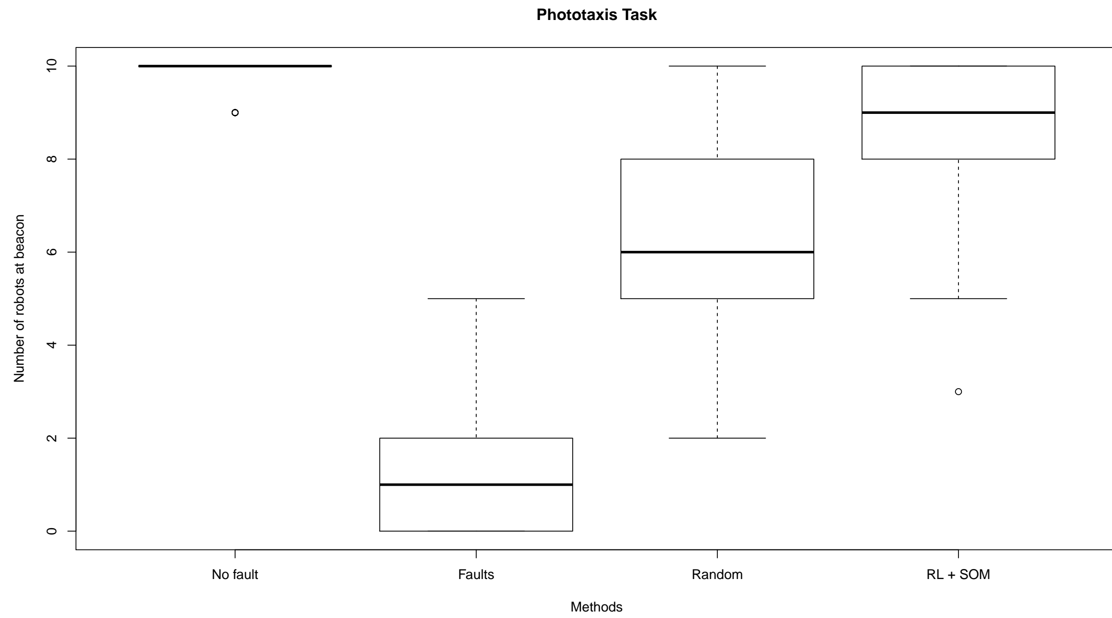
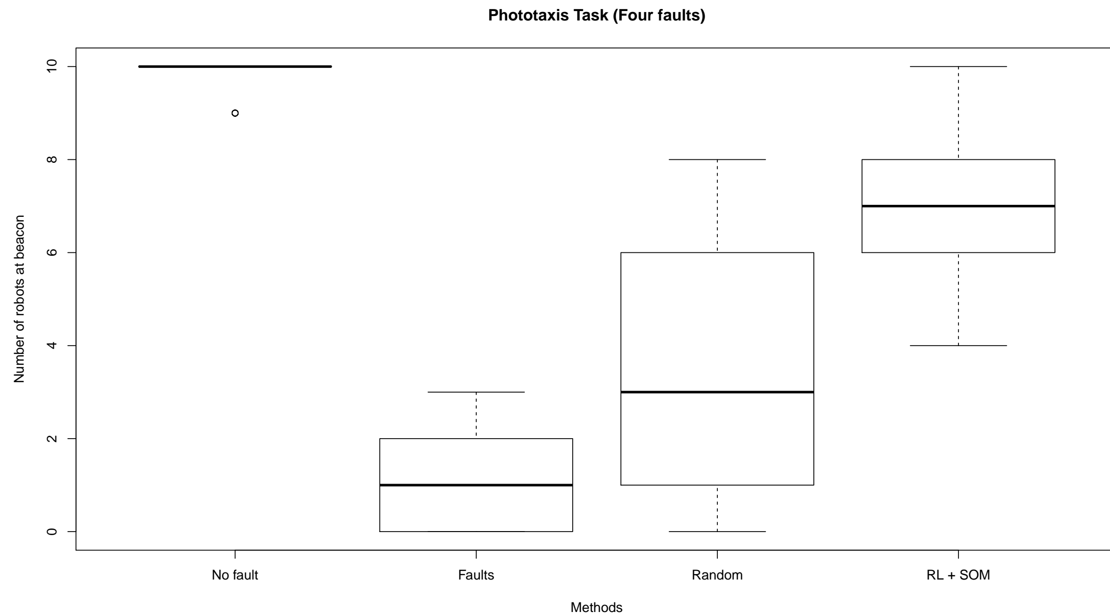
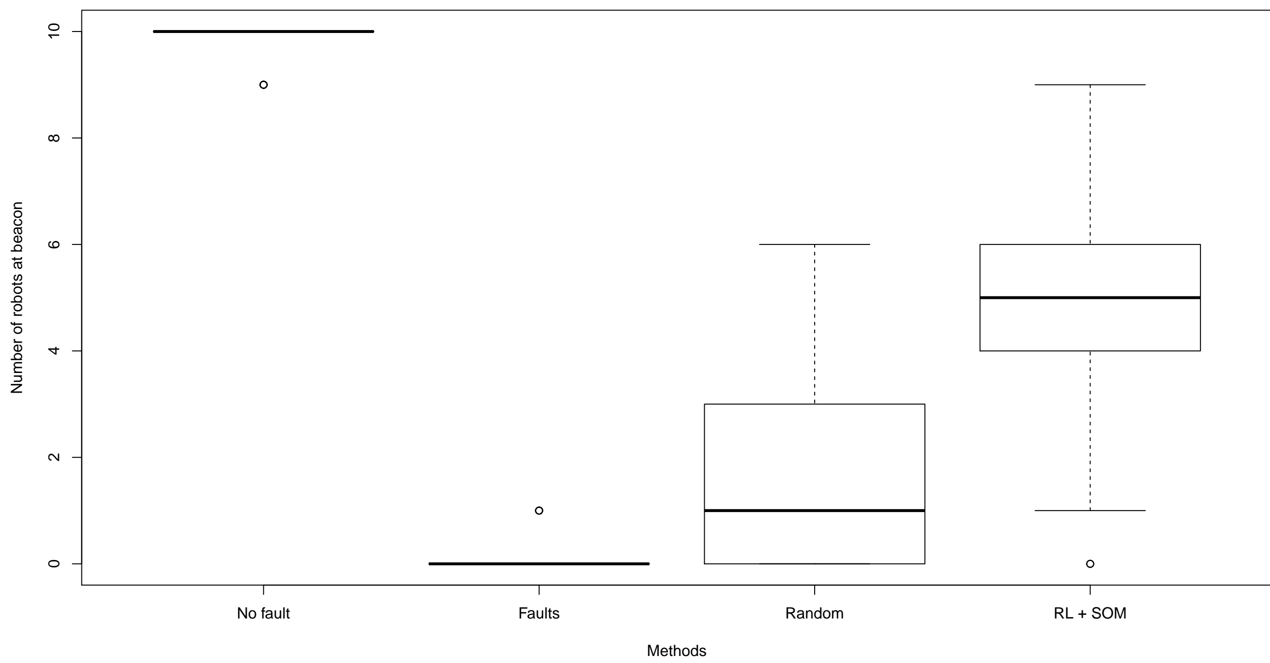


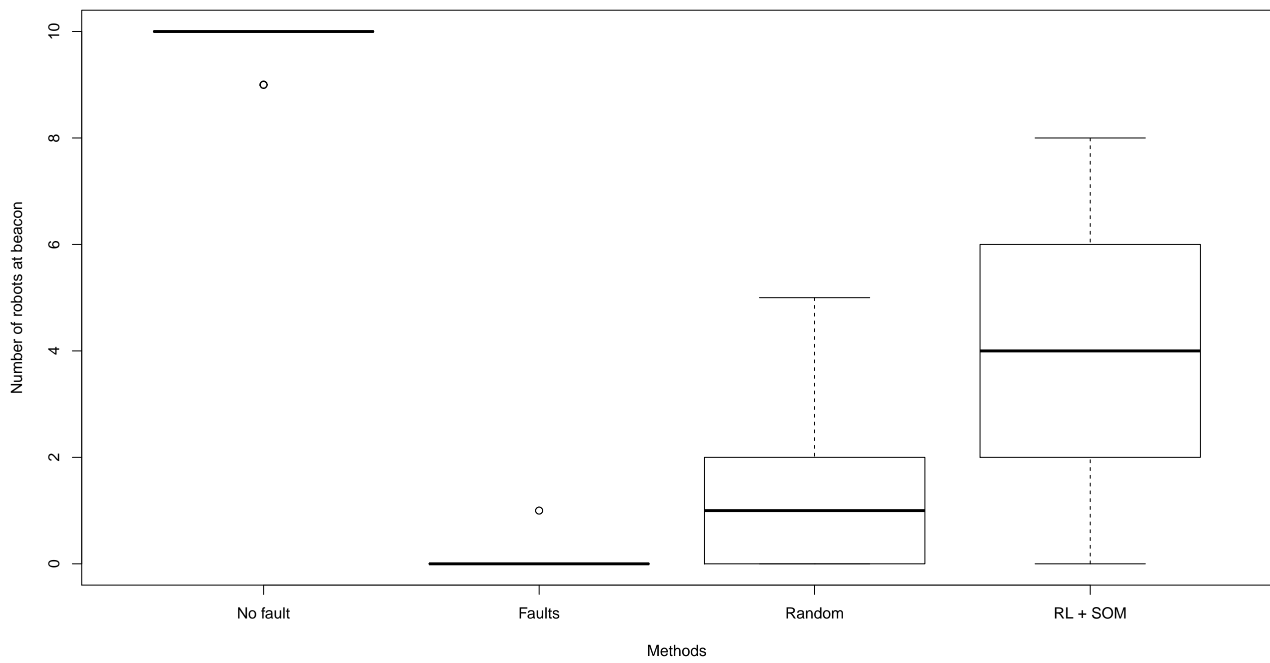
Figure 7.13: Collective Phototaxis: Multiple Failures

7.5.1.3 Multiple combination of faults at different times

Figure 7.14: The Plots below describe Collective Phototaxis for Multiple Failures when injected with 4,6 and 8 faults




Phototaxis Task (Six Faults)
Number of robots at beacon
0
2
4
6
8
10
No fault
Faults
Random
RL + SOM
Methods


Phototaxis Task (Eight Faults)
Number of robots at beacon
0
2
4
6
8
10
No fault
Faults
Random
RL + SOM
Methods

7.5.2 Aggregation

Aggregation involves the robots in a swarm coming together from around the environment to form one cohesive aggregate in the environment. The robots in the swarm make use of the range and bearing actuators and sensors onboard to sense the other robots and objects in the environment to move towards each other to form one aggregate. The results from the scenarios described in the previous section is shown below:

