Towards a Swarm Robotic Approach for Cooperative Object Recognition

David King

A thesis submitted in partial fulfilment of the

requirements of Nottingham Trent University

for the degree of Doctor of Philosophy

September 2012

Copyright Notice

Dedicated to the memory of

Jean C. King, Mum.

Acknowledgements

It would not have possible for me to complete this thesis without the help and support of those around me. First of all my thanks go to my closest friends and family who have all been there at different times, in different ways. A special thanks to Charlotte Neep for listening to me talk for hours about all my ideas, thoughts and problems and to my Dad for his assistance in the final hours.

My respect and admiration go to my supervision team who have given me the freedom to follow the research that I wanted whilst giving me the guidance and pressure I required to keep on track. More specifically I am indebted to; Dr Philip Breedon, my Director of Studies, who always made himself available for countless discussions about the direction my research was going, as well as guiding me through the whole process; Prof Wayne Cranton, my second supervisor, whose questioning made me think critically about my thesis; and Dr Sarah Kettley and Ms Phillipa Marsh, who both took the role of my third supervisor at different times, for providing a different perspective on my research. I also thank Dr Mark Norman who gave critical guidance when it was needed most as well as technical support with my first steps into multi-agent programming and for this I am truly thankful.

My gratitude goes to Nottingham Trent University for its financial, academic and technical support and to its staff in the research office who helped me with endless queries and requests. Without the help of the Vice Chancellor's Bursary I would not have been able to support myself through the hours of work it took to complete my thesis. The opportunity provided by the University to teach and work with undergraduates has not only strengthened my abilities as an academic but also as kept me going when the bursary finally ran out.

Finally I would like to thank all my fellow postgraduate researchers who I have met over the years that I have been working on my thesis. They have provided me with many opportunities to discuss problems and made me feel part of something. From this group special acknowledgements go to; Guy Birkin without whom I would not have got so far in my research; Mathew Malpass for been there in the early years and giving me guidance and many Friday afternoons when they were needed; Julian Robinson for lunch time conversations; Alice Dallabona for drinks, cakes and company; and Rebecca Gamble for encouraging comments when the research process got the better of me.

Abstract

Social insects have inspired the behaviours of swarm robotic systems for the last 20 years. Interactions of the simple individuals in these swarms form solutions to relatively complex problems. A novel swarm robotic method is investigated for future robotic cooperative object recognition tasks. Previous multi-agent systems involve cameras and image analyses to identify objects. They cooperate only to improve their hypotheses of the shape's identity. The system proposed uses agents whose interactions with each other around the physical boundaries of the object's shape allow the distinguishing features found. The agents are a physical embodiment of the vision system, making them suitable for environments where it would not be possible to use a camera.

A Simplified Hexagonal Model was developed to simulate and examine the strategies. The hexagonal cells of which can be empty, contain an agent (hBot) or part of an object shape. Initially the hBots are required to identify the valid object shapes from a set of two types of known shapes. To do this the hBots change state when in contact with an object and when touching other hBots of the same state level, where some states are only achieved when neighbouring certain object shapes. The agents are oblivious, anonymous and homogeneous. They also do not know their position or orientation and cannot distinguish between object shapes alone due to their limited sensor range.

Further work increased the number of object shapes to provide a range of scenarios. In order to hypothesise the difficulty a swarm of hBots has distinguishing one object shape type from any other a system is devised to compare object shapes. Data-chains describe the object shapes, without orientation, by considering how many object cells the empty cells surrounding them are in contact with. Pairs of object shapes could then be analysed to determine their difference value from each other. These difference values correlate to a swarms difficulty in completing the specific scenarios.

Finally, a genetic algorithm (GA) was analysed as a method to determine the behaviours of the hBots different states. The GA is more efficient than both derived and randomly populated methods, showing that a GA can be used to train agents without first determining differences between the object shapes. These insights provide a significant contribution to knowledge through the object shape analyses method and the swarm robotic strategies which establish a unique foundation for further development of novel applications for both swarm robotic and cooperative object recognition research.

Glossary of Terms

GA: Genetic Algorithm

SHM: Simplified Hexagonal Model. The simulation model used to run the experiments for this thesis. Its basic design is a hexagonal arena consisting of hexagonal cells, which contain, either a hBot or object cell, or are considered empty.

hBot: Hexagonal agent used in the Simplified Hexagonal Model

Object Cells: Cells that are bound together to form an object shape.

Object Shape: A shape made from bound object cells for use in the Simplified Hexagonal Model. To be considered an object shape, the grouping of object cells must be solid. The smallest object shape is a single cell.

State: Describes the current state of a hBot, which is a reflection of its observations.

State-Behaviour: The behaviour or actions a hBot will take at any given state.

State-Level: The level which any given state is at. This gives an indication of how many hBots would be required to gain a specific state. At state-level 1, state-level 2 and state-level 3, it requires 1 hBot, 3 hBots and 5 hBots respectively.

Data-Chain: An array of numerical values representing an object shape without considering position or placement by describing the number of object cells in contact with each of the empty cells surrounding the object shape in clock-wise order. The data-chain is written as an array but has no beginning nor end.

Sub-Chain: A sub section of the data-chain of any given length.

Difference Value: A value between 0 and 1 which measures the perceived difference between two object shapes considering all the sub-chains at a specific length for a given data-chain.

Contents

Copyright Notice ii
Acknowledgements iv
Abstract v
Glossary of Terms vi

Chapter 1: Introduction 1
1.1 Background to Swarm Robotics 2
1.2 Research Questions 3
1.3 Summary of the Investigation 4
1.4 Contributions to Knowledge 8
1.5 Outline 8

Chapter 2: Multi-Agent Systems 10
2.1 A Comparison of Methods 10
2.1.1 Centralised and Decentralised Control 12
2.1.2 Communication 14
2.1.3 Identity and Anonymity 16
2.1.4 Homogeneity 17
2.1.5 Recall and Computation 19
2.1.6 Localisation 20
2.1.7 Synchronisation 21
2.2 Summary 22

Chapter 3: Swarm Robotics 24
3.1 Research Areas 24
3.3.1 Aggregation and Dispersion 24
3.3.2 Foraging 27
3.3.3 Self-Assembly and Connect Movement 28
3.3.4 Cooperative Transport 30
3.3.5 Pattern Formation 32
3.3.6 Self-Organised Construction 33
3.4 Discussion on Swarm Robotics Research Fields 36

3.4.1 Efficient Aiding and Necessary Cooperation 36
3.4.2 Identifying Objects 37
3.4.3 Inspirations from Nature 38
3.5 Summary 39

Chapter 4: Related Research 40
4.1 Cooperative Object Recognition 40
4.1.1 Efficient Cooperative Behaviours 40
4.1.1 Necessary Cooperative Behaviours 42
4.2 Genetic Algorithms for Multi-Agent Systems 44
4.2.1 Genetic Algorithms 44
4.2.2 Multi-Agent Systems Research 45
4.2.3 Genetic Representation of Agents 47
4.3 Physical Multi-Agent Platforms 50
4.3.1 E-Puck 50
4.3.2 S-Bot 51
4.3.3 Khepera 51
4.3.4 Miabot 52
4.3.5 Molecubes 52
4.3.6 ATRON 52
4.3.7 Catom 53
4.3.8 Miche 53
4.3.9 Robot Pebbles 53
4.3.10 Systems Review 54
4.4 Simulated Multi-Agent Platforms 55
4.4.1 Cellular Robot Simulations 56
4.4.2 Different Shaped Cells for Grid-Worlds 58
4.5 Summary 61

Chapter 5: Swarm Simulation Methodology 62
5.1 Cooperative Object Recognition Task 62
5.2 Choice of Platform 62
5.2.1 Processing Programming Language 63
5.3 Simplified Hexagonal Model 63
5.3.1 The Arena 63

5.3.2 Object Shapes 64
5.4 The hBots 66
5.4.1 Sensor Capability 66
5.4.2 Communication 67
5.4.3 Random Movement 67
5.4.4 Computation and Recall 68
5.5 hBot Cooperative Object Recognition 68
5.5.1 State-Relationships 69
5.5.2 State-Behaviours 70
5.5.3 Possible State Neighbours 71
5.6 The System 73
5.6.1 Experimental Data 75
5.7 Limitations of the Platform 75
5.7.1 Grid-World 76
5.7.2 The Requirement to Cooperate 76
5.7.3 Perfect Sensors and Communication 77
5.7.4 Neighbouring Up to Three Object Cells 77
5.7.5 Knowledge Equivalent to Five hBots 77
5.7.6 Symmetrical Object Shapes 78
5.7.7 Synchronised 78
5.7.8 Controlled Movement 79
5.8 Training the Swarm 79
5.9 Summary 79

Chapter 6: Initial Research 80

6.1 Methodology 80
6.1.1 The Arena 80
6.1.2 Removing Object Shapes 81
6.1.3 The hBots 81
6.1.4 The Variables 84
6.2 Results 84
6.2.1 Task Completion 84
6.2.2 Swarm Size 86
6.2.3 Energy Consumption 88
6.2.4 Probability of Movement 90

6.2.5 Hexagons and Triangles 91
6.3 Further Investigations 91
6.4 Summary 92

Chapter 7: Object Shapes 93
7.1 Possible Object Shapes 93
7.2 Systematically Creating the Object Shapes 94
7.2.1 Spiral Location 94
7.2.2 Producing the object shapes 96
7.2.3 Checking the Object Shape is Authentic 97
7.3 Object Shapes as Binary Images 99
7.3.1 Describing Object Shapes without Location or Rotation 101
7.4 From Object Shapes to Data-Chains 104
7.4.1 Simple and Complex Object Shapes 106
7.5 From Data-Chains to Object Shapes 107
7.5.1 Tracing a Data-Chain 107
7.5.2 Tracing Complex Object Shapes 110
7.5.3 Single Data-Chains with Multiple Object Shapes 113
7.6 Types of Object Shapes Found 114
7.6.1 Computational Complexity 114
7.7 Summary 115

Chapter 8: Comparing Object Shapes 117
8.1 General Method for Comparing the Object Shapes 117
8.2 Results from General Method of Object Shape Comparison 119
8.2.1 Special Instances of Object Shapes 123
8.3 Differentiating Object Shapes for hBots 126
8.3.1 Calculating Difference Values for hBots 126
8.3.2 The Difference Values 128
8.4 Comparison to hBot Experiments 130
8.4.1 The Difference Value and the Number of Time-Steps 133
8.5 Summary 135

Chapter 9: Training Methodologies 136
9.1 The Object Shape Recognition Scenarios 136
9.1.1 Scenario Selection 136
9.1.2 The Cooperative Object Recognition Task used in Training 137
9.1.3 The Training Methods 137
9.2 Genetic Algorithm Determined State-Behaviours 140
9.2.1 Representation of Candidate Solutions 140
9.2.2 Recombination 141
9.2.3 Mutation Operator 143
9.2.4 Population Model 144
9.2.5 Parent Selection 144
9.2.7 Initialisation and Termination Condition 144
9.3 Randomly Determined State-Behaviours 146
9.4 Comparison of Methods 146
9.5 Summary 147

Chapter 10: Analyses of Training Methods 149
10.1 Results of the Genetic Algorithm 149
10.1.1 Comparison to Predicted Difficulty Metric of Scenarios 153
10.2 The Results for the Random Search Method 159
10.2.1 Comparison to Predicated Difficulty Metric of Scenario 161
10.3 Correlation of Difficulty Measurements to Scenario Difficulty 165
10.4 Comparison of Fittest Solutions 167
10.4.1 Testing the Fittest Solutions 171
10.4.2 Assessing the Suitable Solutions 172
10.5 Summary 173

Chapter 11: Discussion 175
11.1 Task and Application 175
11.2 A Suitable Task for a Swarm Robotic Approach 176
11.3 The Outcomes 177
11.3.1 The State-Behaviours of hBot Swarms 177
11.3.2 The Training of hBots 178
11.3.3 The Metric of Scenario Difficulty 179

11.4 Future Work 182
11.4.1 Moving to a Physical Platform 183
11.5 Closing Statement 185

References 186

Appendices 206
Appendix A: The Simplified Hexagonal Model for Initial Investigation 206
Appendix B: Initial Experiment Results 219
B.1: Number of Hexagonal Object Shapes Removed when Valid 220
B.2: Number of Triangular Object Shapes Removed when Valid 222
B.3: Time-steps Required to Remove Three Valid Hexagonal Object Shapes 224
B.4: Time-steps Required to Remove Three Valid Triangular Object Shapes 226
B.5: Energy Consumed to Remove Three Valid Hexagonal Object Shapes 228
B.6: Energy Consumed to Remove Three Valid Triangular Object Shapes 230
Appendix C: Object Shape Creator and Data-Chain Inspector 232
Appendix D: Possible States for hBots and State-Relationships 247
Appendix E: Advance SHM Program with GA 251

Chapter 1: Introduction

Cooperative object recognition is an area of research concerned with the identification of objects by a group of robotic agents who either are incapable of identifying an object alone or who increase the efficiency of object recognition through parallel cooperation. This research investigates the potential for training a group of agents utilising a swarm robotics approach to distinguish differences between objects through their shape. The aim of which is to provide future physical swarm robotic systems with strategies for cooperative object recognition, where rather than camera and vision recognition being used the agents interact with each other around the object's surface in order to analyses the shape to find features that distinguish them from one another. A proposed method is for these distinguishing features to be learnt by the swarm using a genetic algorithm (GA), the efficiency of which will be tested. The agents will have the following characteristics:

> Homogeneous, anonymous finite-state machines, incapable of remembering previous events other than their current state and are incapable of individual mapping the shape of an object alone; without a common coordinate system; capable of determining how concave or convex a local part of the shape they are in contact with is, with three degrees of accuracy; and also perceiving the current state of any other agent they are in physical contact with.

This approach is unique to cooperative object recognition where swarm robotic techniques have not been utilised in this way before. In order to distinguish between the objects the agents must cooperate with each other, sharing limited local information about the shapes convexity or concavity by changing state. As the agents are aware of the states of the other agents they neighbour a continuous feedback of information can occur allowing the agents to gain more information and find distinguishing features in the objects.

By using individually less capable agents who alone cannot distinguishing between the objects allows the size of the future robotic systems they are implemented on to be reduced as they require less hardware. Possible applications include the identification of cancer growths, or viruses in a body through shape recognition, if the scale of the robots can be sufficiently reduced. The robotic agents could then either destroy the entities or highlight their location, providing temporary or on-going medical care.

1.1 Background to Swarm Robotics

Swarm robotics is a strategy for dealing with multi-agent control that is influenced by research into understanding social insect behaviours (Şahin, 2005). Where individuals in multi-agent systems collaborate, in a predetermined manner, to complete tasks by working together in series or parallel, swarm robotics uses many agents whose interactions with each other cause the solution for the task to emerge. Relative to social insects robots are simpler, for example they cannot reproduce and they do not grow. However, it is not necessarily the complete natural system that swarm robotic research seeks to mimic. The emergence of the solution is key to swarm robotics and takes its inspiration from social insects such as ants, bees and termites (Şahin, 2005). In these insect communities there is no leader, no commander, nor ruling queen neither is there an overruling plan, scheme nor blueprint for the individuals to follow. However, these insects are capable of completing tasks that seem relatively complex, when compared to the individuals within them, through decentralised control. For example ants build nests (Franks and Deneubourg, 1997), can find the shortest routes using pheromone trails (Goss et al., 1989), can coordinate to move objects too large for a single ant to move on its own (Franks, 1986), they can also organise their dead into clusters (Diez et al., 2011), sort their brood (Sendova-Franks , 2004) and organise them all relative to each other and their nest (Deneubourg et al., 1991). A more general overview of the behaviours that control social insects is available, Theraulaz et al. (2003).

Without a leader or specific plans how is it that these simple agents are capable of such tasks. It is partly their simplicity that makes them capable of decentralised problem solving. Each agent only needs to interact with their immediate environment and is ignorant of anything that is happening elsewhere. Their reaction to this local environment then changes the environment around them.

As an example from nature, ants as a group can find and gather food, and are even capable of responding to the relative quality of those sources (Jackson and Châline, 2007). Individual ants do not have this capability. Each ant responds to its local surroundings by releasing pheromones which other ants react to. When enough of these interactions between the individual ants and their environments occur then the solution to this relatively complex problem of food sourcing emerges, guiding the ants to the highest quality source of food. It is this type of simplistic but naturally robust, scalable and flexible system that swarm roboticists dealing with multi-agent systems are using to

control vast numbers of agents without adding further complexity to the agents as the scale of the problem increases.

Swarm robotics research covers different attributes of behaviour, many of which mimic those seen in the natural world. These range from dispersion (Ludwig and Gini, 2006; Hsiang et al., 2004) and aggregation (Payton et al., 2001; Soysal and Şahin, 2005), through foraging (Winfield, 2009; Kriefer, Billeter and Keller, 2000; Shell and Matarić, 2006) and pattern formation (Suzuki and Yamashita 1999; Défago and Konagaya 2002), to self-assembly (Groß et al. 2006; Tuci et al., 2006) and self-organised construction (Wawerla, Sujhatme and Matarić, 2002; Theraulaz and Bonabeau, 1995a). There is an overview of the social insect influence on swarm robotic control by Kube and Zhang (1994).

By considering the strengths of swarm robotics over different multi-agent control systems many real world applications can be found. The applications often considered include, search and rescue (Baxter et al., 2006; Payton, Estkowski, and Howard, 2003), and for use in hazardous areas such as mine-fields (Cassinis et al., 1999) as it allows for less human contact with potential dangers. What these domains have in common is their requirement for a robust system that can dynamically change with the environment as it changes.

1.2 Research Questions

- How capable are different swarms of agents at discerning two object shapes from each other through necessary cooperative object recognition.
- How do the similarities of the object shapes determine the difficulty for the swarm of agents to distinguish between them?
- Are a swarm of agents capable of learning to discern two object shapes from each other through a GA?

1.3 Summary of the Investigation

The focus of this research will be on the strategies involved in controlling the swarm through the actions of the agents. Once these strategies are discovered a control design template would exist for future research with physical robotic systems. To undertake this investigation a simulation was developed using a hexagonal lattice. Before the use of physical hardware initial swarm robotics research was also carried out on square lattices (Beni, 1998). The simulation model developed for this research is the Simplified Hexagonal Model (SHM) which allows strategies for swarm systems to be tested and advanced in order to inform approaches to implement solutions on a physical platform (King and Breedon 2011a; 2011b).

The SHM is built on a hexagonal lattice to allow the agents to have a better approximation of free movement when compared to a square lattice, making this system an appropriate way to develop strategies for a new area of research within the field. Each cell that constructs the arena space is either considered empty, containing an agent (hBot), or part of an object shape. The use of a discrete lattice as opposed to a space allowing continuous movement provides for a clear distinction of whether or not hBots are neighbouring each other or neighbour an object shape. This choice also restricts the number of relative positions the hBots and object cells can be in relation to each other. However, this is at the cost of reducing the similarities with a physical system, which are more accurately portrayed with continuous movement. The SHM could be developed for different research enquiries by changing the attributes of the agents to suit the task methodology.