7.5.2.1 Fault at different times

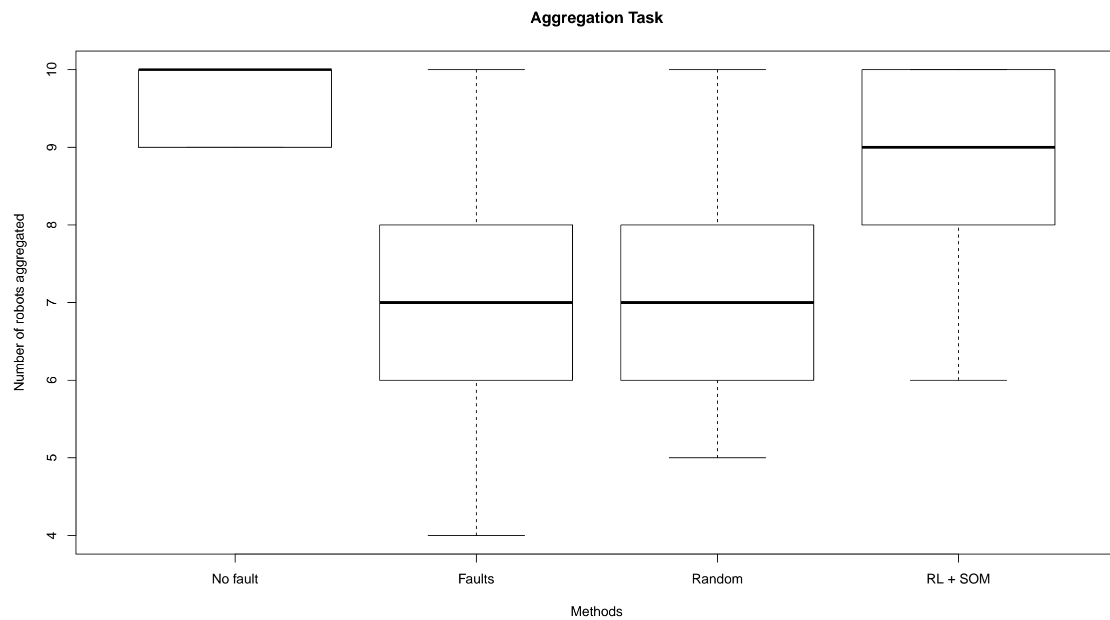
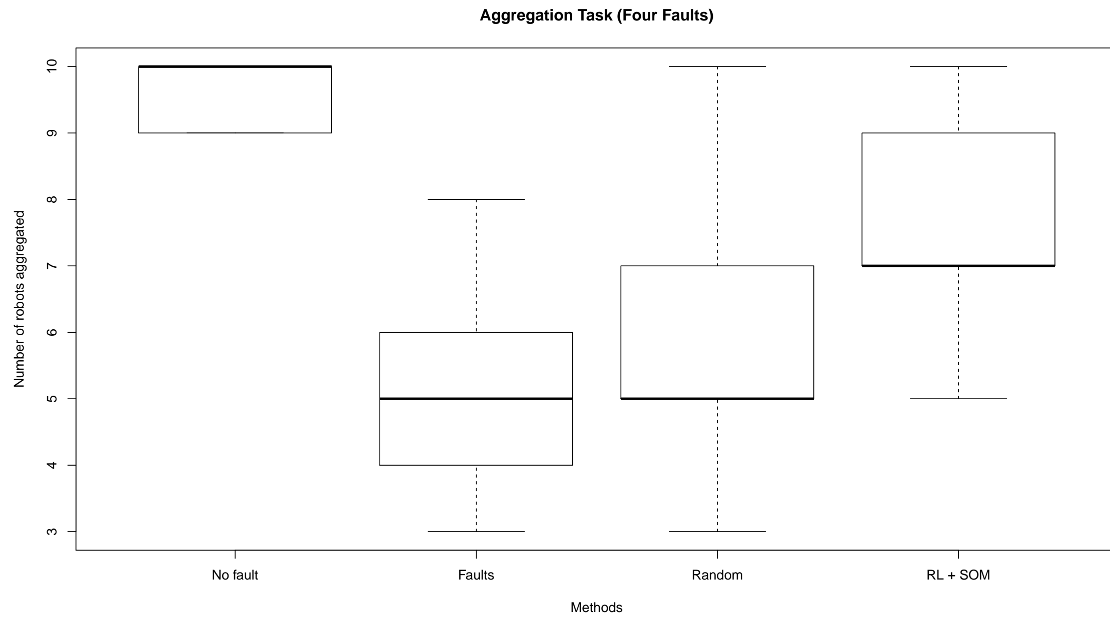
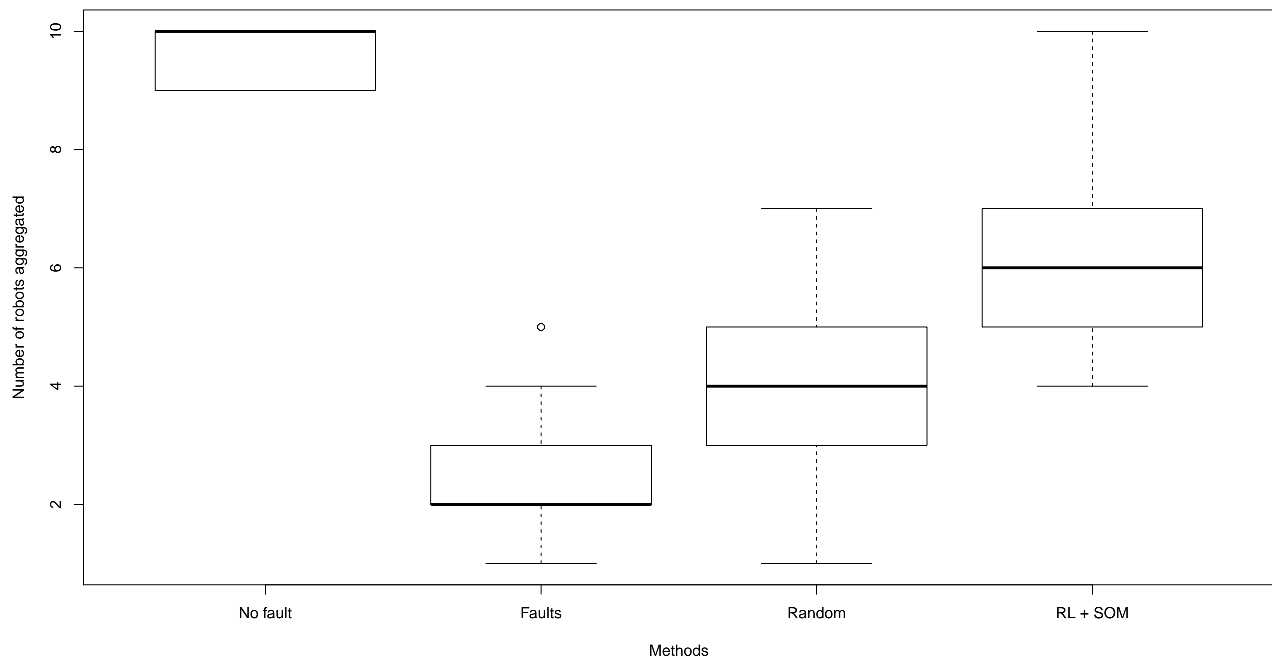


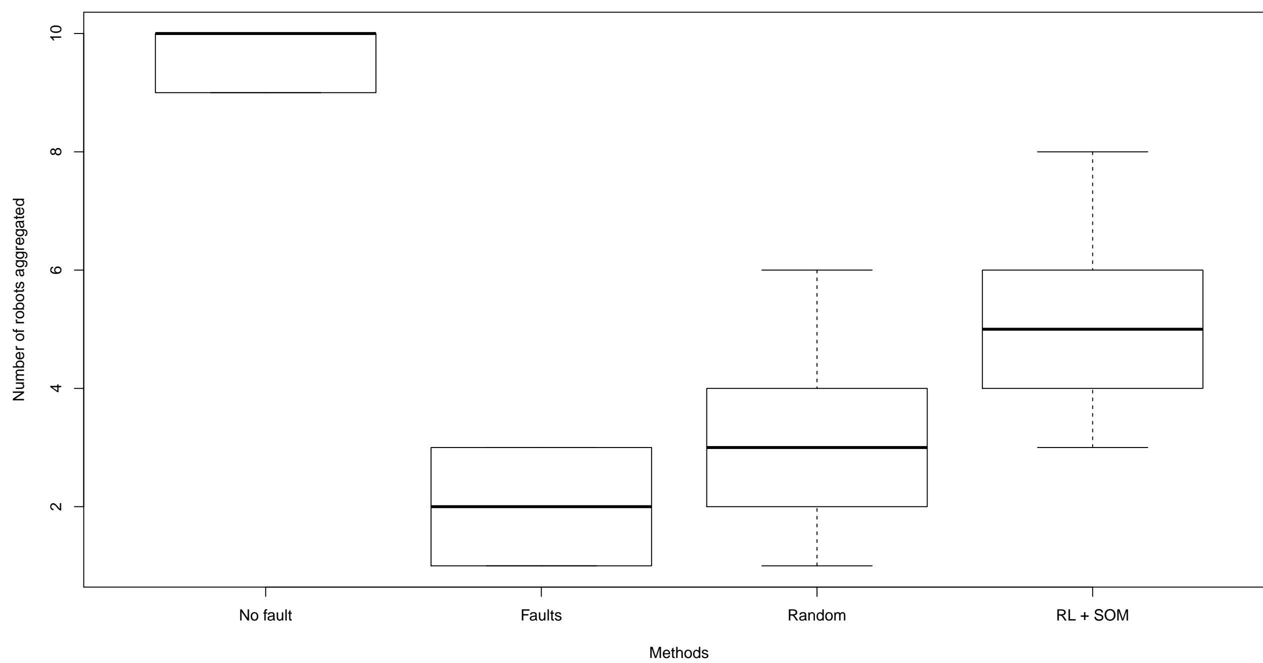
Figure 7.15: Aggregation: Complete Sensor Failures

7.5.2.2 Multiple faults at different times

Figure 7.16: The Plots below describe Aggregation for Multiple Failures when injected with 4,6 and 8 faults




Aggregation Task (Six Faults)
Number of robots aggregated
10
8
6
4
2
No fault
Faults
Random
RL + SOM
Methods


Aggregation Task (Eight Faults)
Number of robots aggregated
10
8
6
4
2
No fault
Faults
Random
RL + SOM
Methods

7.5.3 Foraging

Foraging involves robots in the swarm moving from the 'base' to the environment in search for food items. The robots search the environment for food, collect them and bring them back to base. The goal is for the robots in the swarm to search the entire environment and collect all the 'food items' in the items and return back to the base station. There are light sensors present at the base station which allows the robots to return back to the base station safely to drop the food items and continue searching for food items located in the environment.

7.5.3.1 Two faults at different times

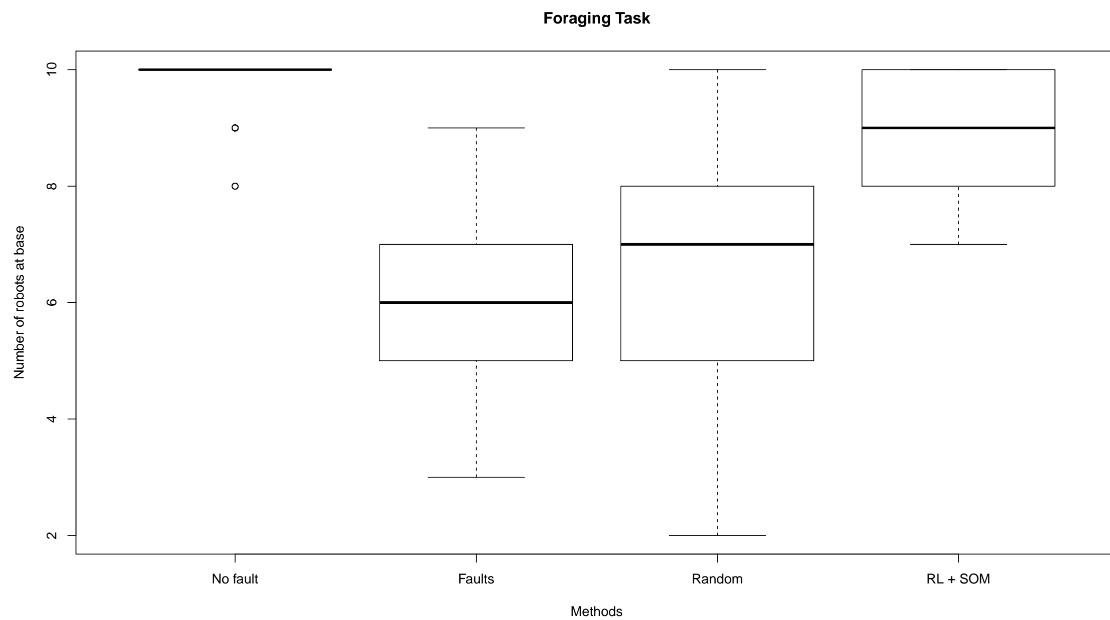


Figure 7.17: Foraging: Light Sensor Failure (Number of Robots at Base)