In order for the swarm of hBots to recognise an object shape it is necessary that the states of the agents in the negative space of the object shape relate to the object shape. In order to determine how this relationship may be created an investigation into the object shapes produced on a hexagonal grid is needed. This investigation involves developing a system to produce different object shapes in a logical manner. Once a number of shapes have been produced it is possible to analyse their relative differences and how the hBots interact with them. Part of the problem for the system producing these object shapes is avoiding congruent shapes being produced. To deal with this problem a study into binary image storage and transfer methods and techniques is carried out. This study provides an insight into the methods used for reducing the information needed to draw an image by considering the relationship of the shapes

boundary pixels, known as chain-coding (Katsaggelos et al., 1998). However, there are drawbacks to this method that need to be overcome. The major issues are that information about location and orientation are not required in the object shape discovery process. In this process any object shapes that are identical except from their positions and rotation are considered to be the same type of shape. These issues are resolved by considering the description of the boundary as a loop of information, with no beginning or end, termed the object shape's data-chain. Each shape has its own data-chain and two data-chains can be compared to see if two shapes are identical or not. There is an exception to this, outside the remit of the research, where object shapes have the same data-chain in specific cases at the cusp of the hBots ability. The system is able to compare two object shapes by comparing their data-chains and hence check if the object shape had already been added to the system.

The hBots, in their locale, use the same information that is used to create the data-chains as they, the hBots, inhabit the same negative space around the object shapes. However, as an individual they do not have access to all the information, only information from their neighbouring cells. The amount of information they can glean from their neighbours determines their current state-level. An individual hBot that is next to an object knows that it is next to either one, two or three object cells of that object shape, and this knowledge is a low state-level. In this research, cases where hBots could neighbour four or more cells are not currently considered. This choice reduces the number of total possible states at all state-levels and removes any case where a hBot could neighbour more than two other hBots in contact with the same shape. This reduces the number and type of object shapes that can be considered. It is a requirement for the project that the object shapes within the arena are not in contact with each other, as this would make them appear as a single object shape with a different shape boundary.

As many object shapes have similar features locally a hBot cannot determine which specific type of object shape they are near from this low state-level. Local object information is not the only information available to the hBots. The hBots can determine the states of their immediate neighbouring hBots, which will also hold information about the object shape in their location. By changing their states based on their neighbours' states the hBots can increase their state-levels gathering further knowledge about the object shape. So despite only having a limited sensor range hBots

can cooperate to build their state-levels allowing them in turn to identify the different object shapes. One limiting factor of the hBots is the number of states and state-levels required in relationship to the size of the object shapes being compared. As the size of the object shapes increases it may require knowledge of a higher region of the boundary before a distinguishing feature is found.

From the comparison of the data-chains of the object shapes it is possible to hypothesise the number of time-steps it would take a swarm of hBots to differentiate one object shape from another. The more the object shapes have in common the more cooperating agents, local to that object shape, it requires to distinguish them. However, knowing how difficult the task may be and having the hBots change states according to their perceived surroundings does not solve the problem of how the hBots should react at a given state.

The solution to this problem is determined by the hBots state-behaviour, a rule table informing them of what actions to take based on their current state. There are a number of possible actions the hBots could take at a given time: move at random around the arena; remain neighbouring an object whilst trying to cooperatively identify it; and collect or remove an object shape that it neighbours. To determine what these state-behaviours are requires a knowledge of the different object shapes' data-chains, what states are achievable by the hBots when reacting with those object shapes, and the differences between the two. For example, given two object shapes, if a state is achievable for one object shape but not the other then that hBot would know it is next to that object shape, however if the state is achievable at both object shapes the hBot does not know whether it is at one object shape or the other.

What is more interesting however, is measuring the hBots capability to learn what actions to take when given training tasks and a fitness value based on their performance. To investigate this idea GA is developed that allows task specific solutions to the state-behaviours to evolve. GA follow Darwin's evolution principle where the members of a population most suited to an environment have a higher chance of producing offspring which share similar traits to their parents (Eiben and Smith, 2007) . They have been used to solve many different types of problem, sewer network design (Afshar, 2012); designing a concert hall for optimal acoustics (Sato et al. 2002); the placement of wind turbines (Grady, Hussaini and Abdullah, 2005) and to aid in stock trading (Kuo, Chen and Hwang, 2001).

There are considerations to be made when using a GA to solve all or part of a swarm robotic system. As individuals in a swarm may act differently each time a test scenario is run to get an accurate representation of performance requires taking average results over a number of trials. The repetition of trials increases the run time of the GA. If the run time becomes so vast and the solution can be found in a simpler manner the use of a GA is inefficient. It is for this reason that a comparison is made with a randomly produced state-behaviour solutions over a range of task scenarios with varying perceived complexity as well as a systematically derived generic set of state-behaviours.

The use of a GA or a randomly derived approach would result in a system that is more adaptable to different environments. In the envisioned final system there would be no need for a user to analyse the difference between object shapes, they would only need to provide feedback on how well the swarm performed. There are other methods available for training a multi-robotic system, such as reinforcement learning (Sutton and Barto, 1998) as used by Matarić (1997), however these are not considered here due to time constraints but should be considered in future research for comparison.

Although determining the real world applications for a swarm robotic approach to the cooperative object recognition task is not the aim of the research there are areas where the system would be ideally suited. With the ever decreasing scale of technology and robotics and by eliminating the need for complex vision sensors, this type of cooperative object recognition would be suitable for nano-scale applications. At this scale agents of a swarm could interact with an object in a physical way, contouring to its shape. There is no need for the agents to understand the object directly but merely the relationships it has with the other robots that surround the object. This capability to understand the placement of neighbouring robots and the states they are in could be directly implemented into the robots. Utilising this system at this scale a swarm of robots could be trained to identify entities inside a human body, such as cancers and viruses, which would aid medical practices.

1.4 Contributions to Knowledge

The main contributions to knowledge are as follows:

- A strategy for distinguishing between two object shapes using a swarm of agents that interact both physically with the object shape's boundary and through state changes with their neighbouring agents. Utilising agents as described in section 1.0.

- A method for producing and describing object shapes constructed on a hexagonal lattice, termed a data-chain. This description of the object shapes does not considering the position and rotation of the object shape and allows analyses of pairs of object shapes to identify their difference value. The difference value is dependent on the length of sub-chain that is considered which reflects the way the hBot agents interact with each other and the object shapes. This method could be applicable to lattices of other shapes.

- An analysis of the capabilities of the difference values and other metrics found between a pair of object shapes to predict the difficulty that a swarm of agents would have in distinguishing those two object shapes from each other. For each pair of object shapes the behaviours of the hBots, for each state, were determined with three methods; a generic baseline method, a randomly produced method, and that of a GA.

1.5 Outline

The proceeding thesis is set out as follows: Chapter 2 contains an overview of the different methods inherent to multi-agent and multi-robot systems; Chapter 3 considers and discusses the current research areas in swarm robotics, including a discussion on the types of cooperation; A review of relative research in cooperative object recognition and the use of genetic algorithms for multi-agent systems is included in Chapter 4, as well as a review of available multi-agent systems for experimentation; Chapter 5 provides a methodology for the current research project; The initial investigation is covered in Chapter 6 showing that the SHM is a suitable model for testing multi-agent systems with object shapes, and provides guidance for the experimental set-up of the final investigation; Chapters 7 and 8 make up the second part of the investigation. Chapter 7 details how different object shapes are constructed on a hexagonal lattice and

how these shapes can be differentiated from each other giving a numerical value of difference. The method for calculating the difference value of a pair of object shapes is modified in Chapter 8 to consider how the hBots view and interact with the object shapes. The difference values found between the object shapes are tested by measuring how long it takes a swarm of hBots to distinguish one shape from another. In Chapter 9 the GA method is described and in Chapter 10 the GA is tested against a random method for determining the state rule behaviours and a generic method. Both the GA method and the random method scenarios are measured for difficulty to find potential solutions against a range of predictions provided by the object shape difference analyses from the second part of the investigation. Finally Chapter 11 provides an overall discussion of the research investigation providing an overview of the outcomes and the future work that could be considered.

Chapter 2: Multi-Agent Systems

This chapter compares the different methods used to control multi-agent and multi-robot systems. The overview specifically considers the potential differences of the individual agents and the effect that these may have on the group's behaviours and its capabilities. A discussion on the influence of natural systems on multi-agent systems and the identification of two types of cooperation are also included.

2.1 A Comparison of Methods

A multi-agent system is one where a group of agents work together to complete a task. The exact method used by the agents or robots can vary depending on the goals of the research. These goals are often trying to find a balance between producing a system that is capable of completing a task and that of simplifying the design of the individual robotic agents. Shiloni, Agmon and Aminikia (2011) identify two different types of robotic agents, which they describe as 'ants' and 'elephants', highlighting the different design choices for multi-agent systems. Their description of robot ants and robot elephants identify two extremes of individual agent capability; where the robot ants are considered to be computationally simpler than robot elephants in a number of key features, making the robot ants more in-line with agents seen in swarm robotics. The ability of the individual agents has an overall effect on the capability of the group and the methods that are required to have those groups achieve any specific task. A comparison of the robotic ants and elephants' capabilities are listed in Table 2.1 showing the potential variance in any specific multi-agent group.

	'Ant' Robots:	'Elephant' Robots:
Instruction Set	Can • *Move* in all directions. • *Sense* a limited radius around them. • *Read* and *write* arbitrary levels of multiple pheromone types.	Can • *Move* in all directions. • *Sense* a limited radius around them.
Memory and Computation	Have a limited recollection compared to the size of the work area, allowing them to remember only a constant number of previous moves. From a computational point of view, are finite state machines.	Have unbounded memory. From a computational point of view, are Turing machines.
Communication	Have an unlimited amount of pheromones, which are essentially traces that can be read from and be written to a space. The pheromones do not evaporate by themselves but can be erased and re-written.	Have reliable, instantaneous communications to all others.
Localisation	Have no means of localization.	Can typically perfectly localize themselves on a shared coordinate system. However there is a tested variant where they cannot localize within a global grid.
Anonymity	Are anonymous, and therefore cannot identify each other.	Have distinct identities, and all know of each other.
Homogeneity	Are homogenous; they all have the same capabilities, and run the same algorithm.	Are homogenous in the sense that they all have the same capabilities and run the same algorithm.
Centralisation	Work in a decentralized fashion.	Work in a centralized fashion.

Table 2.1: The differences in capability of two types of robot which both have the same sensing capabilities. Adapted from Shiloni, Agmon and Aminika (2011).

The results from Shiloni, Agmon and Aminika's (2011) experiment showed that "given a large enough space and infinite amount of pheromones, a single ant can simulate any task done by a single elephant that has no localization abilities". However, "there exists some problems that can be solved by *N* elephants, but not with *N* ants" (Shiloni, Agmon and Aminika, 2011, p. 5786). Their research demonstrates that there are some limitations to what simplistic ant type agents can do and often that a more complex

elephant type robot will be more capable. Altshuler, Wagner and Bruckstein (2009) argue in opposition to using complex agents, especially where the original system is designed for simplistic agents cooperating in a swarm robotic approach. The results found for a graph exploration problem were surprising. The experiment showed that upgraded robots performed worse at the exploration task, taking more time than an unenhanced group. In other words their conclusion was that starting with the ant type agent for a swarm solution and developing them into elephants can make the solution less efficient. To gain a greater insight of suitable characteristics of a multi-agent system for cooperative object recognition task a more detailed analyses of the different aspects are examined here.

2.1.1 Centralised and Decentralised Control

A multi-agent system's control can be centralised or decentralised, describing where the main planning is undertaken for completing a given task. For a multi-agent system with centralised control there is usually one, but sometimes more, main controllers of the group. These main controllers gather information from the sub-level agents within their group and then issue commands, which spread back out into the group. In this type of system there is a definite hierarchy, information is sent up the hierarchy and commands are sent back down. The information sent upwards gives the main task control unit, be that a key mobile member of the group or a fixed post, an overview of events across the arena. Using this information the central control system decides which tasks need completing and by whom. Commands are then sent to the lower level agents, which perform those tasks as required.

This type of system can be extremely efficient where all elements of the system are known including the task that needs completing. For example Sanches and Latombe, (2002) find that for the situation of multi-robot welding station that a centralised control method is preferable. The decentralised control strategies were found to be less reliable and only marginally quicker in a minority of successfully completed tasks. It should be noted that for this situation there were only two, four or six robots, which is a relatively small number especially when compared to typical swarm robotic systems.

Problems with centralised control could arise when the systems that are being monitored or controlled by the robotic agents experience unforeseen changes or any additions are made to the system, as they are designed for dealing with specific tasks. These factors could potentially require a complete overhaul of the system. There are

also potential issues with bottle-necking. The capabilities of the centralised controller is finite, if there is too much information to process the speed that the system can react is reduced. For example adding more robotic agents to the system will increase the amount of data that needs to be both received and analysed.

A centralised system requires strong communication between the central control unit and all the other acting agents. In many practical settings there may not be unlimited communications between the control centre and the robotic agents (Clark, Rock and Latombe, 2003). Without this level of communication the system can break down. The solution offered to these problems for multi-agent system control is to use distributed or decentralised control.

A decentralised control system removes the single point of control and has the interactions between the agents locally solve the problem. With this type of control system each agent in the group reacts only to what they perceive in their immediate surroundings. This reaction can either be to something in their environment or to information from another robot. Unlike the centralised control system there is no hierarchy within the group; all members are equally important.

Laengle and Lueth (1994) make the distinction between three types of control for complex systems that consist of several executive subsystems or agents. They do this to clarify the difference between distributed and decentralised control, which is often used interchangeably. The three systems are:

- **Centralized Systems:** A decision is made in a central mechanism and transmitted to executive Components.
- **Distributed Systems:** The decision is made by a negotiation process between the executive components and executed by them.
- **Decentralized Systems:** Each executive component makes its own decisions and executes only these decisions.

This description clearly separates the notions of decentralised and distributed control systems, suggesting that a decentralised system does not have any form of communication. However, Lagoudakis (2005, p. 1) states that "[e]ven in decentralized multi-robot coordination, some information exchange is necessary" in order to aid decision making but also suggests that communication should be kept to a minimum.

The issue is in defining communication and negotiation. Even though there may not be explicit communication between one agent and another, there can still be implicit communication. The act of the agents changing their environment through the movement of objects or the movement of agents themselves alters other agents' perception of the environment they are in and is therefore a type of negotiation. For example, a task or sub-task could be completed or partly completed and the task itself becomes the method of negotiation of what has and what has not been done. Therefore it could be argued that without any form of communication either direct or indirect Laengle and Lueth's (1994) model of a decentralised system is equivalent to multiple agents working individually in parallel, without interacting with the other agents or their changes to the environment. In this type of system it would be impossible for the agents to cooperate intelligently with each other. In the case of swarm robotics there is no need for this precise distinction.

2.1.2 Communication

Communication between robots can be direct or indirect. Direct communication involves a robot giving another specific robot some amount of information or a command. The advantage of direct communication is the intent to communicate with a specific robot. Robot A intends to give robot B a command or some information, if the two robots are in communication range, the information can be sent by A, received by B and also confirmed to be received by B. Therefore the task of communication can be said to be complete. It is slightly more complicated than that as there are some issues with noise and interference, which need to be considered. However, in a perfect or at least robust system robot A has told robot B to do something and both robots A and B know this.

The problem with this method is the need to communicate with a specific robot or in some cases the central control system. In an unknown environment there is no guarantee that the robots will be in contact with the member they are trying to communicate with. Also there is a potential problem with scalability; as more and more robots are added to the group the need for the robots to communicate and keep track of whom they have communicated with becomes increasingly complicated. These issues are addressed with indirect communication.

Indirect communication avoids this problem of scalability. Here the agents of the group can communicate either through their environment or via implicit electronic forms of

communication. This type of communication is closely linked to the use of anonymous robots, discussed in section 2.1.3.

In the first instance of implicit electronic communication each member of the group can express what it is doing or sensing. This local broadcast is achieved by changing an outgoing signal that is not directed to any specific agent but to any agents that are within range of the signal. Rather than the natural pheromones, as seen in ants, electronic forms of communication are used in swarm robotics such as; radio frequency identities utilising tags and readers (Herianto, Sakaakibara and Kurabayashi, 2007), infrared propagated by line-of-sight (Payton et al., 2001) changing light arrays (Nouyan et al., 2006). This form of communication gives the advantage of not having to have direct links and channels between all the robots in the group but at the cost of not knowing if the broadcast was received by any other agents. An issue with both this form of indirect communication and more so direct communication is having to have the agents within communication range of each other.

Having the swarm communicate through their environments alleviates this problem. If the agents are interacting with the environment then the environment can be used to store information about how much interaction has taken place and where it has taken place. This is known as stigmergy. The term stigmergy was first used by Pierre-Paul Grassé in 1959 when discussing the behaviours in termites (Theraulaz and Bonabeau, 1999).

"[Stigmergy] is based on the use of the environment as a medium of inscription of past behaviours effects, to influence the future ones. This mechanism defines what is called a self-catalytic process, that is the more a process occurs, the more it has chance to occur in the future. More generally, this mechanism shows how simple systems can produce a wide range of more complex coordinated behaviours, simply by exploiting the influence of the environment." (Serugendo et al., 2004, p. 5)

Ricci et al. (2006, p. 124) states that "stigmergy is mostly used as the source of simple yet effective coordination metaphors and mechanisms for robust and reliable systems in unpredictable settings." The environment acts as both a memory and communication system for the swarm. This allows the agents to communicate with each other without necessarily being in the same space at the same time. This is one of the major benefits of a stigmergic systems over any form of communication between two robots direct or

indirect. Beckers, Holland and Deneubourg (1994) successfully used stigmergy in a task where different size groups (1-5) of robots had to collect 81 pucks into a single pile.

An interesting example of electronic stigmergy through electronic pheromones can be found in the research by Susnea, Vasiliu and Filipescu (2008). They suggest that the low cost of RFID (Radio frequency identification) devices would allow them to be placed in abundance into an area of interest. The RFID would then act as a digital storage device for electronic pheromone in the environment and allow robots to directly mimic the behaviour of ant pheromones for route planning and shortest path problems.

Werfel et al. (2006, p. 2794) discuss the benefits of 'extended stigmergy' where they increased the capability of the building materials in a construction task to hold information about the structure. They state "[w]hile building structures from inert, indistinguishable blocks is possible, incorporating communication abilities into the blocks brings considerable benefits in speed and robustness."

This demonstrates the potential power of stigmergy to ignore communication issues that are inherent in direct communication and indirect communication between agents. However, this type of behaviour may not be suitable for all applications where communication is required for example if there is a high level of interaction between the agents themselves an indirect method of communication between the robots may be superior and where there are only a few robots completing complex tasks direct communication may be more appropriate.