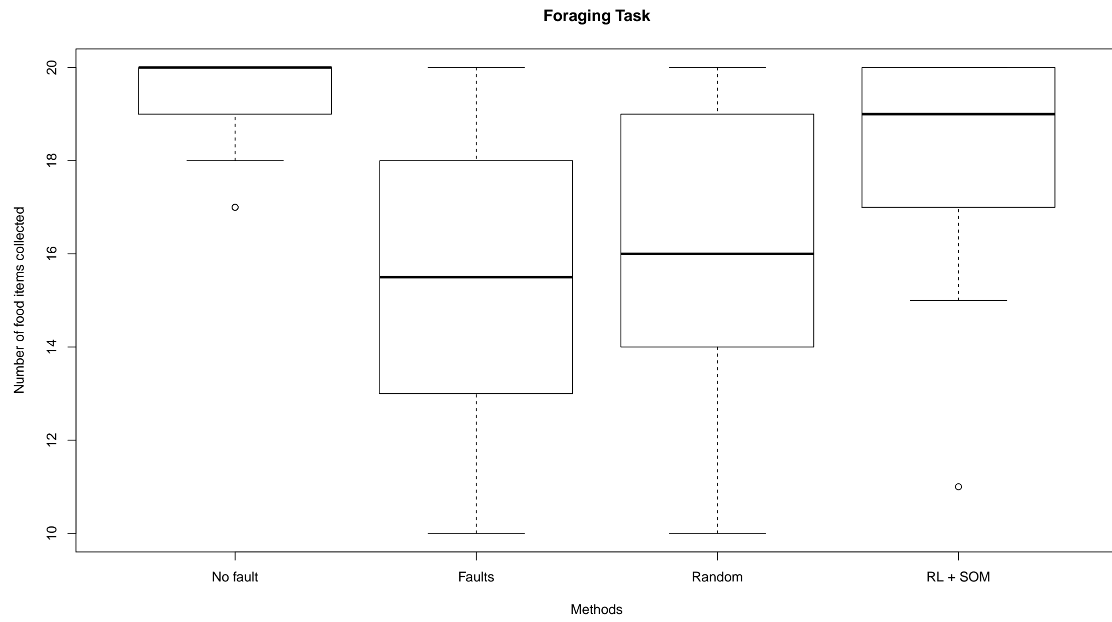
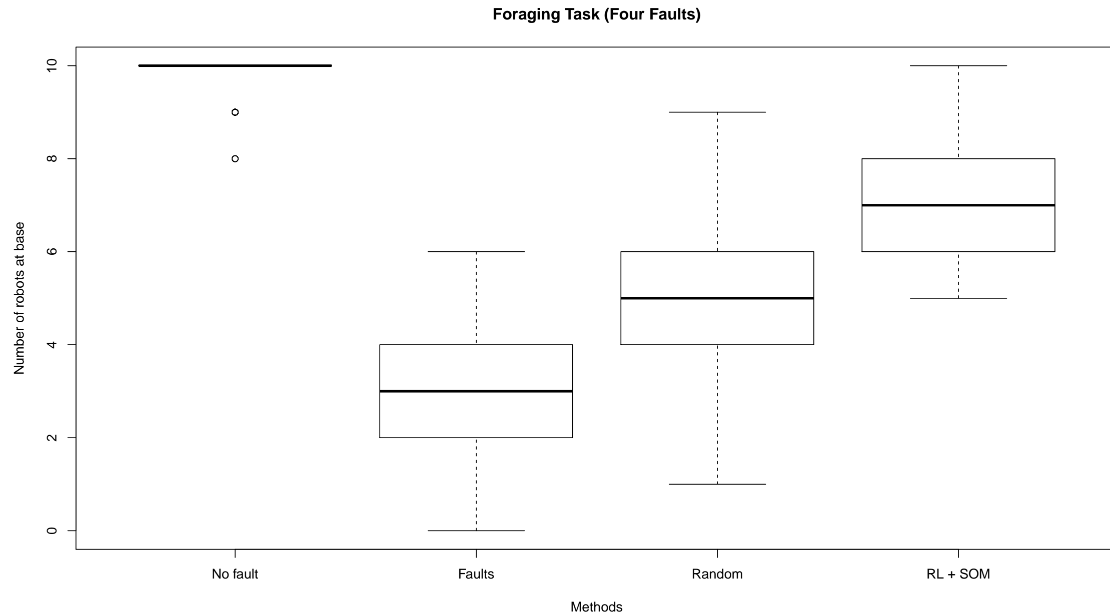
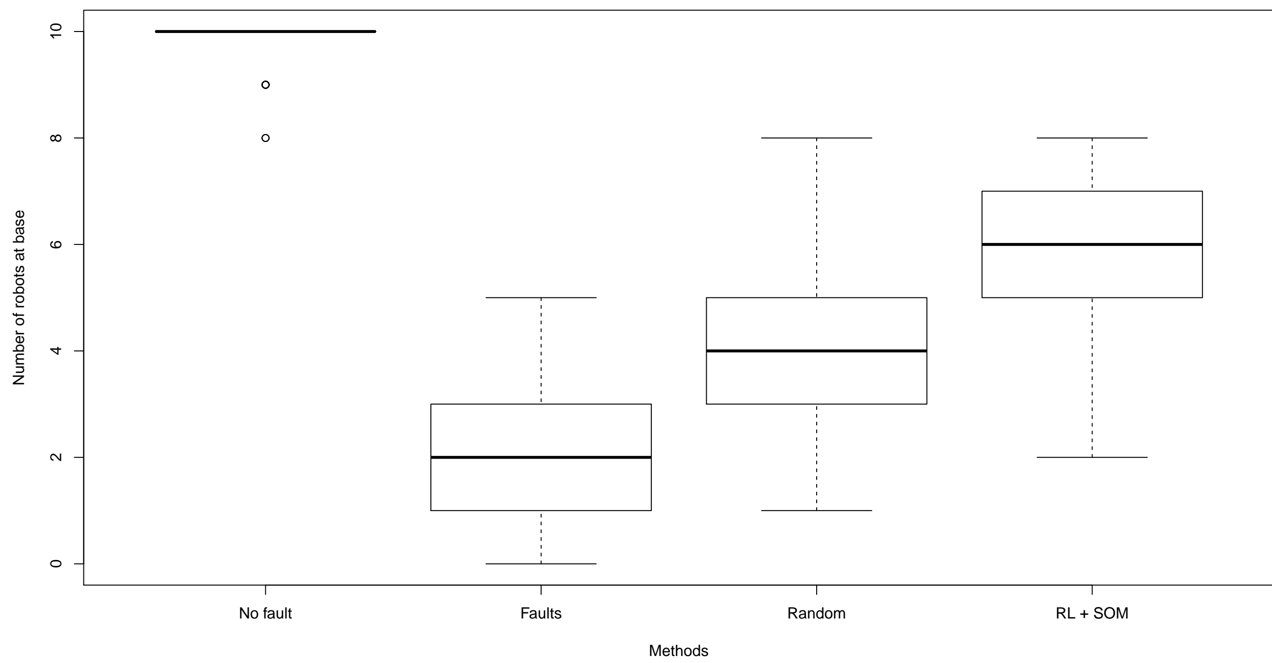


Figure 7.18: Foraging: Light Sensor Failure (Food Items Collected)

7.5.3.2 Multiple faults at different times

Figure 7.19: The Plots below describe Foraging for Light Sensor Failure (Number of Robots at Base) when injected with 4,6 and 8 faults




Foraging Task (Six Faults)
Number of robots at base
0
2
4
6
8
10
No fault
Faults
Random
RL + SOM
Methods

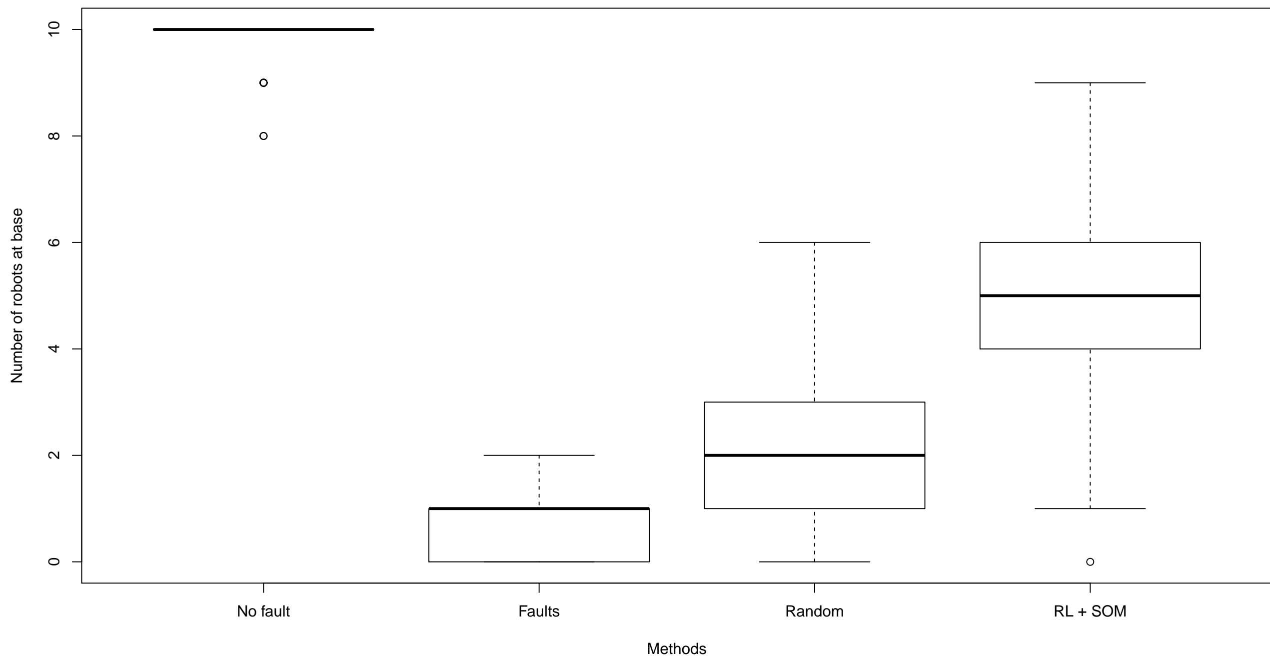
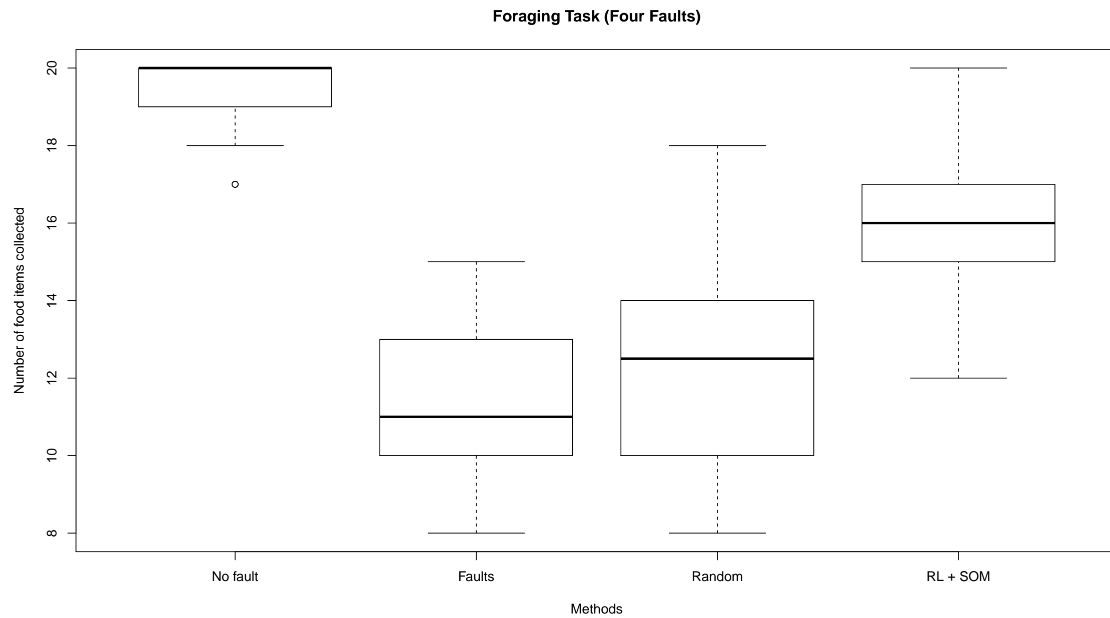
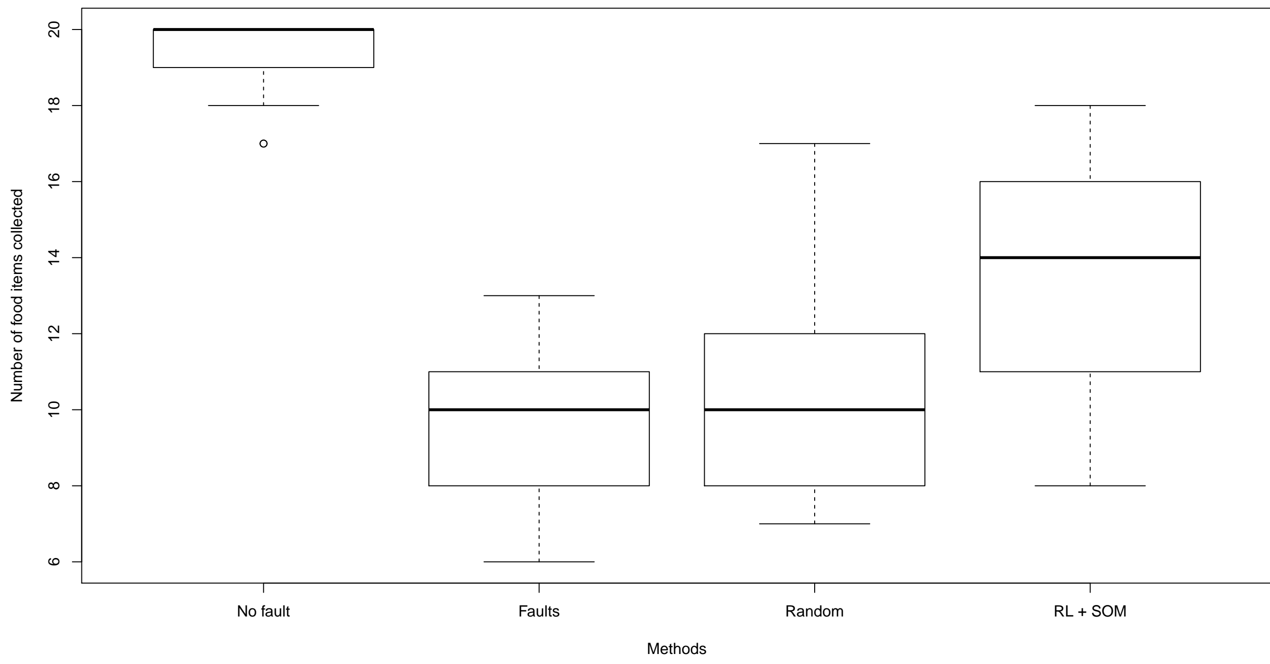
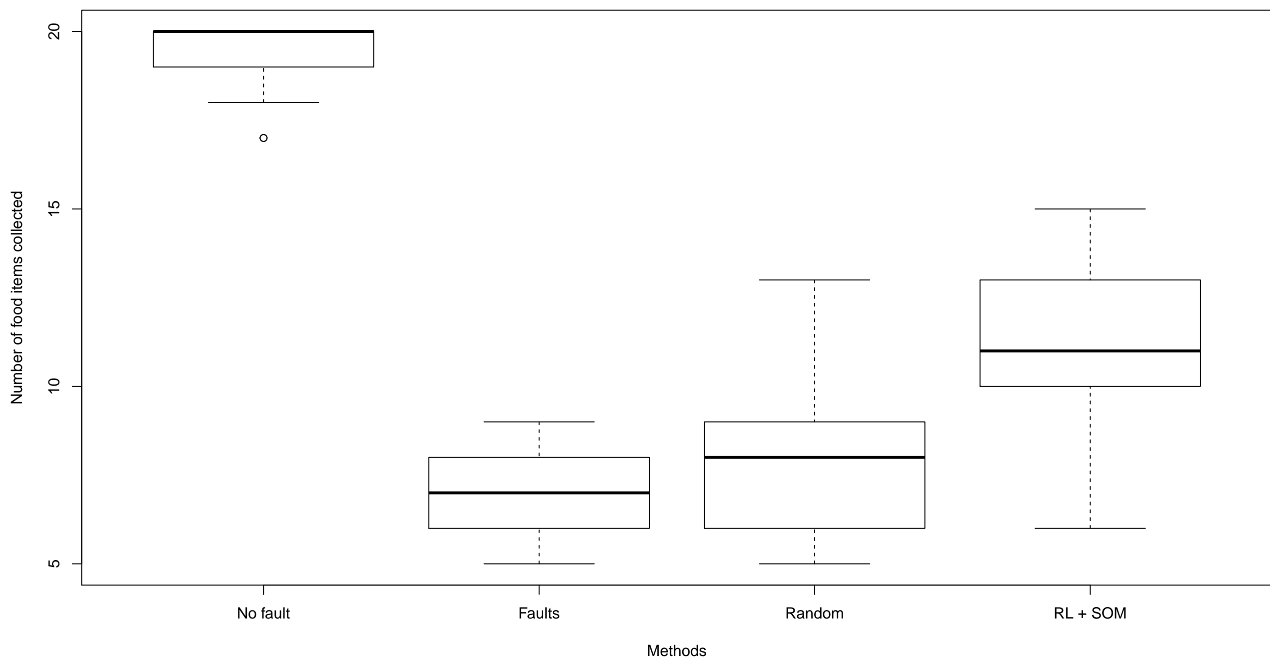