2.1.3 Identity and Anonymity

Robots and agents in multi-robot tasks can either be anonymous or identifiable. An identifiable robot has its own individual ID that it is aware of and that the other robots in the group are aware of. This can be useful in completing a task as specific robots can be commanded to do certain sub-tasks or jobs and this way the work can be distributed efficiently. There is a potential risk with this type of system in a situation where a specific robot or agent is required or told to do a job and that robot is not accessible.

There are numerous examples of multi-robot systems that use robots with individual identification for different problems. For example:

- A task where the robots need to move to a goal whilst remaining in formation and avoiding obstacles (Balch and Arkin, 1995). In this case the ID of the

robots determine the position they take in each of the possible formations, therefore the robots are acutely aware of their relative positions to each other.

- As a diagnostic system where each robot continually gives out an 'I am alive' message along with its ID. If the robot fails to send out the message the rest of the team re-evaluates the task (Schneider-Fontán and Matarić, 1998). Utilising IDs allows the robots to determine how many robots are currently active without which knowledge they would not be able to revaluate the task successfully.

- Each of three robots are given a different colour collar to distinguish themselves whilst cooperatively moving an object (Spletzer et al., 2001). The unique IDs allowed the robots to cooperate to localize themselves and also assign a leader to follow to remain in formation.

In contrast to these examples the robots can be anonymous. Anonymous robots do not have individual identification for the purpose of task solving. Batalin and Sukhatme (2002) found that simple anonymous robots slightly outperform more complicated robots with ephemeral identification techniques when deploying a mobile sensor network into an unknown environment. Anonymous agents are also used in Défago and Konagaya's (2002) research into circle formation.

The advantage of this type of anonymous system is that there is no need to communicate to a specific robot or change the information that the robots use based on how many robots are currently trying to complete the task. This idea ties into homogeneity, in that having all the agents exactly the same allows for any robot to be interchanged with any other robot.

2.1.4 Homogeneity

The homogeneity of a group describes how similar each member of the group is to the other members. A homogeneous group is identical in every way including both physical design and their control behaviours whilst a heterogeneous group can be varied in either one or both of these categories. A heterogeneous group may be made up of completely unique individuals or sub-groups of identical individuals in any proportion to the overall group size. Getting heterogeneous groups to work together is an interesting research problem as each member can be capable of completely different things. The following are examples of research which utilise heterogeneous robotic groups:

- A task where three different robots are used to map an indoor area (Simmons et al., 2000). The robots are aiming to complete the task as efficiently as possible by considering their placement relative to each other. In this scenario there is no clear advantage in using a heterogeneous group of robots as they all have identical scanning and communication capabilities. One of the robots has tracks rather than wheels which may allow it to travel on different surfaces, however this is not used.

- Heterogeneous robots are shown to be capable of cooperating to identify their location (Fox et al., 2000). In this scenario there are two types of robot, those that use sonar sensors and those that use laser-range finders. In the environment there are obstacles only identifiable by either of the sensors. By sharing the information of these positions with each other both types of robot can determine their location in a space.

- Robotic agents capable of obstacle detection and avoidance guide a group of robots with only kin-recognition to create a sensor network (Parker et al., 2004). This system would in practice allow simpler, and therefore cheaper, robots to be distributed autonomously whilst the more able robots could be used again for other tasks making the overall system more cost effective.

- A cooperative anchoring problem where differently capable robotic agents share information with each other in order to confirm the locations of objects of interest (LeBlanc and Saffiotti, 2008). In this system there are two mobile robots both with vision systems and one with a symbolic task planner, as well as static camera and RFID reader, which are also considered robots. By identifying features distinguishable to each robot they can cooperate to complete the task of locating a specific object.

An advantage of having heterogeneous groups is that the individuals or sub groups of individuals can be built for specific tasks. However there are potential risks with this system. If within a group there is a specific task that needs to be completed by a specific, specialist agent and something happens to that agent that task cannot be completed. The end result is that a malfunction to a small percentage of the group can cause complete failure of the task. Another issue to consider is the number of each type of specialist robot required in the group. This information cannot always be known in

advance. A solution to these problems can be found with homogeneous groups. Every agent of a homogeneous group is identical to each other. One immediate benefit of this is the production cost of making a group or multiple groups of robots. As there are no differences in the physicality of the robots they can be mass produced far more easily. There is also no difference in the control architecture of the robots. Having a homogeneous group should allow for greater robustness. In the previous example, the heterogeneous group losing a key member or portion of the team could mean complete task failure; the same situation with a homogeneous team losing a portion of the group will not mean key tasks cannot be completed as each member is interchangeable.

The interesting aspect is making the group capable of completing the task or tasks without there being any difference in the members. One easy way to do this would be to have multiple highly capable robots working in parallel but this negates the point of using simplistic robots in the first place. A more suitable way to do this would be to have the group or swarm change the way they react and behave when dealing with different tasks.

It should be noted that Lerman (2006, p. 249) states "[r]eal robot systems are heterogeneous: even if the robots are executing the same controller, there will always be variations due to inherent differences in hardware." However, this point is pedantic and would not allow for the distinction between the two classes. Even though it is correct it does not make sense to term a group as heterogeneous when it was designed both in behaviour and physicality as a homogenous group.

2.1.5 Recall and Computation

When discussing recall capabilities of robots it is confusing to use the word memory, all robots have memory in the computational sense, however not all robots are built to remember their past actions. Robots that cannot recall any of their past actions are termed oblivious robots (Défago and Konagaya, 2002). Oblivious robots must determine their actions based only on the information currently available to them (Yamashita and Suzuki 2010).

Robots which can only recall their current state are termed finite state machines. Finite state machines have a limited number of possible states, based on their current state and what is being sensed they can change from this current state to another. The state that the agent is in determines it current behaviour. Hsiang et al. (2004, p. 79) use agents that

are modelled as finite state machines, stating they have only, "local communication, local sensors, and a constant-sized memory." Shiloni, Agmon and Kaminka (2009; 2011) use robotic agents that can recall a limited number of previous actions. The amount of previous actions the agents can recall is incredibly small compared to the amount of actions it takes to complete the task. The advantage of this system is that the amount of recall is set and designed into the system, there is no need to consider edge cases where memory limits will be reached as it is an intrinsic design choice.

In some instances the task that needs completing may require that more information is stored than a current state, for example this could be in mapping an unknown area (Thrun, Burgard and Fox, 2000; Simmons et al., 2000; Rekleitis, Dudek and Milios, 1997). There is a limit to what this robots can recall of an environment, but this limit should be larger than the general task requires. In these situations these robots recollection of the space could be considered unbounded, relative to the task, whilst their behaviours in that space are limited. The advantage of systems like these is that they can record information that cannot only be used by the robots themselves to make more informed decisions it can be used at a later time by a third party. This ability is perhaps a disadvantage in dynamic spaces when mapping, as the recollection may not match the current environment. Having a distributed system would allow for quicker responses to an environment like this.

2.1.6 Localisation

Knowing the position and bearing within a coordinate system can be helpful to each member of a robot team. Having this shared knowledge allows for easier interaction and information sharing between the group, as they all share the same reference points.

Localising a robotic agent can be done in a number of different ways. For example, depending on the size of the robots it would be acceptable to model them knowing their location through Global Positioning System (GPS). An alternate or additional method to GPS could be a shared beacon (Parker, 2002; Madhaven, Fregene and Parker, 2002). Even in an unknown environment a beacon could be sent with the robot team. The robots could find their location and orientation relative to the beacon and therefore would be able to coordinate around a shared origin.

The issues with these systems depend on the task been completed. For example if there is any interference with or there is no signal from a GPS the system will fail. The same

is true of the beacon, if this is damaged on deployment the agents would not be able to work as planned. Also any environmental interference could cause issues.

An alternative to having a shared localisation of the team is to have individual and independent localisation. This method can make tasks more complicated to solve in the design phase and increase the complexity of the individual robots. However, they would make up from the lack in technological capabilities by being more robust where faults occur and universal in where they can be implemented.

A compromise is found where the team of robots cooperate in order to localise themselves in the space. Fox et al. (2000) propose a system for the self-localisation of a group of robots. The robots are capable of finding their location in a space by sharing information with each other, which helps each agent confirm their position. This task is completed with a probabilistic approach. A similar approach is used by Spletzer et al. (2001). Here a team of robots keep in formation, whilst travelling along a prescribed trajectory, by identifying their locations relative to each other.

2.1.7 Synchronisation

Synchronisation describes the way that the robots or agents of a group complete their computations. At the very simplest level, the agents of a system can each observe, calculate their next action based on the sensed data, and perform an action based on their calculations. In a fully synchronised system each agent performs each of its routines (for example sense, calculate, or move) at the same time as the other agents. Synchronised systems are most likely used when simulating the behaviour of large groups of agents as it is simpler to have all the calculations of one type performed by each member of the group in sequence. For example, all the agents sense their surroundings, then all the agents calculate the response to the surroundings, then finally all agents act according to this response and this is then all repeated.

In general a synchronised method is not how physical robotic multi-agent control systems are implemented. If a synchronised system were to be developed for a physical robot group, some mechanism would need to be put in place to keep the synchronicity. This could be a control pulse of some sort or by having all the agents synced and allowing a certain amount of time for each action to be performed. The problem here is being sure that the agents can be kept in synchronisation with each other. Any slips in

the synchronisation could make drastic changes between the simulated model and that of the physical model.

Due to these issues of implementation from software simulations to physical simulation more realistic simulation models where each agent performs the same actions but not necessarily at the same time have been devised. These asynchronous simulations are discussed most often in multi-robot formation and gathering problems, as these are the base level tasks for multi-agent behaviours.

Flocchini et al. (2005, p. 150) state their robots are fully asynchronous and have "no common notion of time, and the amount of time spent in observation, computation, movement, and in inactivity is finite but otherwise unpredictable." This is done in order to try and make the simplest robot capable of completing a gathering task. The robots gather as long as they share some knowledge of a common compass. Souissi, Défago and Yamashita (2005) follow this research to describe a solution to the same problem with unreliable compasses. In this case the compasses start unstable but will eventually stabilise, however when this occurs is unknown to the agents. Klasing, Markou and Pelc (2008) follow a similar approach where each of their three actions of look, compute and move are asynchronous relative to each other. However, this is completed without a common compass bearing. A ring formation is possible for an odd number of agents in all cases and an even number of agents as long as the number of agents is not two, the initial configuration is not periodic and there are no edge-edge symmetries.

2.2 Summary

Through the study of the different aspects of multi-agent control architecture it has become apparent that there are two schools of thought. These different agent types are covered partially by Shiloni, Agmon and Aminikia (2009; 2011), the simple ant robot and the more capable elephant robot. The elephant robot seems capable of completing tasks due to its superior capability yet the research into different aspects of multi-agent systems suggests that the simpler ant like robot might provide a way for making a more adaptable system. One that is robust, scalable and flexible. These types of systems, within the robotics domain, are often linked to swarm robotics research. In fact the use of the word 'ant' to describe one type of the individual agents capability ties neatly with the inspiration that was taken from the social insect group to create swarm robotic behaviours. Beekman, Sword and Simpson's (2008) introduction to the biological

foundations of swarm intelligence shows the strong influence between the two. It is these adaptable traits of swarm robotic systems that make the method a suitable choice for the cooperative object recognition task.

Chapter 3: Swarm Robotics

From the investigation into multi-agent systems it was determined that a swarm robotics approach would be applicable for the cooperative object recognition task with limited capability agents. This chapter provides an overview of the most prominent research areas of swarm robotics. A discussion on the natural influences on swarm robotics is considered as well as a characterisation of the cooperation in the individual tasks being either necessary or efficient.

3.1 Research Areas

Those researching multi-robot control architecture examine ants' behaviours as they are visibly capable of completing so many complex tasks and yet are relatively simple individually. What makes these naturally occurring social insect groups so interesting is that there are no leaders to control the groups' behaviours nor is there a master plan that any individual has knowledge of. It is the simple interactions and behaviours of each agent on a local level that causes solutions to tasks on a colony level to emerge.

Ants in nature, for example, are capable of finding the shortest route to sources of food (Goss, 1989), they can determine which areas would make the most suitable homes (Franks et al. 2003), they can cooperate to move objects more efficiently (Franks, 1986) and they can sort their brood (Sendova-Franks, 2004) or organise their dead into clusters to avoid diseases (Diez et al., 2011). The ants complete these tasks despite their relative individual capabilities and without understanding what their part in the whole operation is. As well as influencing swarm robotic behaviours these social behaviours have also lead to swarm intelligence for optimisation (Kenedi and Eberhart, 1995; Eberhart and Shi, 2001; Blum and Li, 2008). By looking at these characteristics and the research in swarm robotics closely a greater understanding of what makes these behaviours so interesting can be achieved.

3.3.1 Aggregation and Dispersion

Before a swarm of robots can complete a given task they may need to move to the correct locations relative to each other. Generally this is done by the swarm aggregating (moving together) or dispersing (moving apart). Aggregation is a commonly seen behaviour in nature where animals gather into groups. Some fish move in schools, birds migrate in flocks and mammals herd together (Reynold, 1987). Reynold (1987) emulates

this group movement behaviour with the simulation Boids, this shows again how the interaction of simple rules can cause more complex behaviours to emerge. Often this natural aggregation behaviour is for protection either from predators or the elements (Hamilton, 1971).

In swarm robotics aggregation is a prerequisite requirement for swarm movement and self-assembly especially if the swarm is initially dispersed. Dorigo et al (2004) discuss how the act of aggregation is used to bring together a group of s-bots which then can connect to each other to form a swarm-bot. They show how their aggregation behaviours, which utilise sound emitters and sensors on the robots, are scalable for group sizes up to forty.

Soyal and Şahin (2005) discuss two forms of aggregation. Firstly, the agents in the system use environmental clues to organise themselves; examples of this behaviour in nature are seen both in flies who use light and temperature to aggregate and also in sow bugs who use humidity. The second behaviour is one that uses cooperative behaviours between the agents, this is harder to mimic in swarm robotics but leads to a more robust system. This difficulty arises because the behaviours in the second system are tied to the robot, where there are no external environmental clues directing them one way or another. However, in an environment where there is uniformity this second system could still function by reacting to each other, making it more robust.

Other insects have influenced aggregation control in swarm robotics. Soysal and Şahin (2007) use characteristics of cockroach behaviour to aggregate a group of robots initially dispersed in a closed arena. The individual agents have no information regarding the size of the arena or the number of agents in the arena at any time. As each agent's sensor range is less than the arena size it ensures that it is not capable of knowing the conditions of the whole arena at any time. These limitations allow for easier scaling for future applications as the number of agents and the size of the arena are unknown to the agents. Therefore the agents do not need updating to cope with different tasks that vary in scale.

Looking at only the key behaviours, of avoidance, approach, repel and wait, involved in an aggregation task Soysal and Şahin (2005) devised four different behavioural models for bringing a group of agents together by combining them in different ways. These low level behaviours show that it is not necessary to mimic actual individual natural

behaviours to produce recognisable group behaviours and that by using basic key techniques in different configurations new systems can be created.

Dispersion is the opposite behaviour to aggregation, spreading the agents out across an area. This generally needs to be done in a way that covers the most area without individuals becoming too detached from the rest of the group. McLurkin and Smith (2007) examine five different dispersion techniques for indoor environments. These include:

- **Ideal gas motion:** The agents move as molecules would in an ideal gas.
- **Disperse from source:** The agents move away from a single source.
- **Avoid closest neighbour:** The agents each move away from their closest neighbour.
- **Disperse uniformly:** The agents move in the opposing direction to the vector sum of the positions of their nearest neighbours whilst remaining within a set maximum boundary distance.
- **Directed dispersion:** A combination of two types of dispersion technique, disperse uniformly and frontier guided dispersion, where agents move towards unexplored areas.

Again these techniques are not directly influenced by the behaviours of any specific individual animal. The same can be said for Pugh and Martinoli (2007) whose methods for constructing a searching swarm is inspired by Particle Swarm Optimisation. Although Particle Swarm Optimisation was originally influence by the movement and organisms in a bird flocks and fish schools (Kennedy and Eberhart, 1995).

Dispersion is used in foraging, section 3.3.2, as well as building dynamic networks in multi-robot control. Ludwig and Gini (2006) propose a method of distribution for a surveillance sensor network application. The agents use the intensity of wireless signals transmitted from the robots to provide a rough estimation of their distance apart from each other. However, the agents have no idea of the bearing of the signals. They can however estimate the direction by tracking how much the signal intensity changes as they move. This choice simplifies the design of the robot, something that is important to consider as it leads to simpler manufacturing processes.

A real world application is described by Payton et al. (2001) where the gas expansion and guided growth dispersion of a swarm of simple robots through an unknown environment can provide guideposts for larger more powerful robots or for human operators to follow with the aid of augmented reality glasses.

3.3.2 Foraging

Foraging is the act of collecting items or objects and moving them to a certain area and is highly influenced by the behaviours of ants. The agents of the swarm need to disperse into the search area, find the objects and either return them back to a certain point or move them into clusters. It is usual for the objects to be moveable by a single robotic agent, however there are some instances, which combine both foraging and cooperative transport, section 3.3.4. Objects in foraging tasks are often referred to as pucks or where more influenced by natural systems the objects are prey and the collection point the nest. Winfield (2009) states that foraging is a bench mark test for swarm robotics as it is inspired by social insects, it is a complex task that deals with three subtasks; exploration, physical collection and navigation to certain points and finally to be efficient it requires cooperation between the members of the swarm.

The way in which the agents cooperate with each other is important to consider. It would be possible to have a swarm simply act in parallel without interactions with each other, however this is not exhibiting an intelligent behaviour on the part of the swarm. In both cases simply adding more and more agents into the system reaches a point where it does not help any more in fact it can cause problems. This is because of the amount of interference that happens around the collection point of the foraged items. For this reason foraging is the most widely used domain to investigate the effects of size scalability on performance (Shell and Matarić, 2006). In order to deal with this problem Shell and Matarić (2006) compare a 'bucket passing' idea, where the items are moved gradually by different groups of agents towards the base, with that of a more standard homogeneous method. They go on to suggest that there is a difference in degree rather than a difference in absolute type when considering different strategies. By considering this it is possible to get a gradient of different strategies. The need for this type of adaptive system is also clear from Sugawara and Sano's (1997) research. In a foraging task they measure the success of uncooperative and cooperative agents to deal with different distributions of pucks in an arena. The pucks were spread uniformly across the whole arena, spread uniformly in a quarter of the arena and also placed in a tight cluster.

Where the spread was uniform the uncooperative agents performed better, where there were tight clusters the cooperative agents, who signalled the other agents, performed better. It has been demonstrated by Liu and Passino (2004) that swarms deal better in noisy environments than individuals in a foraging task, where the agents used an attraction and repulsion function to forage. Atkin, Balch, Nitz (1993) also consider the use of communication in foraging. They find that the addition of communication increases the efficiency of the group as agents spend more time moving prey rather than finding prey.

Rather than specifying a single point for the cluster to be formed, it is possible to have the cluster form naturally. Kazadi, Abdul-Kahliq and Goodman (2002) completed research where items can be picked up moved and dropped. Items are more likely to be picked up if they are on their own and more likely to be dropped if near a cluster. This was found to eventually lead to a single large cluster.

Mimicking the behaviour of social insects Krieger, Billeter and Keller (2000), Liu et al. (2007) and Liu, Winfield and Sa (2007) have the agents forage for energy. This forms an interesting balance as the agents both need energy to forage and gain energy from successfully foraging. Within the models the items being foraged replenish over time, as would a natural food source. In a similar foraging task Labella, Dorigo and Deneubourg (2006) examine the efficiency of a group of agents that divide their labour between agents. The division of labour is determined by their individual ability to retrieve items of prey. In all these examples the agents attempt to react to their environment to work efficiently, with individuals only collecting prey when it is beneficial for the swarm to do so.

The real world applications of foraging could be as varied as toxic waste clean-up, search and rescue, the removal of mines and collection of terrain samples (Campo and Dorigo, 2007). As it can be time consuming to run both physical and soft simulations it is common for analytical mathematical models of foraging behaviour to be developed for more efficiently finding variables (Lerman, 2006; Hamann and Wörn, 2007; Campo, 2007).

3.3.3 Self-Assembly and Connect Movement

Within swarm robotics self-assembly describes the act of multiple agents attaching to each other in different ways to form structures that change the functional capabilities of

the swarm. An example of this in the biological world is ants forming chains to bridge over gaps that would be too long for a single ant to traverse alone (Lioni et al. 2001). For this act to happen without any central control and with truly homogeneous agents is a difficult feat. Although once successfully implemented the system could be capable of many varying real world situations allowing the swarm to adapt to tasks of different magnitudes, for example pass through a small tunnel individually and then combine to traverse a chasm larger than an individual.

Groß et al. (2006) list five ways that the self-assembly of robots can aid the robotics domain; self-repair; self-replication; additional mobility; parallelism and increased force.

- **Self-repair:** If the robots are modules within a larger construct, any damaged module can simply be replaced with a working module. This may, depending on the location of the module and function of it, require different ranges of disassembly.

- **Self-replication:** Using suitable building block agents available in the environment the assembled robots can replicate themselves.

- **Additional mobility:** As the robots assembly together to form different shapes they may be able to traverse terrain or obstacles they could not move over individually. This could be bridging a gap or climbing over a step.

- **Parallelism:** By detaching the individual agents from the assembled robot they can then perform tasks in parallel.

- **Increased force:** The capable force output of the robots could increase as additional robots are added to the assembled structure.

In practice self-assembly can be a difficult task to complete as once a new structure is formed, each member needs to know what its current role is in the task. Gilpin et al. (2008) describe a method for building different shape structures from dissembling a three-dimension block of multiple cubes. This method would allow agents to form into generic shapes, which are easier to build and then disassemble into more complex forms, which perhaps could not be built in a purely additive manner. A comprehensive overview of modular self-reconfigurable robot systems was written by Yim et al. (2007).

One of the more common used platforms for self-assembly is the s-bot system. Each agent, or s-bot as they are termed, has a LED ring array that displays the robots current state, a 360 degree camera and a gripper. The gripper allows the s-bots to link together to form different shapes without the need for specific linking locations. Groß et al. (2006) showed how the s-bots could link together even on rough terrain. The s-bots can assembly into different formations which can allow them to traverse gaps in terrain or climb over obstacles (Tuci et al. 2006). Another application to bridge crossing is suggested by Arbuckle and Requicha (2004) where rather than using passive components for nano-scale construction a swarm of self-assembling robots could be used. This would reduce the problem of one thing manipulating another to one thing manipulating itself.

3.3.4 Cooperative Transport

An extension of the foraging problem is the cooperative transport task. Within this research field the swarm of robots need to work together in order to move an object that is too large or heavy for a single agent to move on its own. To do this the agents must coordinate with each other in such a way that the majority of them are all trying to move the object in the same direction. Without a centralised system in place there needs to be an emergent way to do this task. This problem has been solved in the natural world by ants. Kube and Bonabeau (2000) go into great detail considering the movement of objects both individually and cooperatively by groups of ants and how this inspires their control techniques for robots. More specifically they consider the avoidance of stagnation, which could occur if all the agents are trying to move the object in different directions. The cooperative transport tasks can be reduced into subtasks; find object, move to object, push object and push object to goal (Kube and Zhang, 1996). In order to complete the task the robots changed between these subtasks based on the perceptual clues; can they sense the object; are they in contact with the object; are they moving and where is the goal?

As agents in the system are simplistic in nature they often have no explicit knowledge of how their reactions will affect the world or what they have done in the past. This means they are unable to predict what will happen to the box in the future if they push it and cannot learn from their previous moves, therefore they can only react to the present situation. Matarić, Nilsson and Simsarin (1995) studied this problem with a system using two six-legged robots and an elongated box. They found that two robots were more

efficient at moving the elongated box than one. This was a general multi-robot control problem and not one inspired by social insect emergent behaviours as in swarm robots. Simply adding more robots to this system would not necessarily improve the efficiency. In fact even in a swarm robotic system adding more agents into the system does not necessarily make the task completion more efficient. Groß and Dorigo (2004a) used s-bots to move different size and weight objects by either pushing or pulling them. The s-bots assembled together in order to increase the force they were capable of applying on the object. They experimented with groups of between four and sixteen robots and found that although once the agents had assembled they could move the objects quicker with more robots, it took longer for the agents to assemble in the first place. Kube and Zhang (1994) also carried out research on how the number of robots trying to move an object changes the efficiency of completing that task.

The issue here is organising the robots behaviours in a suitable way to deal with the cooperative transport task. Campo et al. (2006) research the uses of negotiation between robots in a cooperative transport task and how it affects the time taken to move the object and the accuracy of the direction which the object is moved. They use four robots, of which three are required to successfully move the object. However, a single robot acting in the incorrect way may make the object un-moveable. The four strategies they suggested were:

- Transport directly.
- Negotiate then transport.
- Negotiate then transport and negotiate.
- Negotiate and transport.

Each of these was tested with different levels of noise. The most successful was the negotiate and transport strategy. They suggest this outcome is perhaps counter intuitive as less time is spent negotiating than the negotiate then transport and negotiate strategy. They suggest this outcome is because before the object is moved the robots are unable to assess if what they are doing is correct, therefore making decisions without any initial feedback slowing the early progress.

Given that it is possible to physically move the object it may be necessary to guide it to the correct place. Using the s-bot Nouyan et al. (2006) devised a system where the

agents who initially randomly disperse in the arena search for the base and the prey. Those that find the base or the prey display the same colour as the base or the prey becoming part of that object. Eventually a chain is formed between the two which is shortened until the prey is moved to the base.

All these cooperative transport tasks use a single object but could be expanded upon for the collection of multiple large objects. Zhang, et al. (2007) research this issue with a task where agents move multiple objects of unknown size and weight. Unlike the traditional foraging task the agents here must cooperate in order to move any single object. In this case the foraging task would be impossible to complete by a single agent to complete, as is true of all cooperative transport tasks by their definition.

3.3.5 Pattern Formation

Many tasks in swarm robotics could benefit from the agents forming certain patterns and shapes before completing those tasks. An example of this is self-assembly, having the agents move to the correct positions relative to one another before linking together could improve the efficiency of the whole process. For example in Balch and Hybinette (2000) research the robots are attracted to attachment sites around neighbouring robots in order to move a formation across an arena containing multiple obstacles. Where different attachment sites affect the formation produced. The problem, however, in pattern formation is completing it with the lowest capability agents possible. This restriction leads to agents that, in different combinations: do not have a shared frame of global reference or bearing; are oblivious (section 2.1.5); are anonymous (section 2.1.3); and are non-synchronised agents (section 2.1.7).

There have been numerous studies into pattern formation of a group of decentralised autonomous agents. In some instances the agents were given individual IDs meaning they were not considered anonymous. In the research of Lemay et al. (2004) four agents are used, each agent is aware of the position of the other three agents and can then determine the best position for themselves using this information. The agents could form an arrow, column, circle, diamond, wedge or line. Fredslund and Matarić (2002) reduced the need for the agents to consider the position of all other agents to considering a single agent; their leader. However, doing this reduced the formations possible to chains that follow the main leader, either with the leader at one end of the chain or in the middle. Having a single leader gives a single point of failure. If the leader does the wrong thing, the whole group does the wrong thing. Ren and Sorensen (2008)

therefore devised a method where the number of leaders could be changed without affecting the complexity of the control system. The four agents' ability to remain in a square formation was measured using physical robots. Increasing the number of leader agents increased the robustness of the group by avoiding a single point of failure. Pavone and Frazzoli (2007) used the idea of cyclic pursuit, where one agent follows its leader to form circles and spiral formations. They adapted their system so it could be completed with autonomous agents using a convex hull system.

Défago and Konagaya (2002) also use autonomous agents to form a circle formation but in order to do so the agents required unlimited vision allowing them to know the position of all the other agents in the system. Using non-synchronised agents modelled as points with no common coordinate system Défago and Souissi (2008) have the initially randomly placed agents form non-uniform circle formations with agents evenly spread around the circumference within a finite amount of time. It is not necessary to have the agents aware of the entire arena space. Fujibayashi et al. (2002) show this with the use of virtual springs that act between the agents. The properties of the springs depend on the formation required. The spring like connections can be broken with a certain probability allowing either ladders, triangles or hexagon formations to be formed. Suzuki and Yamashita (1999) show that the initial configuration of the robots can determine the geometric formations that the robots can go on to form when using anonymous agents, due to issues of symmetry and agreeing on a shared coordinate system and bearing.

3.3.6 Self-Organised Construction

The self-organised construction task requires the swarm of robots to work together to build structures. To do this the agents must find the required piece and move it to the correct place at the correct time. Often the pieces are identical and there are numerous possible places to put that piece which means the task can be carried out in parallel by multiple agents.

Although construction is a three-dimensional problem, the problem is often reduced to that of a two-dimensional problem. The building blocks themselves are often identical meaning that there is no specific order to collect them in, increasing the scalability and robustness of the task. Wawaerla, Sukhatme and Matarić (2002) found that increasing the number of robots also increased the efficiency of the system to complete the task of constructing a wall from alternating coloured blocks. However, there was often

interference between the robots trying to complete similar tasks in the same area. This issue could be addressed by splitting the agents into teams as done by Crabbe and Dyer (1999). Here they use one group of robots to move the blocks into roughly the correct place and one group to arrange them neatly. This allowed them to build more complex corridors and intersections.

One idea to improve the capability of the construction is to use smart building blocks. In the research of Werfel, Bar-Yam and Nagpal (2005a) the smart blocks act as beacons informing the robotic agents where to construct. The structure itself contains the information of its state and what needs doing to it so the agents do not need to hold this information. This idea is expanded upon (Werfel et al., 2006) by exploring four types of blocks; inert blocks, distinct and inert blocks, writable and inert blocks and finally communicating blocks. The distinct blocks hold information that makes them distinguishable from each other so the robots can build a dynamic label map of the system. With writable blocks, the robots can change the states of the blocks. Finally the communicating blocks can store, process and communicate information to each other and the robots. Their findings show that more capable the blocks the better the system performed overall. They discuss the problems of failing smart blocks which may hamper the system. An error correction procedure is described (Werfel, Bar-Yam and Nagpal, 2005b). Again the issue of interference arises even with smart building blocks. Terada and Murata (2006) consider the construction of T-shapes and L-shaped structures. The corner sections caused problems of interference. Two methods were used to deal with collisions between the robots; module relay where any robots colliding passed the module on and also blackboard planning which prevents collision by using local communication.

The use of smart block techniques are also used in three-dimensional construction problems (Werfel and Nagpal, 2008). These techniques are useful due to the inherent difficulty of constructing in a three-dimensional lattice. Werfel (2006) shows that it is possible to construct a two-dimensional shape from identical two-dimensional blocks, with directions indicated on each edge, that would allow a robot following those directions to visit every outside edge of that object and return to where it started, even if blocks are added or subtracted to the object. However, this is not possible with three-dimensional cubes where a robot is required to visit every face. Using smart blocks allows information that the cubes contain and communicate to the robot to change

meaning the blocks can adapt to their current constructed shape, making visiting each face possible.

Social insects' construction techniques have also been mimicked. Theraulaz and Bonabeau (1995a; 1995b) draw inspiration for the construction of three-dimension lattice formations from wasp colony nest construction. They were capable of producing structures with regular patterns and those which resembled the nest of different wasp colonies. Virtual pheromone plumes can also be used to guide the construction of blocks into an arcing wall in a two-dimensional simulation (Mason 2002). Termites have also inspired a physical robot system capable of producing balls of foam that harden over time to produce ridged structures. The robots can climb onto this structure and add further foam at higher heights in order to produce different structures (Bowyer, 2000). The main difference is that the construction material is not found in the environment but given the robots initially.

It may be the case that any structure built will need to be adapted or changed. De Rosa et al. (2006) show that by adding spaces or voids into a construction this is possible. Initially starting with a square source shape, which had randomly placed voids inside of it, they were able to successfully produce the target shapes of a T-shape, a rectangle and a circle. This is as long as the voids are sufficiently large to cope with the movement of the blocks.

3.4 Discussion on Swarm Robotics Research Fields

3.4.1 Efficient Aiding and Necessary Cooperation

From the research found in this literature survey there seems to be two general types of swarm robotic tasks. The first is where the actions of the swarm improve the efficiency of completing a task which could be completed by an individual. The second type is where the swarm or sub-group of the swarm work together to complete a task which they could not complete individually.

For example consider collecting pucks into a cluster:

- A single agent could move around the space collecting pucks one at a time, therefore not cooperating at all.
- A group of agents could perform the same actions as the single agent parallel to each other. Probably reducing the amount of time it would take to collect a specific number of pucks.
- A swarm of agents could share information with each other on puck rich areas potentially increasing the efficiency of the collection by targeting these specific areas.

Although agents individually could find rich puck areas and keep returning to them, sharing information about the different areas allows the entire swarm to have a wider knowledge of the environment to target the most rich areas. Depending on the specific task this is the same for construction, an individual could construct on its own using the same techniques as a group of robots working in parallel, however, potential swarm techniques could further improve the efficiency of the system.

The second type of general task has robots that have to work together or they simply will not be able to complete the task. This type of system is of more interest to this cooperative object recognition research. It is not simply a matter of increasing efficiency through effective communication, it is a matter of getting something to work which can only work through the successful cooperation of a number of agents. Behaviours of this type are apparent in aggregation and dispersion as you cannot gather or disperse a single robot, this is also true of pattern formation. These tasks involve the organisation of the robots either relative to each other or their environment but without necessary direct interaction with the environment. Finally there is cooperative transport and self-

assembly. In the cooperative transport task it is not possible for a single robot to move the object, it takes coordination between many robots. Any transport task could be solved by building a sufficiently strong robot but this solution would be limited to a certain object parameters. Solving the problem through coordination between many robots provides a solution that is scalable and flexible enough to deal with different object sizes and weights. Again with the self-assembly the combination of the robots makes them capable of something they were not capable of before as individuals. By assembling together they can cross gaps they could not cross before, climb over objects or move over terrain they could not traverse before.

Although both these general types of tasks are part of swarm robotics one focuses more on efficiency and the other is more concerned with capability of the group verses the individual's inability. Efficiency in this case been a measure of both how quickly a task can be done and the amount of energy required, and capability meaning simply to be able to do the task at all. Although it should be noted that both issues are a concern for each area type.

3.4.2 Identifying Objects

In all situations, within the literature, involving the finding of, retrieval of and manipulation of objects each agent or robot in the system was aware when it found the object required. This individual object identification has been done in different ways for different pieces of research. In some case a lit box is used to draw the attention of the robots (Kube and Bonabeau, 2000) or similarly it has also been done using specific colour LEDs to distinguish it from the robots and base point (Groß et al., 2005; Groß et al. 2006). In these examples of cooperative transport there is only a single object to be moved. Even when there is a requirement to move different types of objects, the differences in the objects are made clear to the individual agents, an example of which can be seen in the simulation by Zhang et al. (2007). This same notion is repeated in the general foraging tasks, each robot becomes aware when they have collected a puck and knows that it is a puck they have collected. Finally the same assumption is in self-organised construction. Alternating coloured blocks are used (Wawerla, Sukhatme and Matarić, 2002) for example.

Where the focus of the research is to find workable strategies to, construct, forage or move it is logical to provide information to the robots about which items or objects are required. Adding object distinguishability for a variety of object types would add a

further layer of complexity that would distract from these original issues. However a need for systems which can address cooperative object recognition is shown by the existing research, detailed in section 4.1. The benefits of using a swarm robotic method could potentially improve the scalability, robustness and flexibility of these systems. Another reason that the swarm robotic method may not have been considered so far is that there is no analogy available in the natural world.

3.4.3 Inspirations from Nature

Nature can provide a great amount of inspiration for many forms of technology. This inspiration can be direct, where an aspect of nature is copied almost exactly, or indirect, where the root behaviours used in nature are developed to solve tasks in a similar manner. Many aspects of swarm robotics systems take the more direct inspiration approach. Ants find the shortest routes using pheromones (Goss et al., 1989) and robots find the shortest routes using artificial pheromones (Payton et al., 2001). Wasps construct nests using simple rules, a swarm of agents use similar rules to construct structures (Theraulaz and Bonabeau, 1995a and 1995b). There are groups of ants that cooperate to move objects too large or inefficient for a single ant to move and there are swarms of robots that do the same (Kube and Bonabeau, 2000). The similarities between the agents of social insect groups and the neurons of a brain are considered in research in swarm cognition (Trianni and Tuci, 2011) and have led to vision systems implemented in robots (Santana and Corriea, 2011). The behaviours of schools of fish and flocks of birds have also directly influenced the way swarms of robots move in groups (Beni, 2005). It is perhaps due to the lack of a direct form of inspiration from social insects or any other group in nature that there has not been any research into cooperative object recognition with a swarm robotic approach. However, there is no reason why direct inspiration should need to be taken from a specific social insect behaviour. Although this is a suitable starting point for understanding how the techniques can be implemented it can soon become limiting. Already there are common adaptions of the behaviours seen in insects due to the hardware available in robotics. Systems inspired by ants and termites have to change and adapt how pheromones can be used in an electronic platform. This has been done in different ways; RFID tags (Mamei and Zambonelli, 2007), communication through LEDs (Groß et al., 2005; 2006) and transceivers (Payton et al., 2001). This suggests a sliding scale from direct inspiration to indirect inspiration in swarm robotics applications.