Foraging Task (Eight Faults)
Number of robots at base
0
2
4
6
8
10
No fault
Faults
Random
RL + SOM
Methods

Figure 7.20: The plots below describe Foraging for Light Sensor Failure (Food Items Collected) when injected with 4,6 and 8 faults




Foraging Task (Six Faults)
Number of food items collected
Methods
No fault
Faults
Random
RL + SOM
6
8
10
12
14
16
18
20


Foraging Task (Eight Faults)
Number of food items collected
20
15
10
5
No fault
Faults
Random
RL + SOM
Methods

In this section, we describe the results for various scenarios when using the distributed learnt architecture for this testing phase. As we have described in the experiment overview, the results that are shown are what we expect when testing for these different scenarios. When fewer faults are injected into the swarm regardless of the type of fault, the learnt recovery strategy is able to cope well with these situations. However when the number of faults increases, although the learnt recovery strategy is able to cope, the performance is not as optimal as the scenarios with fewer. The reason for this has been explained in the introduction but we still observe that although the learnt recovery strategy is not optimal (for these scenarios), it still performs better than randomly selecting the predefined behaviours and this goes for all the tasks tested in this section.

Part III

Conclusion and Future Work

Chapter 8

Conclusion

In this thesis, a fault recovery solution was presented for swarm robotic systems based on a swarm of robots utilising a combination of learning algorithms: self-organising maps and reinforcement learning. These learning algorithms are used to learn, before a fault has actually occurred, a set of predefined recovery mechanisms to determine the most appropriate behaviour for a fault recovery operation. The learning is done in simulation using randomly generated data and is learnt and tested in both centralised and distributed settings; they are not to be necessarily compared to each other, rather it is a natural progression to learn in a centralised setting and then upgrade the learning architecture to work in a distributed setting as the robot platform that is being used in this thesis is swarm robots. It is assumed that there is already an architecture within the swarm that is capable to detect and diagnose faults and at the point of fault detection and diagnosis within the task run, the robots are able lookup their 'experiences' to select the appropriate strategy for any state and task. The learning architecture is tested in a distributed setting because the swarm is inherently distributed therefore, having a learning architecture that follows similar rules without having to manipulate how the swarm functions.
The pre-defined behaviours to be learnt includes the following:

Transport to repair station: There is a repair station where faulty robots can be taken to be repaired. This behaviour involves the assisting robots gripping the faulty robot and dragging it to the repair station. The chosen robot(s) returns to the task, leaving behind the faulty robot to be fixed.

Repair on the spot: Following [25], it is assumed that each robot has the ability to repair other robots in the swarm, and that the robots have access to a repertoire of recovery mechanisms which can fix common faults. This behaviour could

be especially useful if a faulty robot is very important and needs to resume its task immediately. However, this takes a significant amount of time and energy. Each fault takes a different amount of time to fix, it takes less time to fix the faulty robot if more assisting robots are recruited. However, there is a limit to how many robots make a difference for the 'cost' of the repair.

Drag Along: This behaviour requires only one robot to drag a faulty robot along. When the 'helper' robot gets to the faulty robot, it grips it and continues on with its task. It should be noted that it takes energy to drag a robot along; therefore it needs to be included when calculating the reward.

Leader-Follower: This behaviour also requires only one robot and does not work for specific faults: complete/partial motor failure and power failure. The faulty robot copies behaviour of helper robot.

The robots are able to learn the most best behaviour that has a minimised cost of repair for different possible states that the swarm might encounter, where the each state vector variable can be defined as follows:

Distance from the faulty robot d_1 ... d_3 describes how far the nearest robot(s) is from the faulty robot.

'How busy' nearby robots are b_1 ... b_3 describes how busy the nearest robot(s) is. The designer can decide how the swarm assigns the 'busy' rating. This is rated on a discrete scale from 0 to 5, where 0 means the robot is not busy and 5 signifies that the robot is very busy. If the robot is not busy, then it can tend to the faulty robot immediately, but as the robot 'busyness' increases, the longer it takes for the robot to be deployed. Although swarms are homogeneous in nature, there are some tasks where different robots have different capabilities and also different sub-tasks. This property is especially useful in these areas; for example in foraging, where other robots are searching for food, some robots have found food items and are carrying them back to the base.

Power left p_1 ... p_3 describes the amount of power left in the robot at the time that the fault is detected. This is in percentage, so as to make calculating the reward easier.

Importance of the faulty robot I describes how important a faulty robot is. For example, if it is actively busy with a task, e.g. transporting an object in a foraging or search-and-rescue task, it will be considered more important. The

designer can decide how the swarm assigns the importance rating. This is rated on a discrete scale from 0 to 5, where 0 means not important and 5 signifies that the robot is very important. If a faulty robot is not important, the recovery process does not aim for fast repair, but at the same time, we want to reduce the overall cost of the repair.

Distance to repair station d_{rb} describes how far the faulty robot is from a repair station (in meters).

The possible faults, which have a varying effect on the swarm depending on the task, that are tested includes: complete motor failure, communication (sensor) failure and power failure. The tasks that are observed during the experiments are aggregation, collective phototaxis and foraging.
It should also be noted that the learning architecture for both the centralised and distributed approach, in chapters 4 and 5, has been tested and learnt in a cluttered environment. The main difference between the cluttered and uncluttered environment is that in the cluttered environment there are obstacles present in the environment that could obstruct robots the robots involved in the recovery from completing the recovery task that has been assigned and a major obstacle that could potentially block the predefined behaviour of 'drag to the base'. This is to account for potential situations where for some reason or the other, the initial chosen recovery strategy cannot be completed, especially the 'drag to base' predefined behaviour, it is able to change behaviour to ensure that the recovery process is completed.
The results, as described in chapters 4 and 5, show that generally, the swarm is able to select the appropriate recovery strategy when presented with a fault outperforming random selection when utilising both the centralised and distributed approaches to solving faults in swarms. However, there are some limitations to the work presented in this thesis. The learnt recovery strategy presented in this thesis is not quite adaptive especially with the limited states that are used for the learning process. However, adding more states, increases the learning time which is not a desired effect but it is necessary to ensure that the learning process has enough time to learn adequately. When presented with new states within the context of what has been learnt previously, the swarm is able to cope with this fault recovery but when pushed outside the constraints of the learning done, some complications would arise. That is, there would be some situations where the present fault recovery architecture would not be able solve or cope with; what is being discussed here is unknown and more complicated environments.
In chapter 6, an extension of the current experiments is presented where more combinations of faults are added within the swarm run for both the centralised and

distributed approaches. Effectively, the system is stress-tested to observe the behaviour of the algorithm when it is pushed to recover from some 'stressed' scenarios. As expected, the algorithm did not work in all of these scenarios as more robots are actively involved in the recovery; the cost of repair becomes more expensive as more power is being used during the recovery process. More robots would not have enough power to complete the task. Additionally, randomly selecting the predefined behaviours for the recovery process hardly copes with these sort of scenarios, however, the learnt recovery strategy copes better but as seen in this chapter, the learnt recovery strategy is still affected.

Additionally, there are set predefined behaviours that are not flexible during their execution. There are some scenarios that would require flexibility to deal with an adaptive environment or a fault that could manifest differently from the norm. Possible solutions are discussed in the next section as to a possible way to deal with this situation by extending the present architecture.

This thesis presents a new approach towards active fault recovery methods which adds to creating a complete fault tolerant system. Reinforcement Learning and Self Organising Maps have been used together previously therefore, it is not novel on its own, however using it in this context where it is being implemented in a centralised and distributed approach, in conjunction with having predefined behaviours to learn the most effective recovery strategy is a novel approach in fault recovery literature.

Chapter 9

Future Work

There are multiple possible future work solutions that can be done to improve the performance of the present fault recovery architecture.
A possible extension would be to test other predefined recovery mechanisms found in the literature and include it in the present fault recovery strategy. For example, work done by [19] where robots are able to power share with faulty robots that have experienced power failure could be possible predefined behaviour. The present architecture allows for as many possible predefined recovery mechanisms but adding more will come at the cost of a larger training set and additional learning time.
Currently, the learning is done off-line but a future extension is to allow on-line learning using on-board simulators on the swarm robots [109], [50]. Although in the decentralised, distributed approach, the learning is done on-board the robots, it's not in an on-line leaning format where whilst the robots are actively participating in their environment, task and even during a fault recovery process. The future extension would allow continuous learning throughout the robot's life span where information, from current fault recovery solutions, tasks done etc. is fed into the on-board simulator and learning tool to allow for a more accurate and efficient learning process.
In the distributed approach chapter, a traditional consensus method was used to allow the robots to reach an agreement on the selection of the appropriate fault recovery solution. To allow for a more distributed approach, a possible extension would be to extend the present architecture to allow for distributed consensus.
Additionally, in the previous section, it was discussed that the present learning architecture is not quite adaptive and flexible and will be unable to deal with changing and unknown environments as it is not possible to learn every possible scenario that the swarm could ever be presented. A model of the environment might be used

before-hand but this is not an ideal solution. Ideally, a model free environment would be preferable when it comes to the fault recovery process.
To achieve this, we propose utilising evolutionary approaches to design an 'adaptive' fault recovery swarm robotic system. Evolution can be used to produce novel solutions to problems which could be different from what a human designer might decide to write. Using evolutionary approaches in robotics is an area in robotics that is being researched and it is called Evolutionary Robotics (ER). ER is a term that is used to describe a method that uses evolutionary techniques in the development of controllers for robots. ER creates autonomous robot controllers. It does not require human designers designing the controller or having complete knowledge of the environment that the robots would be deployed in [110], [111]. This theory is applied in the initial experiment tested and although the test failed, there are still merit in utilising this approach and extending the learning architecture by including some form of evolutionary robotics. For example, the next step could be to evolve learnt predefined behaviours that would involve the 'helper' robots that can aid in the recovery process by 'fixing' the faulty robots, utilising decision trees in conjunction with the learning approach to select the most efficient recovery strategy and also using evolutionary approaches so as to create new behaviours or evolve the controllers that can be used for major faults in various swarm behaviours (aggregation, foraging etc.) which would make recovery process adaptive to different unknown scenarios.

Bibliography

[1] S. S. Haykin, S. S. Haykin, S. S. Haykin, and S. S. Haykin, *Neural networks and learning machines*. Pearson Upper Saddle River, 2009, vol. 3.

[2] (2015). [Online]. Available: http://argos-sim.info/plow2015/

[3] (2018, February). [Online]. Available: http://www.e-puck.org/

[4] M. Brambilla, E. Ferrante, M. Birattari, and M. Dorigo, "Swarm robotics: a review from the swarm engineering perspective," *Swarm Intelligence*, vol. 7, no. 1, pp. 1–41, 2013.

[5] A. F. Winfield and J. Nembrini, "Safety in numbers: fault-tolerance in robot swarms," *International Journal of Modelling, Identification and Control*, vol. 1, no. 1, pp. 30–37, 2006.

[6] E. Sahin, S. Girgin, L. Bayindir, and A. E. Turgut, "Swarm robotics." *Swarm intelligence*, vol. 1, pp. 87–100, 2008.