Using indirect inspiration further along that scale can potentially be more open; there is nothing in swarm robotics research that states it cannot be done in this way. Şahin (2005, p. 12) states "swarm robotics is the study of how large number 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." According to Beni (2005, p. 7) "the principles underlying the multi-robot system coordination are the essential factor" to making swarm robotics what it is "the control architectures relevant to swarms are scalable, from a few units to thousands or millions of units, since they base their coordination on local interactions and self-organisation."

Although direct and specific inspiration is helpful on two accounts, improving multi-robot control and understanding social insects or other natural occurring swarms, it is really the core elements that distinguish swarm robotics as a type of multi-robot control. By taking these control ideas a system can be produced for identifying objects cooperatively.

3.5 Summary

Many of the tasks research with swarm robotic techniques are inspired directly by naturally occurring systems. Restricting research to mimicking behaviours reduces the amount of potential applications to those that are already visible. Instead it also advantageous to look at the core behaviours present and use those as building blocks for solving different tasks.

A division between two types of cooperative swarm task was identified. The behaviour of the agents can be considered necessary or efficient. Where the cooperative behaviour is considered efficient the robots can complete the tasks individually but do better by working together intelligently. In the case of necessary cooperative behaviour the agents must cooperate or they will not be able to complete the task.

Chapter 4: Related Research

This chapter reviews the research into cooperative object recognition, distinguishing them by whether or not the cooperation is necessary to complete the task or it allows for a potentially more efficient system. A brief introduction into Genetic Algorithms (GA) is provided before a detailed review of multi-agent systems that have used the evolutionary algorithm to determine their behaviours and responses in a range of scenarios. Finally, an overview of both physical and simulated platforms for multi-agent systems is included.

4.1 Cooperative Object Recognition

The act of a group of agents working together in order to recognise an object can be considered cooperative object recognition, this has been achieved with both multi-agent and multi-robot systems. A portion of these systems consisted of robots, using relatively complex sensors and computational abilities, that are capable of completing the task alone and are individually capable of identifying objects and only utilising cooperation to increase the efficiency and robustness of the task. There are also systems where the cooperation is necessary in order for the agents to identify the object.

4.1.1 Efficient Cooperative Behaviours

Ye and Tsotsos (1997) research how a multi-agent group searches for a non-moving object in a given space. The search area is divided into a known number of cells which the robotic agents search. Each agent has a camera that can pan, tilt and zoom which is used to attempt to identify the objects position. The agents' individual knowledge consists of a probability distribution of the area which notes the likely hood of the object being in a specific cell. Through their use of this individual knowledge and shared group knowledge they cooperate to determine their next individual action in order to locate the object's position accurately. The robot considers seven parameters, the x,y and z coordinates of the camera, the width and height of the solid viewing angle of the camera and the pan and tilt of the camera in its search of the object shape. A limitation of their research is that it considers only the position of a single object rather than the identification of the object from a range of possibilities.

Oswald and Levi (1997; 2001) research a method where individual agents' hypotheses of the identity of an object shape are compared and combined, and expressed in degrees of

belief using a Bayesian approach. Each agent receives their own image to analyse through a statistical recognition algorithm. The algorithm compares the objects to a list of potential hypotheses of what the object could be. This method improves the robustness of a hypothesis through shared information. However, the individual agents are still capable of predicting the type of object it is through the recognition algorithm, which requires a camera and knowledge of existing shapes.

Büker (2000) agrees that the "robust recognition of complex 3D objects is often impossible when evaluating only a single 2D image of a scene" and provides a variation on the method for collaboration between the agents. As in the other cooperative object recognition research each agent is equipped with a camera and image processors. In this case the agents use Blackboard communication, where the information is stored centrally on a specific agent. The agents attempt to identify an object individually, when they hypothesise the object type a request for verification is sought from another agent. It is through these actions that they reduce the amount of errors in the object recognition.

In these three examples the agents require, cameras and the capability to analyse the images they receive. The agents also need to know where they are in relation to each other and the object. All these amount to relatively complex individuals with complex interactions. It is possible for agents to distinguish between objects without these complex capabilities. However, this is at the cost of the complexity of the shapes they can distinguish between. Tuci, Trianni and Dorigo (2004) evolve the neural-network of a single agent to determine if it is in one environment or another. In one of the environments a light source is completely surrounded by an obstacle ring, in the other environment there is a gap in this ring allowing the agent to complete its task of reaching the source of light. If it is in the environment where the task is not achievable, the agent must determine when to give-up trying to reach the goal. This has been expanded in to a two robot system which evaluates the benefit of communication between the robots in completing the task (Ampatzis et al, 2006). In these systems, the environments are considered different, but the only difference in their environments is the single obstacle that potentially stops the robot reaching the goal. Therefore this could be considered an object recognition task, where the obstacle is the object that is being recognised.

In all these systems the agents are individually capable of distinguishing between objects on their own. It is the robustness and accuracy or efficiency of the individual agent's hypothesis on the object type that are improved through the sharing of information. If it were possible to have a swarm of agents cooperate as the system of recognition rather than verifying each other's actions the potential for further applications could be increased. Agents forming round the shape of an object, reacting to each other's placement around the contours of the shape could cooperate to identify the object through its shape and not through the analyses of numerous images.

4.1.1 Necessary Cooperative Behaviours

Research has been carried out where the agents must cooperate with each other to identify objects as individually they are not capable of doing so. McLurkin and Demaine (2009) describe a distributed boundary detection algorithm for multi-robot systems. The agents in the system identify when they are at either an internal or external boundary by the relative positions of their neighbouring robots. This is done by the agents who note gaps where they expect neighbouring robots to be, due to the way they are dispersed in the space, in order to identify convex and concaved boundary regions. For the robots to identify the boundary requires them to have unique identification and be able to estimate the position of their neighbours.

Giplin and Rus (2012a) have developed a method for a multi-agent system to duplicate inanimate objects using modular robotic cubes, where an estimation of position is not required as the agents only communicated when in contact with each other. To achieve this the cube shaped Robot Pebbles attach to each other around the object, providing them with direct communication links in specific locations. Once the shape, which is constructed from inert cubes the same size as the Robot Pebbles, is completely surrounded a signal is passed around the object mapping its shape. This information is used to create a copy of the shape from the active robots, who remain attached to each other when the rest of the robots are removed. Although this research was initially achieved with shapes with a constant height, equal to one cube, the work has been expanded into three-dimensional cube constructed objects (Gilpin and Rus, 2012b). The robots used in their research provide an example of the type of physical system the methods devised by this swarm cooperative object recognition research could be implemented on in the future. Where they have focused on replication of objects

through complete knowledge of the object's shape, the focus here is on the minimum knowledge needed to distinguish between two types of object shape.

By considering each sensor in a system as an agent the research by Tuci, Massera and Nolfi (2010) could be considered a heterogeneous multi-agent system. They evolve the neural controller for a simulated anthropomorphic arm that can distinguish between spherical and ellipsoid objects. The arm and hand joints are aware of their position through proprio-sensors, and touch sensors across the hand indicate if they are touching any object, be that the table, the ellipsoid, the sphere or another part of the robot arm. However, this multi-agent system would be considered to have strong communication capabilities and limited relative movement between the agents.

Other than physical systems, multi-agent approaches for object recognition are used for understanding visual data. Here the environment that these agents inhabit is the images of the environment that the overall vision system is analysing. Examples of this type of system include (Santana and Corriea, 2011; Ramos and Almeida, 2004; Fernandes, Ramos and Rosa, 2005). These systems require further analysis of the agents response to the images to determine the higher level reactions required.

The need for a system of cooperative object recognition is highlighted by these pieces of research that address the difficulty of identifying objects that have common features, both where the cooperation is efficient and when it is necessary. It is only through finding the differences of the objects that the objects can be identified distinctly from one another. There is currently no research that utilises multi-agent systems to distinguish between different objects through their shape alone where the agents are mobile and have limited sensor and communication capabilities.

4.2 Genetic Algorithms for Multi-Agent Systems

A Genetic Algorithm (GA) allows solutions to evolve naturally without the system necessarily understanding the mechanics of the problem. This capability would be suitable for determining the state-behaviours of a swarm of agents in a cooperative object recognition task. Using this method the swarm could evolve to distinguish the differences between two objects without requiring pre-analyses of the object shapes to program the agents to behave in a certain way for that specific task.

4.2.1 Genetic Algorithms

GA are a type of Evolutionary Algorithm which describe methods for solving complex problems through techniques inspired by Darwinian principles of evolution. A population of candidate solutions compete against each other in an environment that represents the problem space. Candidates that perform better have a higher fitness for that environment and therefore are given a better probability of being selected as parents to produce offspring. By combining the traits of different well performing candidate solutions through recombination, also known as crossover, new hopefully better performing offspring will be produced. Mutation occurs to maintain diversity between one generation and the next by changing a part of the genome at random. To keep the population consistent only a selected number of candidate solutions and their offspring survive the selection process. A diagrammatical description of this entire process, with initialisation and termination, is shown in figure 4.1. Eiben and Smith (2007) provide information about evolutionary computing and how it can be used for different scenarios.

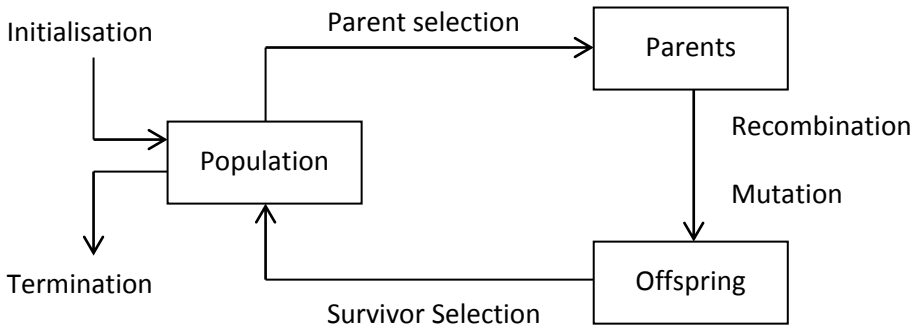


Figure 4.1: Flow diagram for evolutionary algorithm process (adapted from Eiben and Smith, 2007)

One benefit of using a GA to find solutions to problems is that it can produce unexpected and counterintuitive outcomes. For example in the design of a wire antenna (Altshuler and Linden, 1997, p43) that "radiates near uniform power over the hemisphere" whilst appearing as a counter intuitive "crooked-wire" and a satellite boom (Keane and Brown 1996) that appears twisted and warped. GA have been used for many diverse problems which include: sewer network design (Afshar, 2012); designing a concert hall for optimal acoustics (Sato et al. 2002); the placement of wind turbines (Grady, Hussaini and Abdullah, 2005); to aid in stock trading (Kuo, Chen and Hwang 2001). They have also been used for swarm robotic research, which will be discussed in more detail in section 4.2.2.

However, this ability is countered by the amount of time a GA takes to test each candidate solution. The more computer intensive each test, the longer the GA will take. In the previous example of the satellite boom (Keane and Brown, 1996) the GA was only run for 10 generations due to the finite element analyses required for each of the candidate solutions, despite which solutions where found that outperformed the original design by many factors. In an ideal world it would be possible to have very large populations and numerous generations, however this is not possible as there are restraints to how long can be spent running the program. This restriction leads to balancing the numerous GA variables in order to get the most effective output in the time allowed. These variable are considered in more detail in section 9.2.

4.2.2 Multi-Agent Systems Research

GAs have been used to develop and optimise different aspects of swarm robotic and multi-agent systems including the classic problems of aggregation, dispersion, foraging, self-assembly and cooperative transport.

Trianni et al. (2003) use s-bots with auditory and proximity sensors and sound producing capabilities to solve an aggregation problem through the use of GA. They observed two types of clustering behaviour, static and dynamic. Static clusters formed tightly packed groups within the larger group whilst the dynamic clustering has less tight groups but proved to be scalable for different group sizes resulting in a single group. Aggregation behaviours where also research by Bahçeci and Şahin (2005) where the task was to evolve strategies which maximise cluster size and minimise the number of clusters.

The force laws that governed solid, liquid and gas like dispersion and movement of a swarm of robotic agents which represent suitable behaviours for distributed sensing, obstacle avoidance and coverage tasks respectively were evolved through a GA by Spears et al. (2005).

The notion of efficient foraging is visible in Bassett and DeJong's (2000) research that used GAs to find the most effective technique of getting Micro Air Vehicles to observe as much of a map as possible whilst avoiding collisions. A collision in this scenario would cause the agent to be removed from the test.

Groß and Dorigo (2004b) use an evolutionary algorithm to produce control methods for two robots in order to have them cooperatively move an object as far as possible in an arbitrary direction in a limited amount of time. In this task the robots could only communicate through stigmergy and not directly with each other. Ampatzis et al. (2006) research the possibility of communication emerging in a swarm that was attempting to move towards a target light source in. In one version of the task it is possible for the agents to reach the target location through a gap in a ring surrounding the target and in another task it is not, as the ring is complete. Despite there being no specific fitness reward for using communication the emergence of signalling occurred in both tasks.

In the effort to explore the effect of communication capabilities on a group of connected s-bots Trianni, Labella and Dorigo (2004) used GA to evolve neural network based responses to both the auditory and traction signals produced by robots that where trying to avoid holes. The fitness function considered three elements, the speed of the robots movement, the straightness of motion and the traction between the connected robots. The GA provided suitable results for both the groups, one using only the traction sensors the other also had the auditory sensors.

The notion of selfish and cooperative agents is discussed by Yang and Luo (2007) where they form agent coalitions within the swarm through a GA approach. In order to do so effectively they introduce a two-dimensional chromosome encoding, crossover and mutation technique. These variations demonstrate how GA on a whole can be changed and adapted from the normal approach to suit specific tasks in more suitable ways, this is one reason why GA are such a versatile tool for problem solving.

4.2.3 Genetic Representation of Agents

One interesting concept that comes out of using GAs for multi-agent systems is how to create the genetic representation of the agents. There are four main methods that are described in the literature which covers evolution of multi-agent systems where the agents may have different behaviours.

1. Evolving the behaviours as a single group. The individuals' behaviours and responses are grouped into a single genome whose candidate solution fitnesses are measured.

2. Evolving the behaviours of individuals as they work in a group. The individual behaviours and their responses are encoded separately in the genome but their fitness is measured as a team.

3. Evolving the behaviours of individuals separately, before putting them in a group. The individual behaviours and their responses are evolved separately as well as having their fitness measured separately.

4. Evolving the behaviours of individuals where each member of the group has the same set of behaviours. The behaviours of the agents and their responses are identical, the fitness of the agents is measured as a group, but only one genome is required for evolutionary purposes.

These four evolutionary methods are shown in figure 4.2.

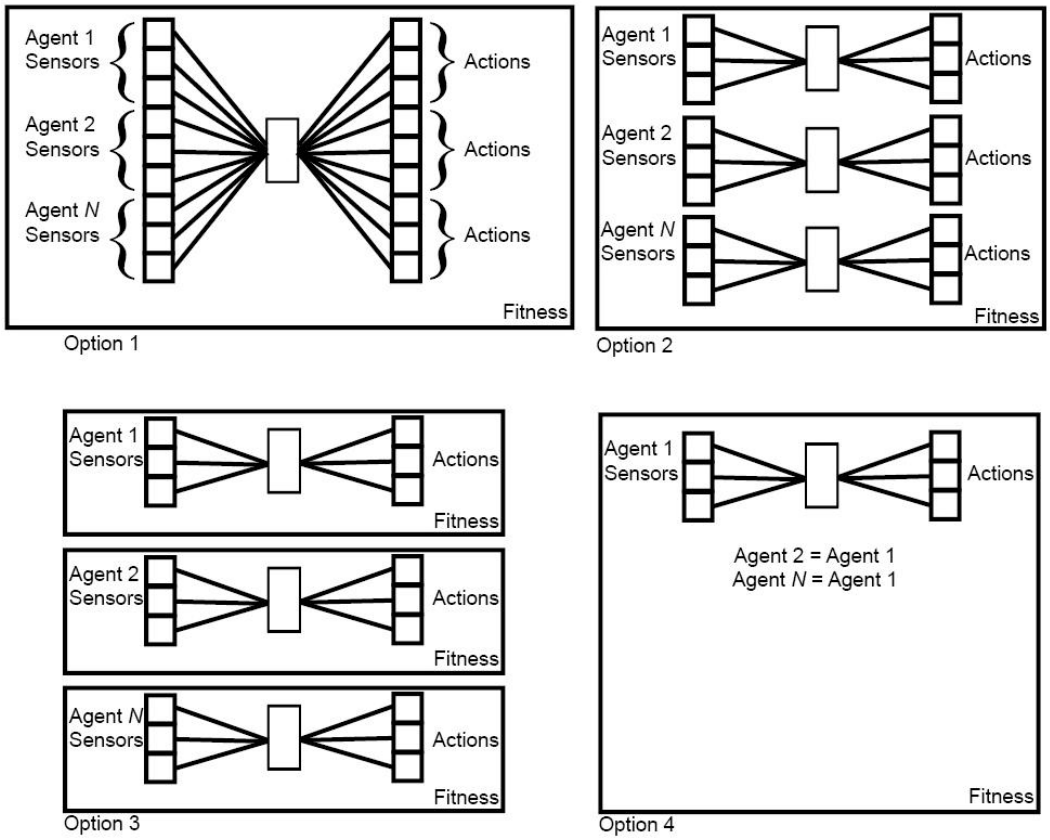


Figure 4.2. Four methods for evolving behaviours of multi-agent systems utilising a genetic algorithm. Option 1: Sensed information of all agents affects actions of all agents, fitness measured as a group. Option 2: Sensed information of agents affect only the actions of that agent, fitness measured as a group. Option 4: Sensed information of agents affect only the actions of that agent, fitness measured individually. Option 4: Sensed information and actions of all agents are identical, fitness measured as a group.

The problem with allowing a group of agents to be represented as a single genome, as in option 1, is that implies a strong communication between the agents, which is opposed to the nature of swarm robotics and often causes problems with large multi-agent systems. It also increases the length of the genome with the number of agents. Therefore the size of the search space is relative to the number of agents. The problem of increasing the number of agents is present in options 2 and 3 but in the number of search spaces that are present rather than the complexity of the search space itself. Therefore, these would be easier to process in parallel. Miconi (2003) compared cooperative co-evolution (option 2) with population orientated genetic algorithms (option 1). They found that replacing apparently less fit members of the group, as in option 2, could change the whole dynamic of the group causing an overall less fit team.

This change was apparent due to the continual sharp decreases that could be seen in the fitness values at regular intervals, suggesting that option 1 was better. However, the group numbers were relatively small; seven and fifteen agents.

Yong and Miikkulainen (2001) also explored these different methods in a cooperative tasks where predators much act together in order to capture a prey which moves at the same speed. They found that evolving the behaviours of the individuals separately whilst testing their fitness as a group (option 2) produced the best results. They used three predators. Bassett and Jong (2000, p. 158) also found that "only when the team was evaluated as a whole did cooperation occur." Option 3, was found by Yong and Miikkulainen (2001) to produce selfish agents which did not cooperate with each other. In this case all agents would chase after the prey but as they acted in the same selfishly optimal way were never capable of catching it.

The consideration of these types of group GA are more important when evolving heterogeneous teams as they aim to produce different types of behaviours in different members of the group.

A more general approach (option 4) where each agent of the swarm acts the same and is therefore homogeneous does not require this additional complication. Instead each candidate solution is mapped to every member of the swarm. Numerous different swarms have their fitness measured and compared against each other, their offspring producing a new swarm of identical agents. An example of this type of homogenous group GA can be seen in research by Reynolds (1993) where agents had to avoid both predators and obstacles and failing to do so would result in destruction. It should be considered that there is potentially no reason that this type of homogenous GA could not produce a group or swarm which behave differently in different scenarios and therefore still be cooperating with each other by completing different tasks. However, it is justifiable to consider that the behaviours needed and therefore the genome required would be far more complicated, although this is not confirmed.

More generally Potter, Meeden and Schultz (2001, p. 1342) suggest that the "(coevolving) a team of homogeneous agents can take much less time, since each evaluation of an individual in the population goes towards all individuals' progress and the search space is smaller. In a heterogeneous group, the available CPU time during evolution must be divided among the different skill sets."

Bahçeci and Şahin (2005) completed a series of four experiments in order to give guidance on future evolutionary behaviours for swarm robotics. These are the fitness combining method (which method of fitness to use); varying the number of runs per controller and the simulation duration whilst maintaining the total number of simulation steps; varying the number of runs and the number of generations whilst also maintaining the total number of simulation steps; and finally the set-up of the experiment considering how many agents are used and how effective these evolved solutions are for different size set-ups. In summary they give the following advice:

- It is preferential to use less optimistic functions, such as median or minimum of performance values rather than maximum performance values.
- Maximising the number of runs per controller should be prioritised over the number of time-steps in each run when a trade-off is considered.
- It is difficult to determine the number of runs per controller and the number of generations. It is best to allow the evolution to run for many generations initially to see when the performance reaches a reasonable level.
- Rather than running a simulation with multiple set-ups it is more beneficial to reduce the drawbacks of outlying results by repeating more runs per controller.

These guidelines were used, in part, to help design the GA which was used for determining the state-behaviour rule relationships for the hBots in the cooperative object recognition task, section 9.1.

4.3 Physical Multi-Agent Platforms

There are numerous multi-agent platforms that have been utilised for research into swarm robotics and other multi-agent problems. A number of these platforms are detailed here.

4.3.1 E-Puck

The e-puck robot was designed as an educational tool for teaching robotics (Mondada et al., 2009; Guyot et al. 2011). Its basic configuration consists of a range of sensors and actuators. The sensors include eight IR proximity sensors, a 3D accelerometer, the microphones, and CMOS camera. The actuators include, two stepper motors to control the wheels, a speaker, and LEDs for communication between the agents. According to

the e-pucks main website there have been over 65 publications which have used the e-puck robot up till the end of 2010 (E-Puck, 2010). Liu and Winfield (2011, p66) developed a Linux extension board which is compatible with the e-pucks mother board which provided "improved computation, memory and communications" and "a flexible control architecture, that allows us to develop and test more demanding embedded robot controllers and swarm algorithms". This has allowed them to carry out research in the investigation of social learning in a robot collective where robots imitate each other's behaviours (Winfield and Griffiths, 2010).

4.3.2 S-Bot

The swarm-bots project developed the s-bot, capable of self-assembly, physical cooperation and coordination, for tasks inspired by social insects for decentralised and distributed control (Dorigo et al. 2005). The s-bots are capable of sensing their surroundings with a custom camera and spherical mirror as well as multiple infrared sensors, microphones, accelerometers and humidity and temperature sensors. The s-bot is capable of displaying different states through a coloured ring of LEDs which can be sensed by the other s-bots. Using a gripper the s-bots interact with the physical world around them and are capable of connecting to each other to form a swarm-bot. The research that has been carried out with the s-bot and swarm-bot include but are not limited to: cooperative hole avoidance (Triannia, Nolfi and Dorigo, 2006), pattern formation (Şahin 2002), self-assembly (Groß et al., 2005) and cooperative transport (Dorigo et al., 2005).

4.3.3 Khepera

The Khepera robot was developed by K-Team (Mondada, Franzi, Guignard, 1999). It is a two-wheeled robot, with eight infrared sensors capable of determine light-level and the proximity of nearby robots. It also has a modular design which allows for the addition of different components, such as gripper arm or two-dimensional vision sensor Harlan, Levine and McClarigan, (2000) discuss it as a suitable platform for introducing undergraduates to robotics and introduce a development platform for it. The research it has been used to conduct includes different learning methods for robots (Stolzmann and Butz, 2000; Sehad and Touzet, 1994). Expansions to the system to include blue-tooth communication have also been developed (Grosseschallau, Witkowski and Rückert 2005).

4.3.4 Miabot

The Merlin Miabot Pro is a two-wheeled robot with two-way blue tooth communication. It has an expansion port that allows for the addition of a programmable gripper and an LED starboard, which improves overhead tracking. One of the main uses of the Miabot Pro has been robot football (Robinson et al., 2004). This system is available for research at Nottingham Trent University with a 3x3 metre arena and 20 Miabot Pros all enabled with LED starboards for tracking. The tracking system consists of four overhead cameras whose images are stitched together to give the relative location and orientation of the agents. This allows the user to simulate different types of sensor capabilities without necessarily having to change the hardware whilst still allowing for true physical collisions and interactions between the robots. Baxter et al. (2006; 2007) have used the Miabot Pro to investigate multi-robot search and rescue strategies through the use of shared information about potential fields from obstacles.

4.3.5 Molecubes

The Molecube is a cube shaped modular robotic system (Zykov, Chan and Lipson, 2007) Each of the cubes is divided into two triangular pyramid halves which can rotate relative to each other. The Molecubes attach to each other with an interference fit. A mechanical interface of pins allows communication between connected modules. Each module can sense the position and temperature of its servo, and the modules orientation relative to attached modules. It is possible to have a robot system built from multiple Molecubes replicate itself from other Molecubes (Zykov et al., 2007). There is also research that simulates the Molecube to find behaviours of self-replication through evolutionary methods (Studer and Lipson, 2006; Zykov et al., 2007). As well as mechanical gripper and wheel cubes, passive components have being developed for the Molecube to expand its capabilities and functions, these include: cardan cores, hinges, cubes, rod sockets and feet (Zykov et al., 2008).

4.3.6 ATRON

The ATRON is a modular robotic system, each module is approximate sphere which is split through its equator into two parts (Østergaard et al., 2006). These two hemispheres can rotate relative to each other, allowing the modules to switch which other modules they are connected to. The modules connect to each other with three hooks which a mechanically controlled, although this is "power inefficient and relatively slow" it is "power neutral while maintaining a connection" (Jörgensen, Østergaard and Lund, 2004,

p. 2070). A five single slip ring allows communication between the two hemispheres, one which manages power, the other hemisphere contains the accelerometer, rotation actuator and main processor. Communication between neighbouring modules is handled by four infer-red sensors on each hemisphere. The ATRON has been used for research involving: artificial evolution of control (Østergaard and Lund, 2003), a method for a distributed cluster walk (Østergaard and Lund, 2004).

4.3.7 Catom

The Catom system is designed as programmable matter for the purpose of Claytronics, a process where shapes can be built from many individual elements, called catoms. These catoms each have a series of electromagnets that allow them to connect to each other in different formations(Goldstein and Mowry, 2004). Different shapes can be formed by the movement of catoms. An example is discussed (Goldstein, Campbell and Mowry, 2005) where a hole inside an otherwise solid lattice structure of catoms can move around changing the surface's shape. The shape can also be expanded by adding these holes and reduced by removing them. The end goal of the research is to have a system where by a synthetic reality can be built allowing users to interact with it in natural ways, without the use of additional aids, like virtual reality headsets (Goldstein, Campbell and Mowry, 2005).

4.3.8 Miche

Each robot in the Miche system is a module which can be combined into different arrangements. To do this the robots start in a block with physical connections to each other. By releasing certain bonds between the modules, modules begin to fall away leaving the desired shape. Gilpin et al. (2008) liken this to forming a sculpture from a block of material, where unwanted pieces are removed piece by piece. Each module is completely autonomous with its own power supply, processing capabilities, communication interfaces, and actuators (Giplin et al., 2008).

4.3.9 Robot Pebbles

The Robot Pebbles platform is one where each individual module is a cube which is capable of attaching to each other on four of their sides to form different shapes with a uniform height of one cube. To attach to each other they each have four EP magnets, which are also responsible for power transfer, and communication (Gilpin, Knaian and Rus, 2010). The research that has been carried out with the Robot Pebble considers the formation of shapes through subtraction (Gilpin, Knaian and Rus, 2010), the formation

of multiple shapes from the same starting block of robots (Gilpin, Koyanangi and Rus, 2011) and the duplication of inert shapes with a resolution equal to the size of the module (Gilpin and Rus, 2012a).

4.3.10 Systems Review

A review of these systems object recognition capabilities and their ability to communicate between agents is described in table 4.1.

System	Object Recognition Capabilities	Communication between Agents
E-Puck	IR Proximity Sensors; CMOS camera	Speaker; LEDs
S-Bot	IR Proximity Sensors; custom camera and spherical mirror	Speaker; LEDs
Khepera	IR Proximity Sensors	Bluetooth
Miabot	The robot is modifiable and additional sensors can be added. (Can be simulated through the use of overhead tracking).	The robot is modifiable and additional communications can be added. (Can be simulated through the use of overhead tracking).
Molecubes	Can only detect other neighbouring agents.	Eight interlocking pairs of ABS pins and sockets.
ATRON	Can only detect other neighbouring agents.	Infer-red
Catom	Can only detect other agents.	Linx 900Mhz Radio Transmitter and Reciever
Miche	Can only detect other neighbouring agents.	Infer-red
Robot Pebbles	Can only detect other neighbouring agents.	Magnetic Interface

Table 4.1: Object recognition and communication capabilities of different multi-agent robot systems.

4.4 Simulated Multi-Agent Platforms

Perhaps due to the cheaper nature of producing a simulation over a physical robot there are many available types of platforms that have been used for multi-agent research, far too many to list all of them here when they have so much in common with each other. Harris and Conrad (2011) provide a more in-depth overview of popular robotics simulators and toolkits not exclusively for multi-agent control. In the following paragraphs a number of examples are discussed.

In general having a simulation of a swarm of agents on a computer gives a lot more freedom. Experiments can be run in parallel on a single computer or over a number of computers and there is no down time needed between experiments for recharging batteries and physical maintenance. This means that it is possible to gather more data in a shorter amount of time. There are also less issues with the resources of the experiment, for example it is possible to add more robots without any additional financial cost making the experimentation less restricted to what exists in the laboratory. In general for swarm robotic and multi-agent system simulations there is a gradient between those that model existing hardware platforms and those that are more abstract in execution. The systems that model existing hardware attempt to mimic the behaviours of the physical robots in order to alleviate the time considerations but resulting in strategies that can be transferred direct to the physical robot platforms for testing. Constructing these types of simulators is more time consuming than those of the abstract models but provide data similar to that of the actual hardware. For example the Webot is a simulation and prototyping platform designed to work with commercially existing hardware (Michel, 2004). Guyot et al. (2011) use a combination of the Webots simulation platform with the e-puck for teaching robotics. The Player/Stage platform and simulation system also falls into this category of simulation (Gerky, Vaughan and Howard, 2003). The Player is designed for the control of physical robots and does this through communication with the robot, reading the sensors and then controlling the actuators. The Stage is designed so the Player aspect of the system can be simulated as if it was using real robots allowing for simple transfer of strategies from simulations and physical robots.

There is a middle ground between the physical and computer simulated platforms where the simulations are designed for developing and testing strategies without having a specific hardware model in mind. An example of this type is MASON. MASON is a

highly variable Java based simulation platform designed with multi-agent systems in mind (Luke et al., 2005). The MASON platform has been used to research the use of moveable beacons, which act as an approximation of pheromones to identify the shortest routes between objectives (Hrolenok et al., 2010). It has also been used for research in training agents to complete different tasks by building up numerous behaviours on top of each other (Luke and Zuparo, 2010; Sullivan and Luke, 2011). The benefit of this is not having to limit the control schemes to currently viable platforms whilst retaining the majority of the behaviours and nuances which make the physical platforms. The most abstract simulation that is in common use for the study of multi-agent behaviours is that of a grid-world simulation.

4.4.1 Cellular Robot Simulations

The earliest recorded work on swarm robotics was completed using a grid-world simulation, although under the name Cellular Robotics (Beni, 1998). According to Beni (2005) he was inspired by Wolfram's collective research into cellular automata in 'A New Kind of Science' (Wolfram, 2002).

A cellular automaton consists of a sequence of sites carrying a discrete set of values which can be arranged on any regular lattice. "The value at each site evolves deterministically with time according to a set of definite rules involving the values of its nearest neighbours" (Wolfram, 1982). Although this is a relatively simple process it can be used to model physical, computational and biological systems. Further work specifically with two-dimensional cellular automaton was carried out (Packard and Wolfram, 1985). The main change between this research and Beni's (1998) initial swarm robotic research was that the agents could move through the system from cell to cell rather than the cells simply flipping states. The same sort of relationships were important; what is the agent doing, what is happening near the agent and what will the agent do next based on these previous factors. All of these issues are closely tied to swarm robotics research.

In these grid-world simulations any agent modelled in the system takes up a single cell and moves discretely from one cell to another. This is opposed to more realistic simulations where discrete grids are also used but the distance between grid nodes is far smaller than the size of the robot, therefore their movement is smoother and the distance travelled in a single step is smaller (Winfield, 2005). However, despite the

opportunity to build simulations in this manner lattice based simulation have been used for multi-agent and swarm robotics research over the last twenty years:

- Lucarini et al.(1993) uses a grid of 100x100 cells to model and research the capabilities of a group of robots to navigate between two points in an unknown area with obstacles.

- Şahin et al. (2002) use discrete movement between the nodes of a hexagonal grid to simulate the robots' movement. This simplification was to make the "implementation of connecting and disconnecting of the s-bots [robots] easier" as the research was interested in linking robots together to form new robotic structures.

- Matarić, Sukhatme and Østergaard (2003) use a grid-world to initially implement a simplified version of a multi-robot handling task for finding alarmed areas and fixing problems. This was then moved onto a physical system.

- Shen, Will and Galstyan (2004) discuss a distributed control method for robot swarming behaviours and self-organization which they term the Digital Hormone Model. In this they model the behaviours of the swarm using a discrete grid measuring 100x100 cells.

- Hsiang et al. (2004) explore dispersion algorithms for robotic swarms. The simulations they use are based on a discrete grid.

- Engels and Kamphans (2006) discuss the NP-Hard (non-deterministic polynomial-time hard) Randolph's Robot Game which is built on a grid-world.

- Shiloni, Agmon and Kaminka (2009) describe the difference between highly capable 'elephant' type robots and very simplistic 'ant' type robots using a discrete grid.

- Chouhan and Niyogi (2012) illustrate the importance of communication whilst planning multi-agent actions in a grid-world containing obstacles.

The main advantage of using this type of grid-world is the simplicity of the programming especially for testing initial concepts and strategies. This is especially true in the case of modular robotics. Once agents are connected to each other into a lattice

formation, they may communicate and solve problems in a way that is similar to agents in a simulated grid-world.

4.4.2 Different Shaped Cells for Grid-Worlds

The vast majority of grid-world simulations are built on square lattices. This is most likely due to the simplicity of representing the system both mathematically and visually on a computer screen which is also constructed from a square lattice. However, there are other possible lattices possible which have uniform cells where all the internal angles of the individual cells are equal. In total there are three types; triangular, square and hexagonal.

In a square grid-world each cell has eight cells that could be considered neighbours, four of them side contacts and four of them corner contacts. The distance between the centre points of cells varies depending on if they are side contacts or corner contacts. There are two ways of modelling single cell movement within a square grid-world.

- Allow the agents to only move to the four neighbouring side contact cells, as in a Von Newmann Neighbourhood.

- Allow the agent to move to the eight neighbouring cells, sides and corner contacts, as in a Moore Neighbourhood.

Using these two methods the shortest distance to travel between two cells discretely can be found. Figure 4.3 shows the least number of moves required for agents that can move discretely from cell to cell through either the four side contacts only or for agents that can travel to any of their eight neighbouring cells.

				4				
			4	3	4			
		4	3	2	3	4		
	4	3	2	1	2	3	4	
4	3	2	1	0	1	2	3	4
	4	3	2	1	2	3	4	
		4	3	2	3	4		
			4	3	4			
				4				

Von Newmann Neighbourhood

4	4	4	4	4	4	4	4	4
4	3	3	3	3	3	3	3	4
4	3	2	2	2	2	2	3	4
4	3	2	1	1	1	2	3	4
4	3	2	1	0	1	2	3	4
4	3	2	1	1	1	2	3	4
4	3	2	2	2	2	2	3	4
4	3	3	3	3	3	3	3	4
4	4	4	4	4	4	4	4	4

Moore Neighbourhood

Figure 4.3: The minimum distance of discrete travel between cells on a square grid-world for a Von Newmann Neighbourhood and a Moore Neighbourhood.

These square grid systems produce patterns of concentric squares, where one is at a 45 degree angle to the other. The distance travelled to the corner points of these squares is equal to the distance to the centre of the sides of the squares. However, the distance from the centre of a square to the corner is approximately 1.414 (or $\sqrt{2}$) times longer than the apothem, the distance from the centre to the centre of an edge.

In a hexagonal grid each cell only has six side contacts there are no corner only contacts. There is only one way of modelling single cell movement from cell to neighbouring cell.

- Allow the agents to move to any of the six neighbouring side cells.

The minimum distance an agent can travel between any two cells can be found as shown in Figure 4.4.