[7] E. Şahin, "Swarm robotics: From sources of inspiration to domains of application," in *International workshop on swarm robotics*. Springer, 2004, pp. 10–20.

[8] J. D. Bjerknes and A. F. Winfield, "On fault tolerance and scalability of swarm robotic systems," in *Distributed Autonomous Robotic Systems*. Springer, 2013, pp. 431–444.

[9] L. Murray, "Fault tolerant morphogenesis in self-reconfigurable modular robotic systems," Ph.D. dissertation, University of York, 2013.

[10] J. D. Bjerknes, *Scaling and fault tolerance in self-organized swarms of mobile robots*. University of the West of England, 2010.

[11] M. Yogeswaran, S. Ponnambalam, and G. Kanagaraj, "Reinforcement learning in swarm-robotics for multi-agent foraging-task domain," in *Swarm Intelligence (SIS), 2013 IEEE Symposium on*. IEEE, 2013, pp. 15–21.

[12] C. Ye, N. H. Yung, and D. Wang, "A fuzzy controller with supervised learning assisted reinforcement learning algorithm for obstacle avoidance," *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, vol. 33, no. 1, pp. 17–27, 2003.

[13] Z. Shi, J. Tu, Y. Li, and Z. Wang, "Adaptive reinforcement q-learning algorithm for swarm-robot system using pheromone mechanism," in *Robotics and biomimetics (ROBIO), 2013 IEEE international conference on*. IEEE, 2013, pp. 952–957.

[14] S. Garnier, J. Gautrais, and G. Theraulaz, "The biological principles of swarm intelligence," *Swarm Intelligence*, vol. 1, no. 1, pp. 3–31, 2007.

[15] G. Beni, "From swarm intelligence to swarm robotics," in *International Workshop on Swarm Robotics*. Springer, 2004, pp. 1–9.

[16] P.-P. Grassé, "La reconstruction du nid et les coordinations interindividuelles chezbellicositermes natalensis etcubitermes sp. la théorie de la stigmergie: Essai d'interprétation du comportement des termites constructeurs," *Insectes sociaux*, vol. 6, no. 1, pp. 41–80, 1959.

[17] M. G. Hinchey, R. Sterritt, and C. Rouff, "Swarms and swarm intelligence," *Computer*, vol. 40, no. 4, pp. 111–113, 2007.

[18] A. F. Winfield and J. Nembrini, "Emergent swarm morphology control of wireless networked mobile robots," in *Morphogenetic Engineering*. Springer, 2012, pp. 239–271.

[19] A. R. Ismail, "Immune-inspired self-healing swarm robotic systems," 2011.

[20] L. E. Parker, "Reliability and fault tolerance in collective robot systems," *Handbook on Collective Robotics: Fundamentals and Challenges, page To appear. Pan Stanford Publishing*, 2012.

[21] A. L. Christensen, R. O'Grady, M. Birattari, and M. Dorigo, "Exogenous fault detection in a collective robotic task," in *European Conference on Artificial Life*. Springer, 2007, pp. 555–564.

[22] A. L. Christensen, "Fault detection in autonomous robots," Ph.D. dissertation, Université libre de Bruxelles, 2008.

[23] N. Bayar, S. Darmoul, S. Hajri-Gabouj, and H. Pierreval, "Fault detection, diagnosis and recovery using artificial immune systems: A review," *Engineering Applications of Artificial Intelligence*, vol. 46, pp. 43–57, 2015.

[24] H. Lau, I. Bate, P. Cairns, and J. Timmis, "Adaptive data-driven error detection in swarm robotics with statistical classifiers," *Robotics and Autonomous Systems*, vol. 59, no. 12, pp. 1021–1035, 2011.

[25] A. L. Christensen, R. OGrady, and M. Dorigo, "From fireflies to fault-tolerant swarms of robots," *IEEE Transactions on Evolutionary Computation*, vol. 13, no. 4, pp. 754–766, 2009.

[26] J. Timmis, P. Andrews, N. Owens, and E. Clark, "An interdisciplinary perspective on artificial immune systems," *Evolutionary Intelligence*, vol. 1, no. 1, pp. 5–26, 2008.

[27] E. Hart and J. Timmis, "Application areas of ais: The past, the present and the future," *Applied soft computing*, vol. 8, no. 1, pp. 191–201, 2008.

[28] J. Timmis, P. Andrews, and E. Hart, "On artificial immune systems and swarm intelligence," *Swarm Intelligence*, vol. 4, no. 4, pp. 247–273, 2010.

[29] D. Tarapore, A. L. Christensen, P. U. Lima, and J. Carneiro, "Abnormality detection in multiagent systems inspired by the adaptive immune system," in *Proceedings of the 2013 international conference on Autonomous agents and multi-agent systems*. International Foundation for Autonomous Agents and Multiagent Systems, 2013, pp. 23–30.

[30] S. I. Roumeliotis, G. S. Sukhatme, and G. A. Bekey, "Sensor fault detection and identification in a mobile robot," in *Intelligent Robots and Systems, 1998. Proceedings., 1998 IEEE/RSJ International Conference on*, vol. 3. IEEE, 1998, pp. 1383–1388.

[31] A. Fagiolini, M. Pellinacci, G. Valenti, G. Dini, and A. Bicchi, "Consensus-based distributed intrusion detection for multi-robot systems," in *Robotics and Automation, 2008. ICRA 2008. IEEE International Conference on*. IEEE, 2008, pp. 120–127.

[32] G. Heredia, A. Ollero, R. Mahtani, M. Béjar, V. Remuß, and M. Musial, "Detection of sensor faults in autonomous helicopters," in *IEEE International Conference on Robotics and Automation*, vol. 2. IEEE; 1999, 2005, p. 2229.

[33] G. Heredia, B. Remu, A. Ollero, R. Mahtani, and M. Musal, "Actuator fault detection in autonomous helicopters," in *Proceedings of the 5th IFAC Symposium on Intelligent Autonomous Vehicles, Lisbon, Portugal*, 2004, pp. 569–574.

[34] A. G. Millard, J. Timmis, and A. F. Winfield, "Run-time detection of faults in autonomous mobile robots based on the comparison of simulated and real robot behaviour," in *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2014, pp. 3720–3725.

[35] A. L. Christensen, R. O'Grady, M. Birattari, and M. Dorigo, "Fault detection in autonomous robots based on fault injection and learning," *Autonomous Robots*, vol. 24, no. 1, pp. 49–67, 2008.

[36] J. A. Hilder, N. D. Owens, M. J. Neal, P. J. Hickey, S. N. Cairns, D. P. Kilgour, J. Timmis, and A. M. Tyrrell, "Chemical detection using the receptor density algorithm," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 42, no. 6, pp. 1730–1741, 2012.

[37] N. D. Owens, A. Greensted, J. Timmis, and A. Tyrrell, "The receptor density algorithm," *Theoretical Computer Science*, vol. 481, pp. 51–73, 2013.

[38] J. Bongard, V. Zykov, and H. Lipson, "Resilient machines through continuous self-modeling," *Science*, vol. 314, no. 5802, pp. 1118–1121, 2006.

[39] L. E. Parker, "Alliance: An architecture for fault tolerant multirobot cooperation," *IEEE transactions on robotics and automation*, vol. 14, no. 2, pp. 220–240, 1998.

[40] B. P. Gerkey and M. J. Mataric, "Sold!: Auction methods for multirobot coordination," *IEEE transactions on robotics and automation*, vol. 18, no. 5, pp. 758–768, 2002.

[41] M. B. Dias and A. Stentz, "Traderbots: A market-based approach for resource, role, and task allocation in multirobot coordination," 2003.

[42] S. Luke, *Essentials of Metaheuristics*, 2nd ed. Lulu, 2013, available for free at http://cs.gmu.edu/∼sean/book/metaheuristics/.

[43] C.-R. Hwang, "Simulated annealing: theory and applications," *Acta Applicandae Mathematicae*, vol. 12, no. 1, pp. 108–111, 1988.

[44] F. Glover and M. Laguna, *Tabu Search*. Springer, 2013.

[45] T. Bäck and H.-P. Schwefel, "An overview of evolutionary algorithms for parameter optimization," *Evolutionary computation*, vol. 1, no. 1, pp. 1–23, 1993.

[46] V. Trianni, *Evolutionary swarm robotics: evolving self-organising behaviours in groups of autonomous robots*. Springer, 2008, vol. 108.

[47] X. Yu and M. Gen, *Introduction to evolutionary algorithms*. Springer Science & Business Media, 2010.

[48] S. Nolfi and D. Floreano, "Evolutionary robotics," 2000.

[49] M. C. Schut, E. Haasdijk, and A. Eiben, "What is situated evolution?" in *2009 IEEE Congress on Evolutionary Computation*. IEEE, 2009, pp. 3277–3284.

[50] A. Eiben, E. Haasdijk, and N. Bredeche, "Embodied, on-line, on-board evolution for autonomous robotics," *Symbiotic Multi-Robot Organisms: Reliability, Adaptability, Evolution.*, vol. 7, pp. 361–382, 2010.

[51] P. J. O'Dowd, "An embodied simulation approach to distributed evolution for swarm robotic systems," Ph.D. dissertation, University of the West of England, Brstol, England, 2012.

[52] S. Elfwing, E. Uchibe, K. Doya, and H. I. Christensen, "Biologically inspired embodied evolution of survival," in *2005 IEEE Congress on Evolutionary Computation*, vol. 3. IEEE, 2005, pp. 2210–2216.

[53] N. Jakobi, P. Husbands, and I. Harvey, "Noise and the reality gap: The use of simulation in evolutionary robotics," in *European Conference on Artificial Life*. Springer, 1995, pp. 704–720.

[54] R. A. Watson, S. Ficiei, and J. B. Pollack, "Embodied evolution: Embodying an evolutionary algorithm in a population of robots," in *Evolutionary Computation, 1999. CEC 99. Proceedings of the 1999 Congress on*, vol. 1. IEEE, 1999.

[55] L. König, K. Jebens, S. Kernbach, and P. Levi, "Stability of on-line and onboard evolving of adaptive collective behavior," in *European Robotics Symposium 2008*. Springer, 2008, pp. 293–302.

[56] Y. U. Takaya and T. Arita, "Situated and embodied evolution in collective evolutionary robotics," in *In Proc. of the 8th International Symposium on Artificial Life and Robotics*. Citeseer, 2003.

[57] J. C. Bongard, "Evolutionary robotics," *Communications of the ACM*, vol. 56, no. 8, pp. 74–83, 2013.

[58] F. Silva, M. Duarte, L. Correia, S. M. Oliveira, and A. L. Christensen, "Open issues in evolutionary robotics," *Evolutionary computation*, vol. 24, no. 2, pp. 205–236, 2016.

[59] T. Back, "Selective pressure in evolutionary algorithms: A characterization of selection mechanisms," in *Evolutionary Computation, 1994. IEEE World Congress on Computational Intelligence., Proceedings of the First IEEE Conference on*. IEEE, 1994, pp. 57–62.

[60] J.-B. Mouret and S. Doncieux, "Incremental evolution of animats' behaviors as a multi-objective optimization," in *International Conference on Simulation of Adaptive Behavior*. Springer, 2008, pp. 210–219.

[61] R. C. Moioli, P. A. Vargas, F. J. Von Zuben, and P. Husbands, "Towards the evolution of an artificial homeostatic system," in *2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)*. IEEE, 2008, pp. 4023–4030.

[62] S. Celis, G. Hornby, and J. Bongard, "Avoiding local optima with user demonstrations."

[63] J. Lehman and K. O. Stanley, "Exploiting open-endedness to solve problems through the search for novelty." in *ALIFE*, 2008, pp. 329–336.

[64] J. Gomes, P. Urbano, and A. L. Christensen, "Evolution of swarm robotics systems with novelty search," *Swarm Intelligence*, vol. 7, no. 2-3, pp. 115–144, 2013.

[65] G. Cuccu and F. Gomez, "When novelty is not enough," in *European Conference on the Applications of Evolutionary Computation*. Springer, 2011, pp. 234–243.

[66] J. Lehman and K. O. Stanley, "Revising the evolutionary computation abstraction: minimal criteria novelty search," in *Proceedings of the 12th annual*

conference on Genetic and evolutionary computation. ACM, 2010, pp. 103–110.

[67] K. Deb, *Multi-objective optimization using evolutionary algorithms.* John Wiley & Sons, 2001, vol. 16.

[68] J.-B. Mouret and S. Doncieux, "Encouraging behavioral diversity in evolutionary robotics: An empirical study," *Evolutionary computation*, vol. 20, no. 1, pp. 91–133, 2012.

[69] J. D. Knowles, R. A. Watson, and D. W. Corne, "Reducing local optima in single-objective problems by multi-objectivization," in *International Conference on Evolutionary Multi-Criterion Optimization.* Springer, 2001, pp. 269–283.

[70] M. D. Soorati and H. Hamann, "The effect of fitness function design on performance in evolutionary robotics: The influence of a priori knowledge," 2015.

[71] A. L. Nelson, G. J. Barlow, and L. Doitsidis, "Fitness functions in evolutionary robotics: A survey and analysis," *Robotics and Autonomous Systems*, vol. 57, no. 4, pp. 345–370, 2009.