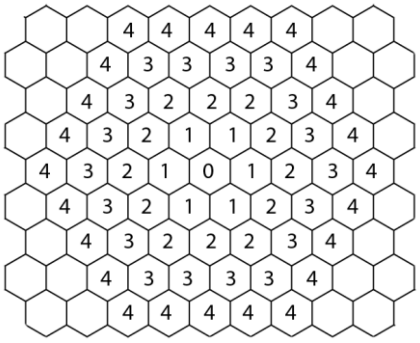


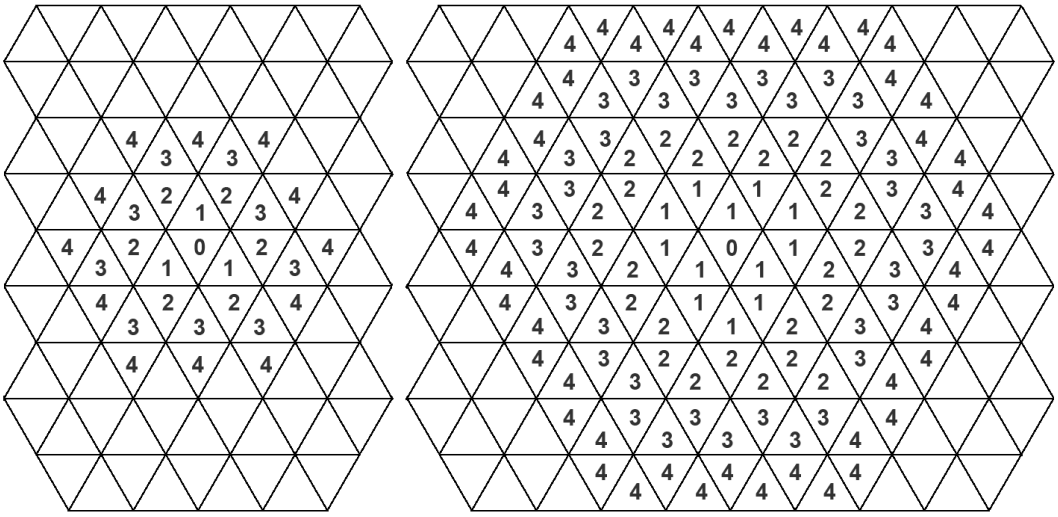
Figure 4.4: The minimum distance of discrete travel between cells on a hexagonal grid-world.

This hexagonal grid-world system produces patterns of concentric hexagons. Although the distance from the centre of a hexagon to the corner is approximately 1.155 (or $\frac{1}{cos30^{\circ}}$) times larger than that of the distance from the centre to the centre of a side, this is smaller than the value (1.414) found for the square grid. This is because a hexagon is a better approximation of a circle than a square. The reason that a better approximation of a circle is beneficial is that it gives a more realistic representation of movement.

In a triangular grid-world each cell has twelve potential neighbours, three of them side contacts and nine of them corner contacts. As with the square grid-world there are two ways of modelling single cell movement within a square grid-world.

- Allow the agents to only move to the three neighbouring side contact cells.
- Allow the agent to move to the twelve neighbouring cells, sides and corner contacts.

Using these two methods the shortest distance to travel between two cells discretely can be found. Figure 4.5 shows the least number of moves required for agents that can move discretely from cell to cell through either the three side contacts only or for agents that can travel to any of their twelve neighbouring cells.



Neighbourhood size 3 Neighbourhood size 12

Figure 4.5: The minimum distance of discrete travel for a triangular grid-world where the neighbourhood is either the 3 side contacts or the 12 side and point contacts.

For the triangular grid world with a neighbourhood size of three, mapping the centre points of each cell of the same magnitude gives three patterns. In the case where the value is one it gives a triangle, in all other odd number cases it gives a six sided shape with uneven sides, and in all even number cases it gives a hexagon. Where the neighbourhood size is twelve, the shape produced is an approximately a six sided shape. Regarding the edges more closely gives a jagged line rather than a continuous one. This shape is the same as the odd number cases for the neighbourhood size three triangular lattice. This shape also gives the largest discrepancy between direct travel from the centre to the most distant corner and the centre of one if the sides. The difference is equal to the ratio of the base length to base height of a equilateral triangle, giving the same ratio as the hexagonal gird-world, 1.155. However, this is an approximation due to the irregular shape produced and the value would be marginally higher, considering the precise furthest point away.

Overall, considering both regularity and the ratio of the distances to the edges and the corners, the hexagonal model gives the best approximation of a circle out of the three

grid systems. For this research a hexagonal lattice was chosen due to the geometric relationships between neighbouring cells in the grid.

4.5 Summary

Research into cooperative object recognition has been varied. A number of the systems use the act of cooperation to increase the efficiency or robustness of their individual object recognition. In other systems the agents must cooperate in order to recognise the objects they are trying to identify. There is currently no research that utilises multi-agent systems to distinguish between different objects through their shape alone where the agents are mobile and have limited sensor and communication capabilities.

The use of GA to evolve the required behaviours of a multi-agent system was also investigated. This method would allow the agents to adapt to different object identification tasks without minimal input from a third party. It may also provide solutions that are more efficient but less obvious in terms of development.

A survey of the different methods of simulation, both physical and computational was carried out. This survey aided in the identification of a suitable system for the current research project. The advantages of using a computational simulation currently outweigh that of using a physical platform for this project since a simulation of a swarm on a computer gives a lot more freedom. There is no down time needed between experiments for recharging batteries and physical maintenance making it possible to gather more data in a shorter amount of time. It is possible to add more robots without any additional financial cost and in addition the benefits or otherwise of using GA can be explored within a reasonable time frame.

Chapter 5: Swarm Simulation Methodology

This chapter outlines the cooperative object recognition task that the agents aimed to complete. The Simplified Hexagonal Model is described as a novel platform for simulating swarm cooperative object recognition along with the motivations for and limitations of using this platform. Specific details, where they vary between individual experiments, are not included in this chapter but are included with the relevant experiments.

5.1 Cooperative Object Recognition Task

The task the swarm had to complete was to identify one of two different object shapes in a closed arena. There was an equal quantity of each object shape in the arena. One object shape was considered valid and the other invalid. In order to complete the task all of the valid object shapes needed to be acted upon by the swarm. In the case of the initial research (Chapter 6), this act involved removing the three valid objects by transporting them to the collection zone. In the later research the six valid object shapes needed to be destroyed by the agents (Chapter 8 onwards). The valid object shapes used had features in common with the invalid object shapes and features that distinguished one from the other. It was rare for an individual agent to distinguish a valid object shape from an invalid object shape alone.

5.2 Choice of Platform

From the review of different simulation methods it was determined to use a hexagonal grid-world simulation. This choice was made to allow the focus of the research to be on the strategies involved for swarm robotic cooperation without the need to consider a specific application and physical platform. The main advantages of this choice over a physical platform are the ability to run more concurrent tests, have less down time between tests and have an arena space large enough to contain numerous objects that can be identified with cooperating agents. Each of these is important especially in the later stages of experimentation where a GA is utilised, which requires numerous repeats of tests.

The choice of a discrete grid-world simulation for the simulation allowed for a clear distinction between agents that are neighbouring each other or object shapes and those

that are not. Using this approach, agents have a distinct understanding of their local surroundings. There are cells that they touch and cells that they do not. Although this choice limited realism of the interactions, as there was no noise or physical interaction, it provided a simplified model in which to examine, identify and analyse the behaviours of the agents and the results of these experiments will provide guidance for future swarm cooperative object recognition research.

5.2.1 Processing Programming Language

The choice was made to build a custom platform, in the Processing language, for the research experimentation despite the availability of the numerous simulators currently available. Building a custom simulation platform allowed for more control over the entire system although there was a limitation in there currently being no other multi-agent system tested on it to compare results with. Processing (Processing, n.d.) is an object-orientated 2D and 3D application programming interface built on the Java language, with a focus on visualisation. These elements made Processing a suitable choice for this research, since, object shapes and agents could be built as objects within the system, Java is a robust and well maintained language and provided a way to watch the movements and reactions of the agents.

5.3 Simplified Hexagonal Model

The Simplified Hexagonal Model (SHM) platform was designed specifically for completing the research experimentation. It has the advantage of providing a platform that could be adapted for other multi-agent problems which would utilise a bounded two-dimensional hexagonal grid-world. Each cell can be: an object or part of an object; a hBot (an agent); a boundary region; or an empty arena space. Time in the SHM is measured in time-steps, where each time-step is a complete cycle of the main program, detailed in section 5.6.

5.3.1 The Arena

In the SHM the arena is a hexagonal shaped lattice where each of the six sides of the hexagonal arena has twenty-one cells. Immediately outside these cells is a boundary that the hBots cannot enter or pass. For the initial experiments only (Chapter 6) a hexagonal collection zone was added to the centre of the arena.

5.3.2 Object Shapes

Object shapes are constructed in the SHM by grouping a number of neighbouring object cells together. The smallest feasible object shape is the same size as a single cell and only solid object shapes are considered. In all of these cooperative object recognition experimentation there were two object shape types used. Within the SHM one of these types is classed as valid and is the object shape the swarm is required to react with. The other type is invalid and acts as a distraction to the agents. Object shapes were initially placed such that they did not touch each other and hBots in contact with an object shape would not neighbour any hBot in contact with another object shape.

In the initial cooperative object recognition experimentation (Chapter 6) the object shapes were approximations of triangles and hexagons, as in figure 5.1. The validity of the object shapes were varied. In total six object shapes were used, three of each type. Any invalid object shapes moved out of the arena by the agents, experiment specific, were deleted. Valid, hexagonal object shapes moved into the collection zone were considered successfully collected. To complete the task all three of the hexagonal object shapes required collecting.

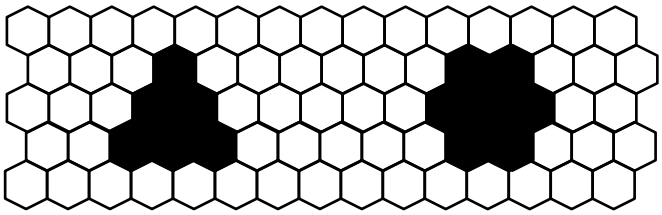


Figure 5.1: Triangular and Hexagonal object shapes constructed from neighbouring object cells.

For the later experimentation (Chapter 8 onwards) a range of object shapes each with four object cells were used. All object shapes with four object cells are shown in figure 5.2.

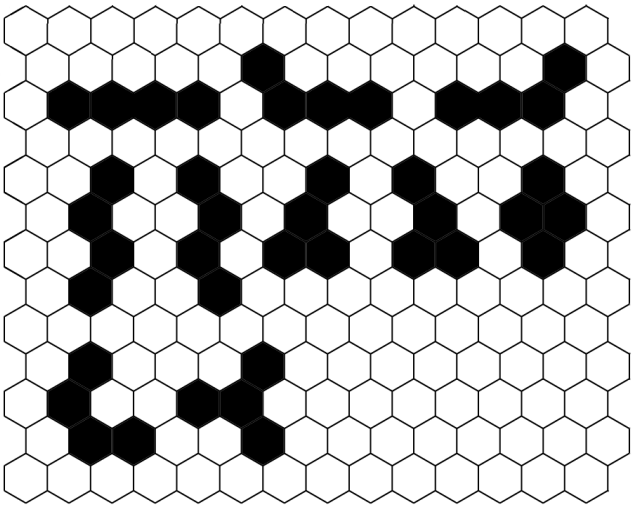


Figure 5.2: All object shapes created from four object cells

5.3.2.1 Data-Chain

An object shape can be described without considering rotation or location by considering the contours of its boundary region. Each cell that neighbours an object shape is given a value determined by how many cells it touches, as per Figure 5.3, forming what is termed a data-chain for the object shape. Traversing the data-chain clockwise produces a sequence of numbers which will be always written using the sequence that is first lexicographically of all the cycles from all starting points and will itself be referred to as the data-chain of the object shape. For example the data-chains for the triangle and hexagon shown are {1,1,2,2,1,1,2,2,1,1,2,2} and {1,2,1,2,1,2,1,2,1,2,1,2}.

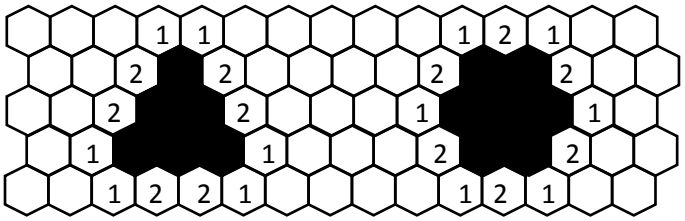


Figure 5.3: Surrounding cells show the number of object cells they are in contact with for both the triangular and hexagonal object shapes. This values are used to form the data-chains for the object shapes: Triangle, {1,1,2,2,1,1,2,2,1,1,2,2} and Hexagon, {1,2,1,2,1,2,1,2,1,2,1,2}.

The data-chains is derived from the same negative space around the object shape that the hBots inhabit and contain the same information the hBots can sense and gather. By comparing data-chains against each other the differences between object shapes can be

determined. The methods for discovering and describing object shapes as well as their differences are discussed in more detail in Chapter 7 and Chapter 8.

5.4 The hBots

An hBot swarm within the SHM is homogeneous, that is all agents have identical characteristics in terms of their capabilities, control architecture and behaviour. As such the hBots are considered to be anonymous and therefore unable to distinguish between specific agent identities and each agent is interchangeable with any other agent. These choices meant that no specialist agents were required for any part of the cooperative object recognition task. The hBots also do not have a common coordinate system and are not aware of their position relative to the arena coordinate system. Therefore there was no requirement for beacons or any other kind of positioning system to be utilised increasing the amount of potential applications that the system could be used for in future research.

5.4.1 Sensor Capability

The hBots can sense the number of object cells of an object shape that it is neighbouring. This gives the hBots an impression of how convex or concave the part of the object shape they are neighbouring is. Neighbouring one, two or three cells represents a convex, straight, or concaved section of the object shape, respectively. These agents can determine the state of any other hBot that it is neighbouring. A cell is classified as neighbouring a hBot when it is one of the six cells that immediately surround the hBot, as shown in figure 5.4, this represents the sensor range of the hBot.

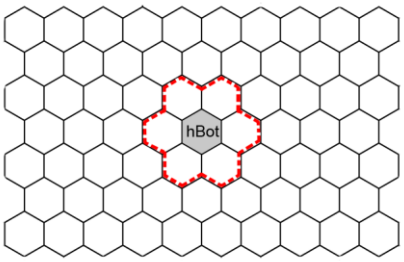


Figure 5.4: The hBot can determine the number of object cells and the states of hBots within its sensor range, the size cells that immediately neighbour it.

This means that the perception of the hBots is considerably smaller than the arena they inhabit. The hBots' sensor range is also too small for them to appreciate the entirety of

the object shapes used in the experimentation. This means that they individually have knowledge of their local surroundings but are not capable of perceiving the whole of an object shape in order to distinguish it from another object shape.

At this stage of the research the sensor capabilities of the hBots are considered to be perfect, the hBots will not mistake one state for another state, or miscount the number of object cells they neighbour.

5.4.2 Communication

The hBots communicate their current state utilising a local broadcast. The range of the broadcast is one cell. Therefore the transfer of current state information is only passable to hBots that are neighbouring each other. There is no direct feedback between the agents to confirm receipt of data, however the system was modelled without noise so communication was considered to be perfect. No other information can be communicated by the hBots, therefore they are unable to coordinate their actions in any manner.

5.4.3 Random Movement

For each time-step hBots move with equal probability to any of their neighbouring six cells, unless one of these six cells contains an object cell or a cell on the boundary of the arena. If at the moment the hBot tries to move to a cell which already contains another hBot they remain in the cell they are currently in.

The behaviour of a hBot neighbouring an object shape changes slightly. When next to an object cell the hBot generally stays still. This is to increase the amount of interaction around the object shape. However, the hBot has a probability, adjusted for each test, of moving away from that cell. This reduces the chance of stagnation by allowing movement away from the object shape. If the hBots simply remained stationary, they could be divided amongst parts of multiple object shapes without enough neighbouring agents interacting to distinguish between those object shapes.

Although the hBots computational behaviours are synchronised each of their positions are updated one at a time, which means they can never occupy the same cell. The order that they move in is randomised each time they perform the move action. This choice was made to reduce any effects that could occur through them always moving in the same order.

5.4.4 Computation and Recall

The hBots were modelled as finite state machines, with a maximum total of 265 states, over four state-levels. The state-level of the hBot represents how much information it has gathered about its local area through the interaction of it and its directly neighbouring hBots. The higher the state-level the more information the agent has about the shape. The process of state-relationships, how they change between states and which states are possible, are described in more detail in section 5.5.

The hBots are not capable of gathering enough information to map an entire object shape, they can only consider part of it. This limitation also means that the hBots are not capable of knowing the entire contents of the arena at any time.

5.5 hBot Cooperative Object Recognition

The hBots cooperate to distinguish the difference between object shapes. As the hBots inhabit the same negative space around the shape that the data-chain is derived from they use this same information to discern the object shapes from one another, although this information is gathered in a cooperative manner. The current state-level of the hBot indicates how much knowledge it has about the object shape, whilst the state describes this knowledge.

- A hBot at state-level 0 is not in contact with an object shape and therefore does not know anything about the object shape

- A hBot at state-level 1 knows the number of object cells it is neighbouring: one, two, or three. This knowledge is represented by states 1, 2 and 3 respectively. This value could theoretically be between one and six, however in this research the object shapes were limited as to only allow situations where the values one, two and three occur, the reason for this are discussed in section 5.7.4.

- A hBot at state-level 2 knows as much as three individual agents, as it knows its previous state-level 1 state and the states of its neighbours. Represented by states 4 – 21.

- A hBot at state-level 3 knows as much as five individual agents, through the changing states of its neighbours in reaction to their neighbours. Represented by states 22 – 264.

A hBot can only increase its state-level and therefore its knowledge about the object shape it is trying to identify when neighbouring two other agents at the same state-level or higher. Therefore an agent is incapable of assessing an object shape on its own. The knowledge the hBots have represent the contours of the boundary of the object shape they are trying to identify. As the object shapes being distinguished have features different from each other certain states are only achievable when an agent is neighbouring that object shape. Using this method it is possible for the hBots to distinguish between two different object shapes.

Theoretically if there were enough hBots and states two object shapes are distinguishable from one another. This is because the object shapes' data-chains are distinguishable, except where there is a tunnel (section 7.5.3), and the hBots use the same information to identify the difference between them. However, the hBots cannot distinguish between object shapes that are symmetrical to each other (section 5.7.6). Currently as there are only three state-levels and 264 states the object shapes the hBots can distinguish between are limited, this is discussed further in section 5.7.5.

5.5.1 State-Relationships

In this research the states of a hBot and its neighbours are described in the following manner:

[own state][state of neighbour with lowest state][state of neighbour with highest state]

All the possible patterns of three neighbouring hBots currently in state-level 1 and their resulting new state are shown in table 5.1. It is only the centre hBot represented by the first value of the three that changes state. However, the other hBots would also change state if they were neighbouring two hBots themselves.

Own State (1st Level)	Neighbour State (Lowest)	Neighbour State (Highest)	Shorthand Description	New State (2nd Level)
1	1	1	[1][1][1]	4
1	1	2	[1][1][2]	5
1	1	3	[1][1][3]	6
1	2	2	[1][2][2]	7
1	2	3	[1][2][3]	8
1	3	3	[1][3][3]	9
2	1	1	[2][1][1]	10
2	1	2	[2][1][2]	11
2	1	3	[2][1][3]	12
2	2	2	[2][2][2]	13
2	2	3	[2][2][3]	14
2	3	3	[2][3][3]	15
3	1	1	[3][1][1]	16
3	1	2	[3][1][2]	17
3	1	3	[3][1][3]	18
3	2	2	[3][2][2]	19
3	2	3	[3][2][3]	20
3	3	3	[3][3][3]	21

Table 5.1: All possible patterns of three neighbouring hBots currently in 1st level states resulting in 2nd level states as described.