[72] N. Bredeche, J.-M. Montanier, W. Liu, and A. F. Winfield, "Environment-driven distributed evolutionary adaptation in a population of autonomous robotic agents," *Mathematical and Computer Modelling of Dynamical Systems*, vol. 18, no. 1, pp. 101–129, 2012.

[73] S. G. Ficici, R. A. Watson, and J. B. Pollack, "Embodied evolution: A response to challenges in evolutionary robotics," in *Proceedings of the eighth European workshop on learning robots*, 1999, pp. 14–22.

[74] S. Wischmann, K. Stamm, and F. Wörgötter, "Embodied evolution and learning: The neglected timing of maturation," in *European Conference on Artificial Life.* Springer, 2007, pp. 284–293.

[75] P. Krcah and D. Toropila, "Combination of novelty search and fitness-based search applied to robot body-brain co-evolution," in *Czech-Japan Seminar on Data Analysis and Decision Making in Service Science*, 2010, pp. 1–6.

[76] J. Lehman and K. O. Stanley, "Abandoning objectives: Evolution through the search for novelty alone," *Evolutionary computation*, vol. 19, no. 2, pp. 189–223, 2011.

[77] R. Das and D. Whitley, "The only challenging problems are deceptive: Global search by solving order-1 hyperplanes," in *ICGA*, vol. 91, 1991, pp. 166–173.

[78] G. Francesca, M. Brambilla, A. Brutschy, V. Trianni, and M. Birattari, "Automode: A novel approach to the automatic design of control software for robot swarms," *Swarm Intelligence*, vol. 8, no. 2, pp. 89–112, 2014.

[79] M. Dorigo, V. Trianni, E. Şahin, R. Groß, T. H. Labella, G. Baldassarre, S. Nolfi, J.-L. Deneubourg, F. Mondada, D. Floreano, *et al.*, "Evolving self-organizing behaviors for a swarm-bot," *Autonomous Robots*, vol. 17, no. 2-3, pp. 223–245, 2004.

[80] M. Duarte, V. Costa, J. Gomes, T. Rodrigues, F. Silva, S. M. Oliveira, and A. L. Christensen, "Evolution of collective behaviors for a real swarm of aquatic surface robots," *PloS one*, vol. 11, no. 3, p. e0151834, 2016.

[81] C. Pinciroli, V. Trianni, R. O'Grady, G. Pini, A. Brutschy, M. Brambilla, N. Mathews, E. Ferrante, G. Di Caro, F. Ducatelle, M. Birattari, L. M. Gambardella, and M. Dorigo, "ARGoS: a modular, parallel, multi-engine simulator for multi-robot systems," *Swarm Intelligence*, vol. 6, no. 4, pp. 271–295, 2012.

[82] M. Dorigo, D. Floreano, L. M. Gambardella, F. Mondada, S. Nolfi, T. Baaboura, M. Birattari, M. Bonani, M. Brambilla, A. Brutschy, *et al.*, "Swarmanoid: a novel concept for the study of heterogeneous robotic swarms," *IEEE Robotics & Automation Magazine*, vol. 20, no. 4, pp. 60–71, 2013.

[83] M. Mitchell, *An introduction to genetic algorithms.* MIT press, 1998.

[84] M. Wall, "Galib: A c++ library of genetic algorithm components," *Mechanical Engineering Department, Massachusetts Institute of Technology*, vol. 87, p. 54, 1996.

[85] M. W. Fagerland and L. Sandvik, "The wilcoxon–mann–whitney test under scrutiny," *Statistics in medicine*, vol. 28, no. 10, pp. 1487–1497, 2009.

[86] K. Alden, M. Read, J. Timmis, P. S. Andrews, H. Veiga-Fernandes, and M. Coles, "Spartan: a comprehensive tool for understanding uncertainty in simulations of biological systems," *PLoS Comput Biol*, vol. 9, no. 2, p. e1002916, 2013.

[87] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction.* MIT press Cambridge, 1998, vol. 1, no. 1.

[88] (2018, October). [Online]. Available: https://medium.com/@jonathan_hui/rl-introduction-to-deep-reinforcement-learning-35c25e04c199

[89] (2018, October). [Online]. Available: https://towardsdatascience.com/self-learning-ai-agents-part-ii-deep-q-learning-b5ac60c3f47

[90] (2018, July). [Online]. Available: https://mse238blog.stanford.edu/2018/07/tanuarya/drawbacks-of-deep-learning/

[91] (2018, March). [Online]. Available: https://towardsdatascience.com/why-deep-learning-is-needed-over-traditional-machine-learning-1b6a99177063

[92] (2018, April). [Online]. Available: https://www.freecodecamp.org/news/an-introduction-to-deep-q-learning-lets-play-doom-54d02d8017d8/

[93] (2016, December). [Online]. Available: https://medium.com/@ageitgey/machine-learning-is-fun-part-6-how-to-do-speech-recognition-with-deep-learning-28293c162f7a

[94] (2014, September). [Online]. Available: https://machinelearningmastery.com/discover-feature-engineering-how-to-engineer-features-and-how-to-get-good-at-it/

[95] (2012, January). [Online]. Available: https://en.wikipedia.org/wiki/Feature_engineering

[96] (2018, Nov). [Online]. Available: https://www.quora.com/What-are-the-advantages-and-disadvantages-of-deep-learning-Can-you-compare-it-with-the-sta

[97] (2018, February). [Online]. Available: https://bdtechtalks.com/2018/02/27/limits-challenges-deep-learning-gary-marcus/

[98] T. Kohonen, "The self-organizing map," *Neurocomputing*, vol. 21, no. 1-3, pp. 1–6, 1998.

[99] C. Pinciroli, V. Trianni, R. O'Grady, G. Pini, A. Brutschy, M. Brambilla, N. Mathews, E. Ferrante, G. Di Caro, F. Ducatelle, *et al.*, "Argos: a modular, multi-engine simulator for heterogeneous swarm robotics," in *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on*. IEEE, 2011, pp. 5027–5034.

[100] M. Alavi, G. M. Marakas, and Y. Yoo, "A comparative study of distributed learning environments on learning outcomes," *Information Systems Research*, vol. 13, no. 4, pp. 404–415, 2002.

[101] I. and Petuum. (2018, February) Intro to distributed deep learning systems - petuum, inc. - medium. [Online]. Available: https://medium.com/@Petuum/intro-to-distributed-deep-learning-systems-a2e45c6b8e7

[102] P. K. Chan, S. J. Stolfo, *et al.*, "Toward parallel and distributed learning by meta-learning," in *AAAI workshop in Knowledge Discovery in Databases*, 1993, pp. 227–240.

[103] M. J. Mataric, "Using communication to reduce locality in distributed multiagent learning," *Journal of experimental & theoretical artificial intelligence*, vol. 10, no. 3, pp. 357–369, 1998.

[104] M. Mataric, "Coordination and learning in multirobot systems," *IEEE Intelligent Systems and their Applications*, vol. 13, no. 2, pp. 6–8, 1998.

[105] G. Weiß, "Distributed reinforcement learning," in *The Biology and technology of intelligent autonomous agents*. Springer, 1995, pp. 415–428.

[106] L. Li, "Distributed learning in swarm systems: A case study," Tech. Rep., 2002.

[107] C. F. Touzet, "Distributed lazy q-learning for cooperative mobile robots," *International Journal of Advanced Robotic Systems*, vol. 1, no. 1, p. 1, 2004.

[108] J. Huang, B. Yang, and D.-y. Liu, "A distributed q-learning algorithm for multi-agent team coordination," in *2005 International Conference on Machine Learning and Cybernetics*, vol. 1. IEEE, 2005, pp. 108–113.

[109] N. Bredeche, E. Haasdijk, and A. Eiben, "On-line, on-board evolution of robot controllers," in *International Conference on Artificial Evolution (Evolution Artificielle)*. Springer, 2009, pp. 110–121.

[110] I. Harvey, E. Di Paolo, R. Wood, M. Quinn, and E. Tuci, "Evolutionary robotics: A new scientific tool for studying cognition," *Artificial life*, vol. 11, no. 1-2, pp. 79–98, 2005.

[111] I. Harvey, P. Husbands, D. Cliff, A. Thompson, and N. Jakobi, "Evolutionary robotics: the sussex approach," *Robotics and autonomous systems*, vol. 20, no. 2, pp. 205–224, 1997.

Part IV

Appendix

The appendix consists of further extensions of the experiments where the swarm size is increased and tested against the present swarm architecture to analyse the results and view how they fare with these scenarios.

Chapter 10

20 Robots in the Swarm

In this scenario, the experimental setup is similar to what has been described in centralised approach. However, it should be noted that there are more faults (6) injected here due to the increase in the swarm size. This is because, more faults are needed to get an adverse effect on the swarm behaviour. The results are displayed below for collective phototaxis, aggregation and foraging. For foraging, the number of food items to be collected are increased (40).

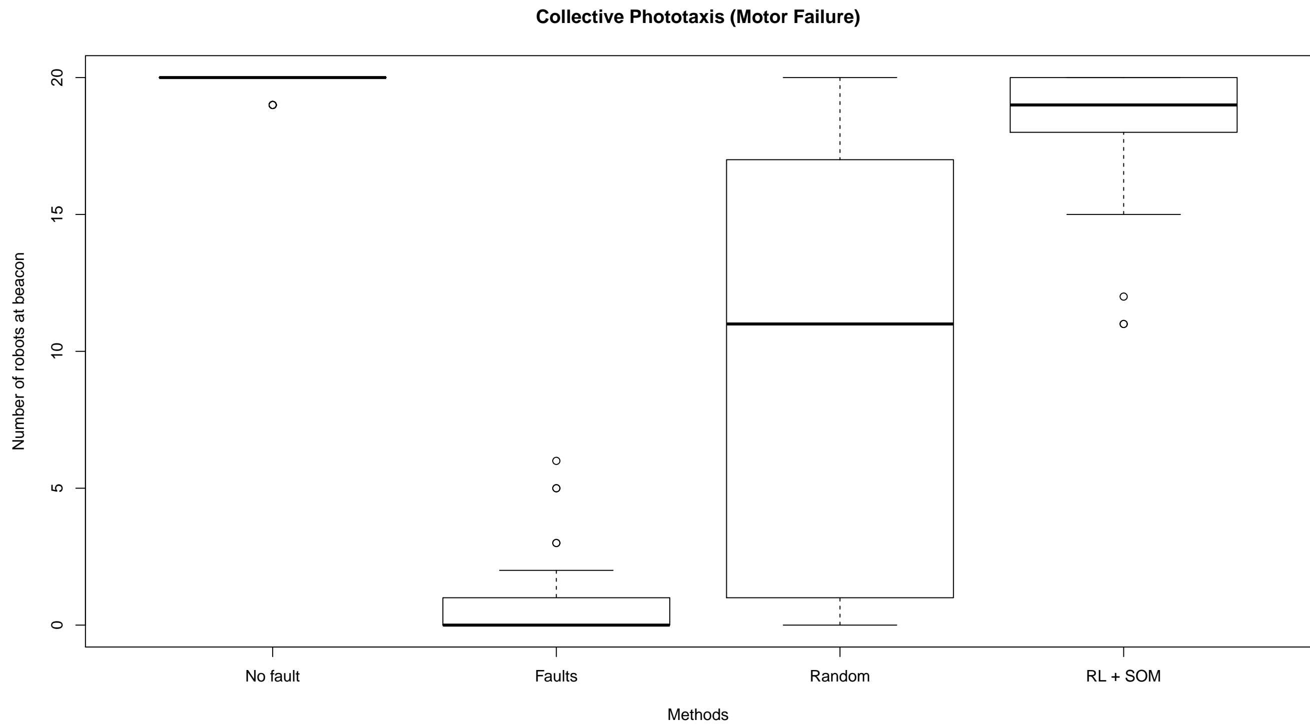


Figure 10.1: Collective Phototaxis: Motor failures

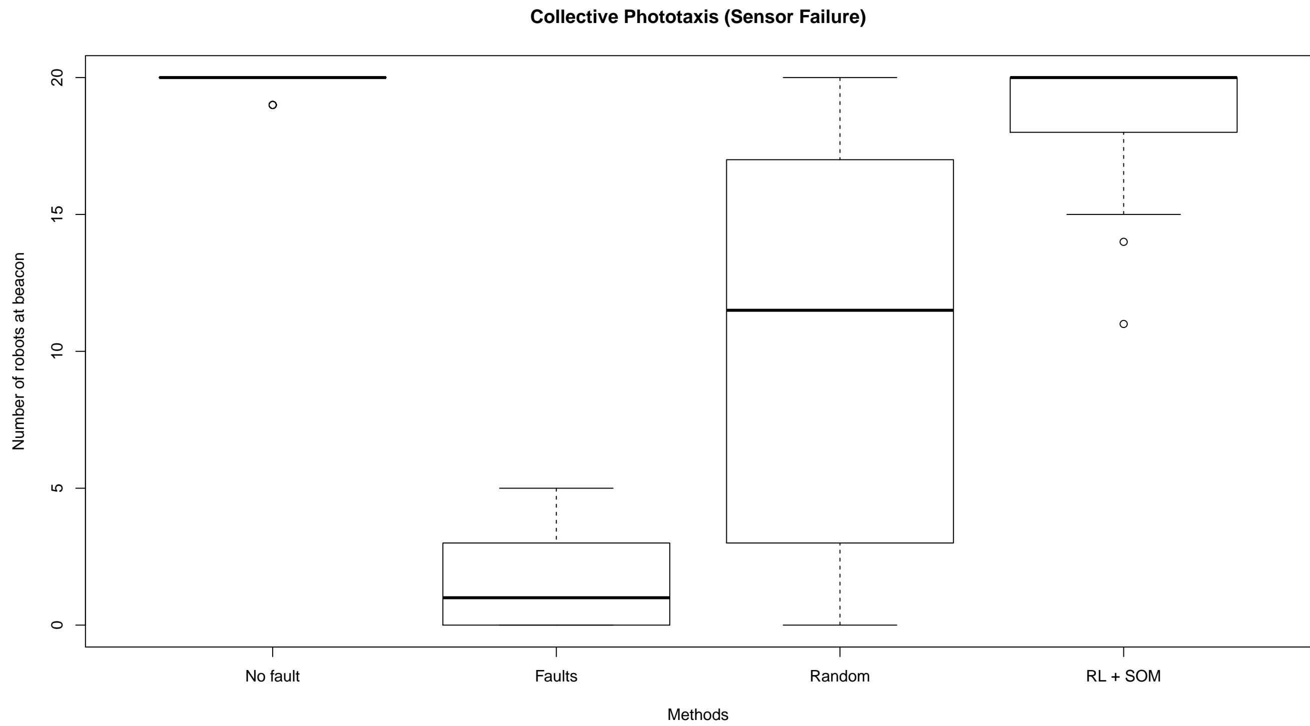


Figure 10.2: Collective Phototaxis: Communication sensor failures

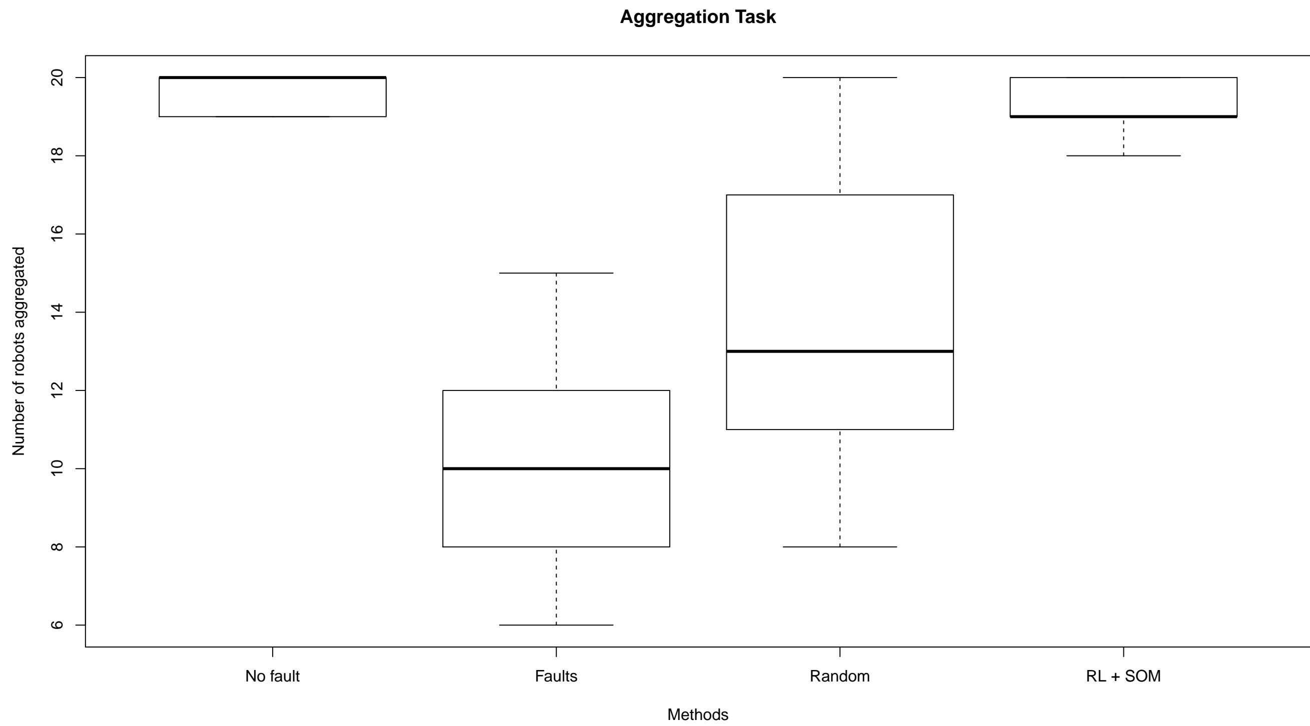


Figure 10.3: Aggregation: Communication sensor failures

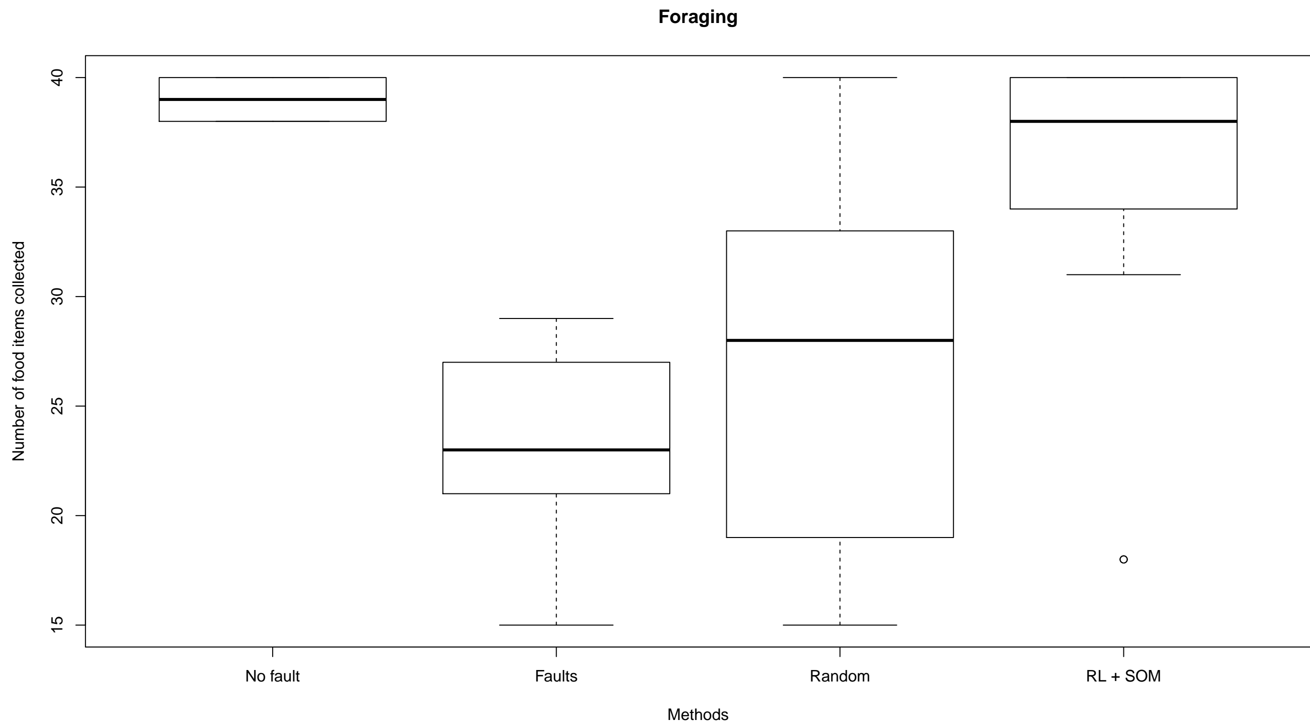


Figure 10.4: Foraging: Light Sensor (Number of Food Collected)

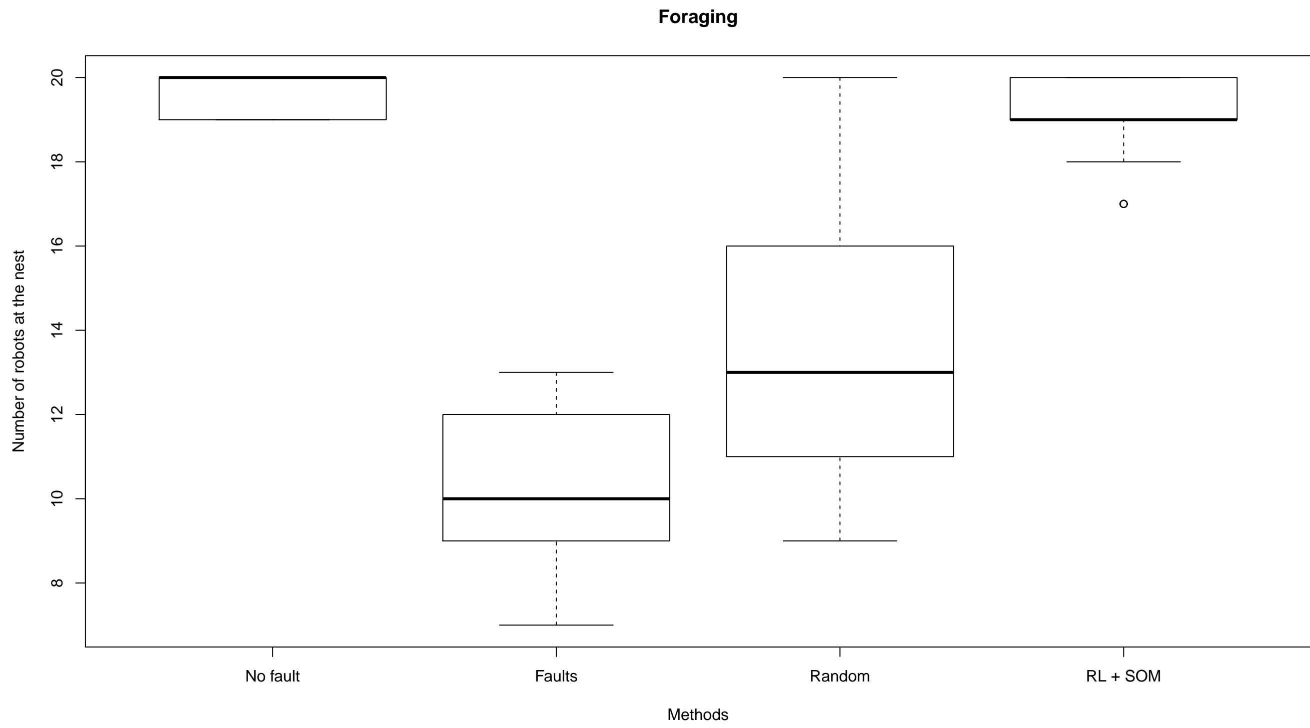


Figure 10.5: Foraging: Light Sensor (Number of Robots at Nest)

Chapter 11

40 Robots in the Swarm

In this scenario, the experimental setup is similar to what has been described in centralised approach. However, it should be noted that there are more faults (13) injected here due to the increase in the swarm size. This is because, more faults are needed to get an adverse effect on the swarm behaviour. The results are displayed below for collective phototaxis, aggregation and foraging. For foraging, the number of food items to be collected are increased (80).

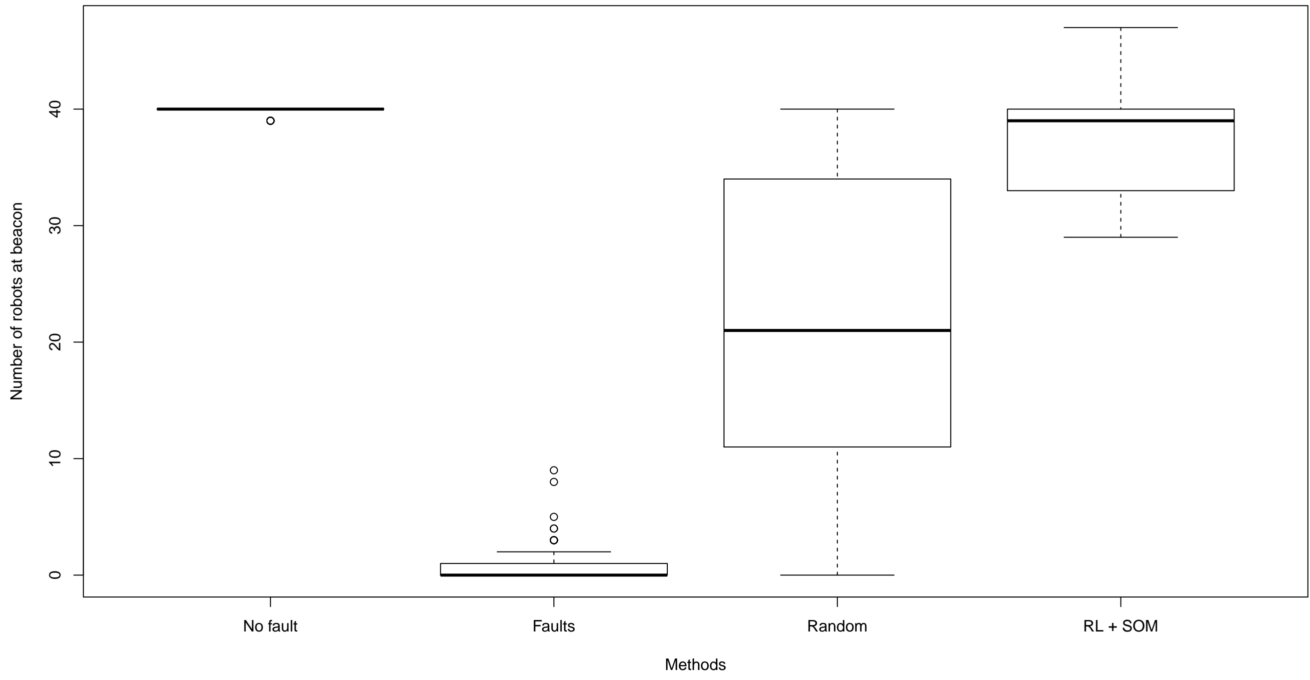


Figure 11.1: Collective Phototaxis: Motor failures

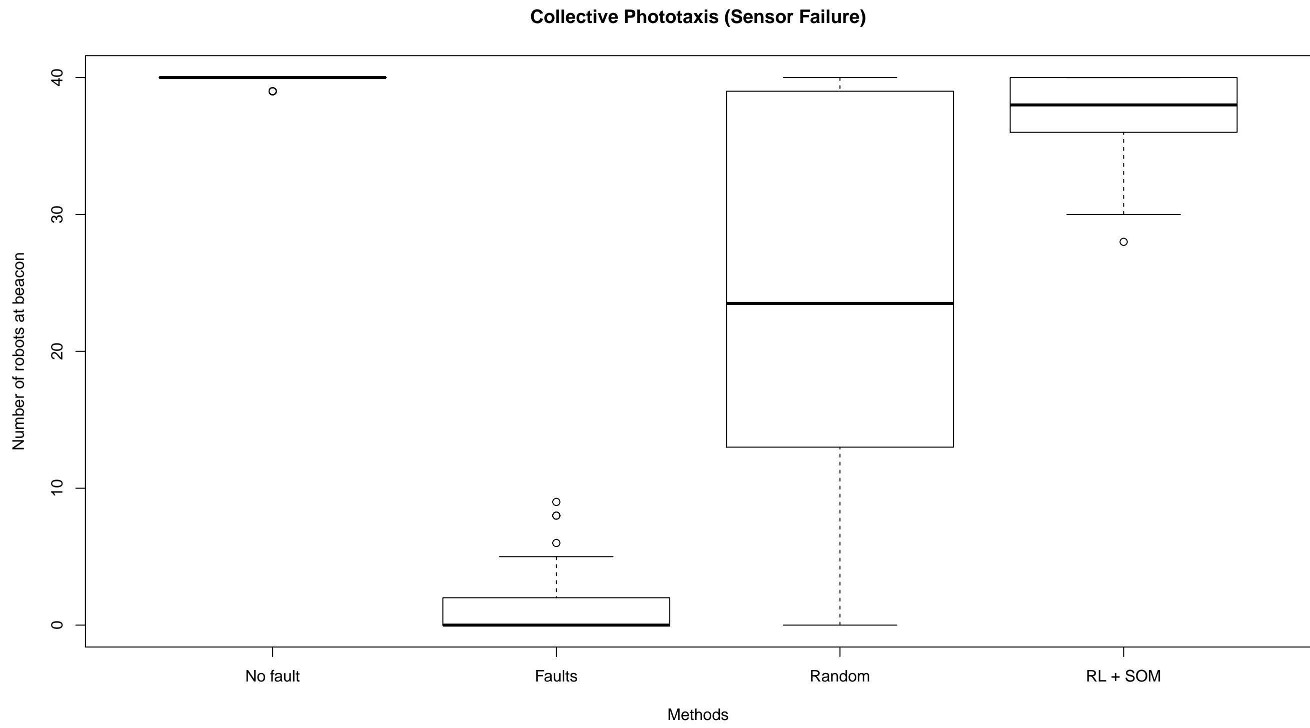


Figure 11.2: Collective Phototaxis: Communication sensor failures

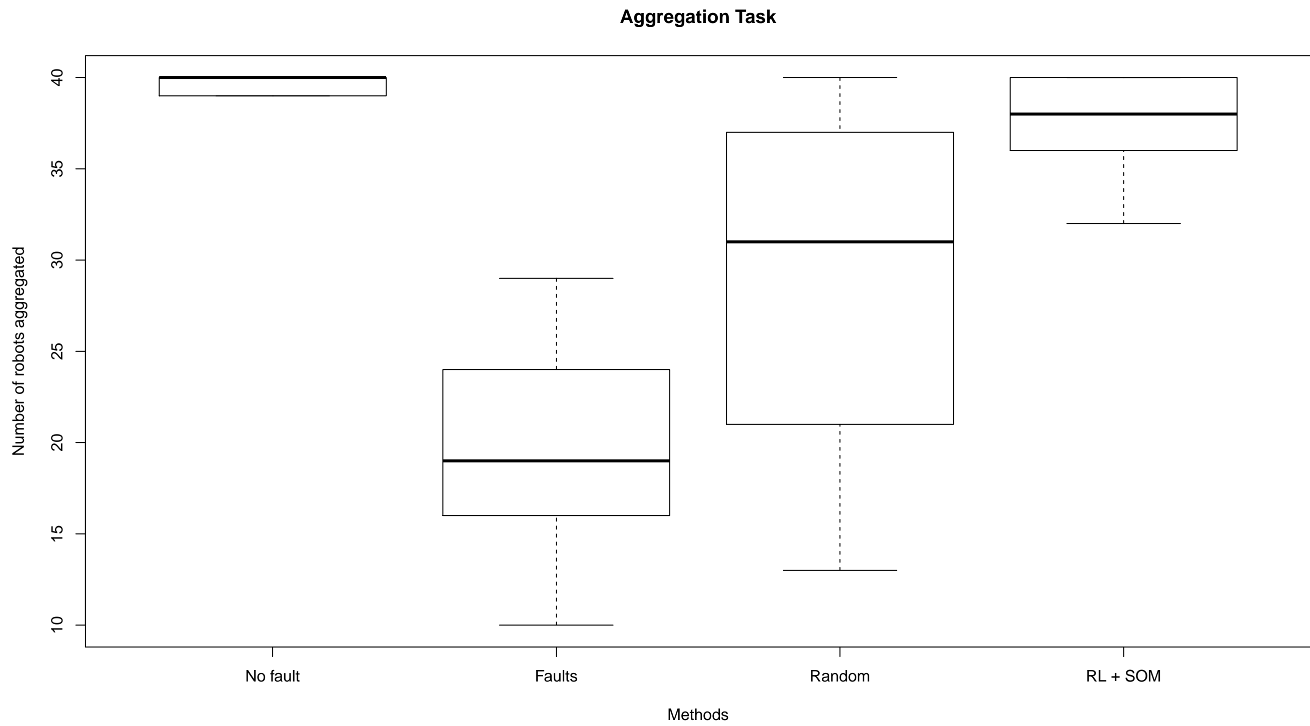


Figure 11.3: Aggregation: Communication sensor failures

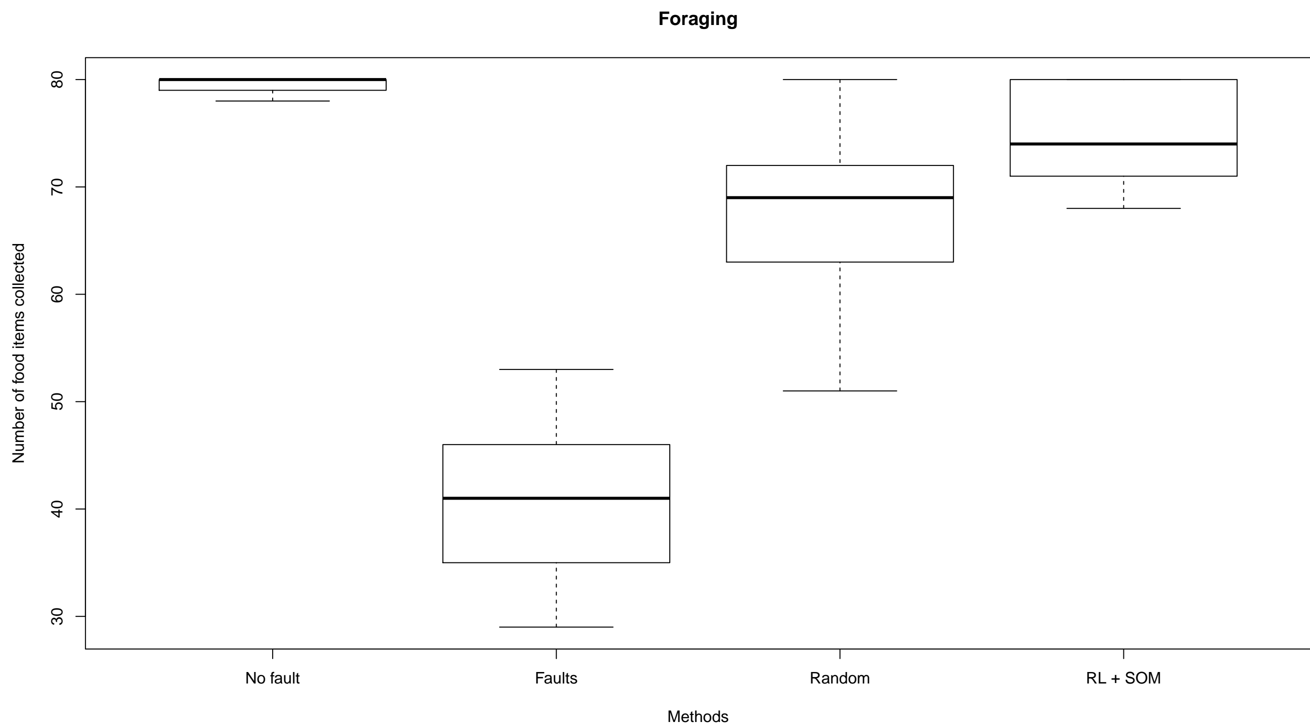


Figure 11.4: Foraging: Light Sensor (Number of Food Collected)

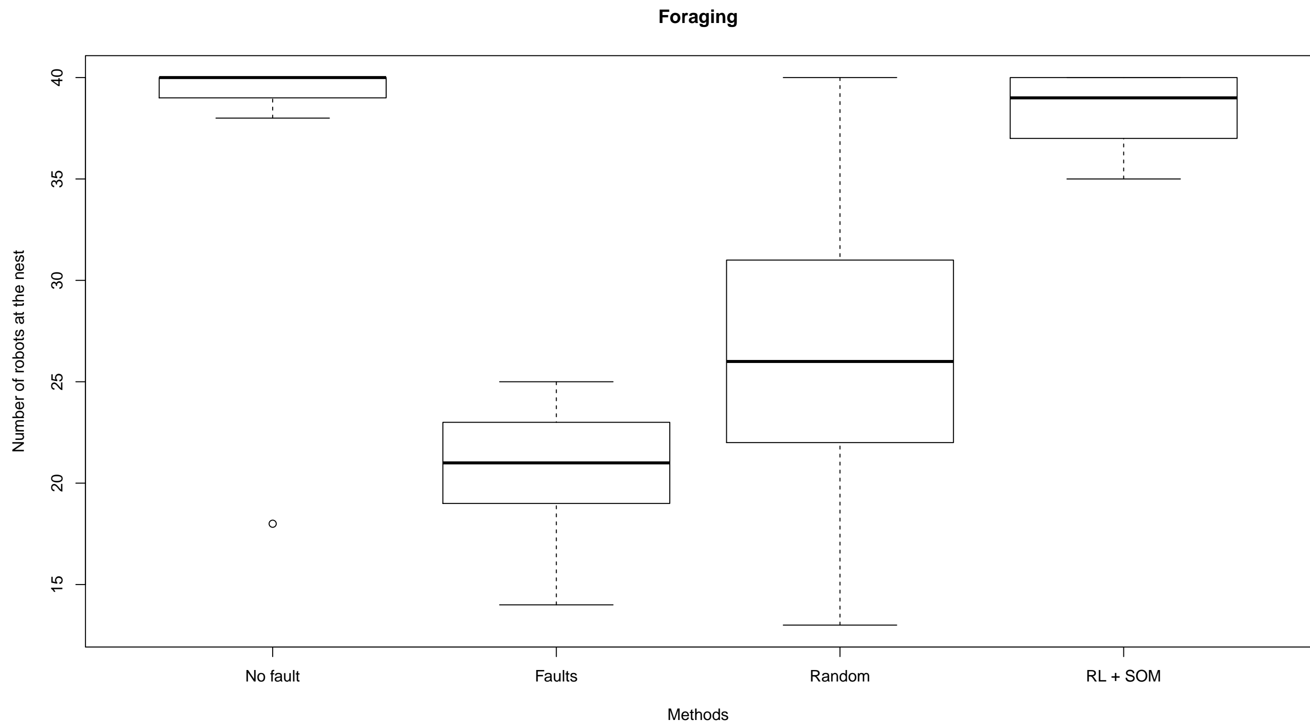


Figure 11.5: Foraging: Light Sensor (Number of Robots at Nest)