Although this simplifies the hBots as they do not have to determine the position of their neighbours there is a limitation to this method of cooperation in that symmetrical sections of an object shape will appear the same and therefore so will object shapes symmetrical to each other, there is further discussion of this in section 5.7.6.

5.5.2 State-Behaviours

For each state the hBots can reach their behaviour changes dependant on the relationship between the two object shapes that are being distinguished from each other. The state-behaviour of the hBot is determined by whether or not that state is achievable for one, both or neither of the object shapes. The number of possible behaviours that the hBots act upon vary between the initial experiments and the later experiments, but can generally be described as staying still with a given probability or acting on the object shape by moving or destroying it. If the hBots have been given the correct state-behaviour rules, they will never remove the invalid object shape. The exact details for each set of experiments are covered in section 6.1.3 and section 8.4.

5.5.3 Possible State Neighbours

Due to the relationship of the position of a hBot to their neighbouring hBots' there are certain patterns of state-level 2 states that are not possible. For example, given a hBot is in state 4 according to the state-relationship rules its neighbours must have both been in state 1, table 5.1. Given that this is true, a hBot in state 4 can only neighbour hBots at state-level 2 which were also originally in state 1 and also had at least one neighbour in state 1. Where this neighbour in state 1 represented the first hBot now in state 4. This means its neighbouring hBots could potentially have been in states 4, 5 or 6 at state-level 2. Meaning that it was impossible for a hBot in state 4 to neighbour a hBot in states 7-21.

Following this logic of possible neighbouring states through the rest of the states it was possible to determine all the state relationships at both state-level 1 and state-level 2. Figure 5.5 shows which states it was possible to be neighbouring each other, where possible state relationships are noted in grey. Given these restrictions the possible patterns of three neighbouring hBots are shown in table 5.2, with the new higher state-level state the centre hBot, noted in the left most bracket set, would change to. These considerations result in there being a total of 264 states across state-levels 1, 2 and 3, not including state 0 at state-level 0.

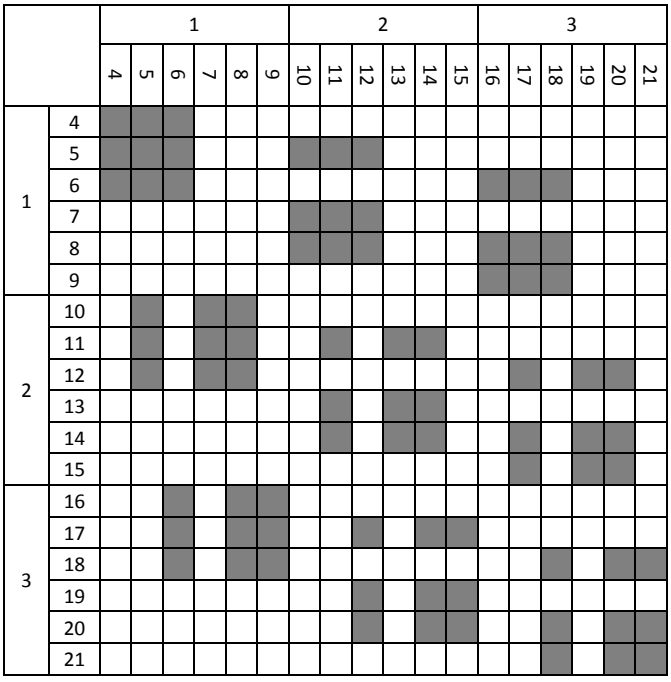


Figure 5.5: The possible neighbouring states for the first 21 states. Where a grey box is shown it is possible for hBots of that state to be neighbouring each other and therefore state relationships which contain those states need to be considered for possible higher level states.

[1][1][1]	4	[6][6][16]	61	[11][7][13]	118	[14][20][20]	177	[19][12][15]	234
[1][1][2]	5	[6][6][17]	62	[11][7][14]	119	[15][17][17]	178	[19][14][14]	235
[1][1][3]	6	[6][6][18]	63	[11][8][8]	120	[15][17][19]	179	[19][14][15]	236
[1][2][2]	7	[6][16][16]	64	[11][8][11]	121	[15][17][20]	180	[19][15][15]	237
[1][2][3]	8	[6][16][17]	65	[11][8][13]	123	[15][19][19]	181	[18][6][18]	214
[1][3][3]	9	[6][16][18]	66	[11][8][14]	124	[15][19][20]	182	[18][6][20]	215
[2][1][1]	10	[6][17][17]	67	[11][11][11]	125	[15][20][20]	183	[18][6][21]	216
[2][1][2]	11	[6][17][18]	68	[11][11][13]	126	[16][6][6]	184	[18][8][8]	217
[2][1][3]	12	[6][18][18]	69	[11][11][14]	127	[16][6][8]	185	[18][8][9]	218
[2][2][2]	13	[7][10][10]	70	[11][13][13]	128	[16][6][9]	186	[18][8][18]	219
[2][2][3]	14	[7][10][11]	71	[11][13][14]	129	[16][8][8]	187	[18][8][20]	220
[2][3][3]	15	[7][10][12]	72	[11][14][14]	130	[16][8][9]	188	[18][8][21]	221
[3][1][1]	16	[7][11][11]	73	[12][5][5]	131	[16][9][9]	189	[18][9][9]	222
[3][1][2]	17	[7][11][12]	74	[12][5][7]	132	[17][6][6]	190	[18][9][18]	223
[3][1][3]	18	[7][12][12]	75	[12][5][8]	133	[17][6][8]	191	[18][9][20]	224
[3][2][2]	19	[8][10][10]	76	[12][5][17]	134	[17][6][9]	192	[18][9][21]	225
[3][2][3]	20	[8][10][11]	77	[12][5][19]	135	[17][6][12]	193	[18][18][18]	226
[3][3][3]	21	[8][10][12]	78	[12][5][20]	136	[17][6][14]	194	[18][18][20]	227
[4][4][4]	22	[8][10][16]	79	[12][7][7]	137	[17][6][15]	195	[18][18][21]	228
[4][4][5]	23	[8][10][17]	80	[12][7][8]	138	[17][8][8]	196	[18][20][20]	229
[4][4][6]	24	[8][10][18]	81	[12][7][17]	139	[17][8][9]	197	[18][20][21]	230
[4][5][5]	25	[8][11][11]	82	[12][7][20]	140	[17][8][12]	198	[18][21][21]	231
[4][5][6]	26	[8][11][12]	83	[12][8][8]	141	[17][8][14]	199	[19][12][12]	232
[4][6][6]	27	[8][11][16]	84	[12][8][17]	142	[17][8][15]	200	[19][12][14]	233
[5][4][4]	28	[8][11][17]	85	[12][8][19]	143	[17][9][9]	201	[19][12][15]	234
[5][4][5]	29	[8][11][18]	86	[12][8][20]	145	[17][9][12]	202	[19][14][14]	235
[5][4][6]	30	[8][12][12]	87	[12][17][17]	146	[17][9][14]	203	[19][14][15]	236
[5][4][10]	31	[8][12][16]	88	[12][17][19]	147	[17][9][15]	204	[19][15][15]	237
[5][4][11]	32	[8][12][17]	89	[12][17][20]	148	[17][12][12]	205	[20][12][12]	238
[5][4][12]	33	[8][12][18]	90	[12][19][19]	149	[17][12][14]	206	[20][12][14]	239
[5][5][5]	34	[8][16][16]	91	[12][19][20]	150	[17][12][15]	207	[20][12][15]	240
[5][5][6]	35	[8][16][17]	92	[12][20][20]	151	[17][14][14]	208	[20][12][18]	241
[5][5][10]	36	[8][16][18]	93	[13][11][11]	152	[17][14][15]	209	[20][12][20]	242
[5][5][11]	37	[8][17][17]	94	[13][11][13]	153	[17][15][15]	210	[20][12][21]	243
[5][5][12]	38	[8][17][18]	95	[13][11][14]	154	[18][6][6]	211	[20][14][14]	244
[5][6][6]	39	[8][18][18]	96	[13][13][13]	155	[18][6][8]	212	[20][14][15]	245
[5][6][10]	40	[9][16][16]	97	[13][13][14]	156	[18][6][9]	213	[20][14][18]	246
[5][6][11]	41	[9][16][17]	98	[13][14][14]	157	[18][6][18]	214	[20][14][20]	247
[5][6][12]	42	[9][16][18]	99	[14][11][11]	158	[18][6][20]	215	[20][14][21]	248
[5][10][10]	43	[9][17][17]	100	[14][11][13]	159	[18][6][21]	216	[20][15][15]	249
[5][10][11]	44	[9][17][18]	101	[14][11][14]	160	[18][8][8]	217	[20][15][18]	250
[5][10][12]	45	[9][18][18]	102	[14][11][17]	161	[18][8][9]	218	[20][15][20]	251
[5][11][11]	46	[10][5][5]	103	[14][11][19]	162	[18][8][18]	219	[20][15][21]	252
[5][11][12]	47	[10][5][7]	104	[14][11][20]	163	[18][8][20]	220	[20][18][18]	253
[5][12][12]	48	[10][5][8]	105	[14][13][13]	164	[18][8][21]	221	[20][18][20]	254
[6][4][4]	49	[10][7][7]	106	[14][13][14]	165	[18][9][9]	222	[20][18][21]	255
[6][4][5]	50	[10][7][8]	107	[14][13][17]	166	[18][9][18]	223	[20][20][20]	256
[6][4][6]	51	[10][8][8]	108	[14][13][19]	167	[18][9][20]	224	[20][20][21]	257
[6][4][16]	52	[11][5][5]	109	[14][14][14]	168	[18][9][21]	225	[20][21][21]	258
[6][4][17]	53	[11][5][7]	110	[14][14][17]	169	[18][18][18]	226	[21][18][18]	259
[6][4][18]	54	[11][5][8]	111	[14][14][19]	170	[18][18][20]	227	[21][18][20]	260
[6][5][5]	55	[11][5][11]	112	[14][14][20]	171	[18][18][21]	228	[21][18][21]	261
[6][5][6]	56	[11][5][13]	113	[14][17][17]	172	[18][20][20]	229	[21][20][20]	262
[6][5][16]	57	[11][5][14]	114	[14][17][19]	173	[18][20][21]	230	[21][20][21]	263
[6][5][17]	58	[11][7][7]	115	[14][17][20]	174	[18][21][21]	231	[21][21][21]	264
[6][5][18]	59	[11][7][8]	116	[14][19][19]	175	[19][12][12]	232		
[6][6][6]	60	[11][7][11]	117	[14][19][20]	176	[19][12][14]	233		

Table 5.2: All possible combinations of state-level 1 and state-level 2 states and the new state they lead to for the 'own state' hBot. [own state][lowest neighbour][highest neighbour]

5.6 The System

There are two major components to the overriding system, one is the SHM sub-system and the other is a supervisor sub-system The supervisor maintains and controls the SHM in such a way to preserve the integrity of the SHM. Each of the hBots is given an individual identity by the supervisor whilst within the SHM sub-system each hBot remains as an autonomous anonymous agent. This allows the supervisor sub-system: to know the states and positions of all the agents as well as the object shapes; to inform the hBots about what they are currently sensing and to display the simulation.

The hBots have a number of actions they can perform: sense; determine current state, move, and act. Act is used here as a vague term as the specific action is determined by the current experiment, and details the precise way hBots handle object shape removal or destruction, the details of which are given with the relevant experiments. In the SHM sub-system it is as if all of the hBots actions are synchronised but in actuality it is the supervisor determining when they perform each action. The supervisor ensures that all hBots, in one time-step, perform their actions in turn, one immediately after the other .

After each action is performed the contents of each cell are updated by the supervisor allowing this sub-system to keep track of the positions and states of the hBots as well as the object shapes. An overview of the hBots' actions relative to the supervisor updating the contents of the cells is given in the pseudo code in figure 5.6.

```
void main
  for (i = 0; i < number of hBots; i++){
    hBot[i] performs interaction with object;
    // this specific action is determined by the experiment
  }
  Update contents of all cells;
  // the order of the hBots move is randomised, where randOrder[] is
  // initialised with values: 0 - number of hBots.
  for (i = 0; i < number of hBots; i++){
    int posA = (int) random(noOfBots);
    int posB = (int) random(noOfBots);
    int tempA = randOrder[posA];
    int tempB = randOrder[posB];
    randOrder[posA] = tempB;
    randOrder[posB] = tempA;
  }
  // the hBots move only if the cell they attempt to move to is
  // currently available
  for (i = 0; i < number of hBots; i++){
    hBot[randOrder[i]] senses surroundings;
    hBot[randOrder[i]] moves to random neighbouring cell, if empty;
  }
  for (i = 0; i < number of hBots; i++){
    hBot[i] senses surroundings;
  }
  for (i = 0; i < number of hBots; i++){
    hBot[i] updates current state;
  }
  Update contents of all cells;
  Cells displayed;
  Time-steps++;
```

Figure 5.6: Pseudo code for main Simplified Hexagonal Model program supervisor detailing the order in which the hBots perform their actions.

5.6.1 Experimental Data

In a real world application for a swarm cooperating for object recognition it would be very important that is was known how well the swarm would perform in completing the task to some pre-determined criteria. For example if time was going to be more important than cost would increasing the number of agents increase or decrease the time taken to find the identify the objects? If time is decreased is the decrease in time worth the extra cost? How important would robustness be? Would it be sufficient to have a swarm that succeeds in finding only 90% of objects but does it quickly as opposed to one that succeeds all the time but takes a long time to do so? These factors would determine the desirability of this system for cooperative object recognition and so it is important from the start to produce results on the efficiency and capability of the swarm within the simulation.

To measure the efficiency and capability of the swarm to cooperatively identify different object shapes numerous pieces of information were gathered. The two main factors that represent the ability of the swarm are the amount of time they take to complete a given task scenario and how much energy they consume as a group. As the experiments were carried out in grid-world simulations time was measured by counting the number of time-steps. The values found allow the comparison of both different scenarios and different control variables for the hBots.

In the SHM the energy required to complete any task is estimated by multiplying the number of hBots in the scenario by the number of time-steps recorded. This estimation does not consider the individual actions of the hBots but provides a suitable measure to gauge where the completion of the task may be time efficient but it is not energy efficient, since a disproportionate amount of agents are used.

5.7 Limitations of the Platform

Due to the nature of the methods chosen for completing this investigation there were some limitations to the SHM as a platform and limitations in the way in which the agents were modelled. These limitations were chosen to make the investigation possible in the time allotted for the project. Future research in the cooperative object recognition area will provide answers to these current limitations the details of which are discussed further in Chapter 11. The limitations themselves are considered in more detail here.

5.7.1 Grid-World

The SHM was constructed on a hexagonal lattice where the hBots were the same dimensions as a single cell and the object shapes were made from a number of cells. The movement of the hBots was discrete from one cell to another, which is a simplification of real world movement.

The choice of a hexagonal grid allowed for a standard relationship between neighbouring cells as there is there is only one type of potential neighbour (section 4.4.2). However, this makes the grid-type less standard and therefore more difficult to compare to existing grid systems. It would be possible to convert the system to a square grid-world, although that conversion is not considered here.

5.7.2 The Requirement to Cooperate

The hBots cooperate to identify differences in object shapes. It would be possible to derive a system were a single autonomous hBot with similar capabilities could complete the same task. To do this the hBot would have to travel around the object shape changing state dependant on its current state and what it senses when it moves to the next cell. This would require additional capabilities. In order to travel round the object shape it would have to know the direction of the object cells and determine the next empty cell to move to. However, there are a number of advantages in the cooperative method which will have greater influence when the system is moved to a physical platform:

- As the scale of the object shape increases relative to the size of the agents, it will require a longer amount of time for a single agent to travel around the object shape, where cooperating agents can cover more of the object shape at the same time.

- A system which involves surrounding an object shape with multiple agents can react quicker to any changes in that object shape when compared to a single agent that needs to travel around the object shape. This increases the number of applicable application for the system.

- When the system is moved to a physical platform, the agents themselves can become a unit of measurement. If they are cooperating and neighbour with each other they form a network similar to a lattice. If however, one agent was to

travel around an object shape it would require controlled movement and absolute position knowledge to build up information of the object shape being identified.

5.7.3 Perfect Sensors and Communication

All the hBots were modelled with perfect sensor and communication capabilities. A hBot would neither falsely broadcast which state it is in nor will a hBot misinterpret another hBots broadcast. A hBot would also always correctly identify the number of object cells it is neighbouring. In a physical application it may not be possible to have agents that have 100% accurate sensors and communication, there will need to be consideration of this in future studies once the generic strategies have been tested.

5.7.4 Neighbouring Up to Three Object Cells

The system assumes that object shapes which would allow a hBot to be in contact with more than three object cells at a time do not exist. This limits the number of potential shapes that the hBots could be used to identify. The method could be expanded to consider up to six object cells. However, this would significantly increase the number of states required by the hBots whilst not affecting the ability of the hBots to identify the chosen object shapes for the experiments. Due to the way a hBot reacts to object shapes it is not capable of distinguishing between two object shapes that are separated by a single cell as they would consider them to be the same object shape.

5.7.5 Knowledge Equivalent to Five hBots

Through their cooperation the hBots can reach a state which where they are aware of the equivalent information of five individual hBots. This makes certain object shapes indistinguishable from each other, where they have the many features similar to each other. An example of two object shape data-chains that could not be distinguished by these hBots are {1,1,1,2,2,2,2,2,1,1,1,2,2,2,2,2} and {1,1,1,2,2,2,2,2,2,1,1,1,2,2,2,2,2,2}, shown in figure 5.7. The difference between these object shapes is that of scale, one of them is one object cell longer than the other. However, the length of the object shapes would require the knowledge of at least seven hBots to distinguish between them. This would be done by considering the same space that the sub-chain of the shorter length object shape (1,2,2,2,2,2,1) is derived from with either of the sub-chains (2,2,2,2,2,2,1) or (1,2,2,2,2,2,2) from the longer object shape.

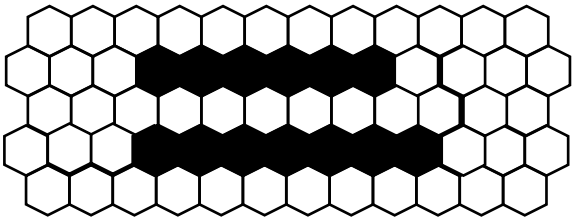


Figure 5.7: Two shapes that cannot be distinguished from each other by the hBots due to their similarity to each other and the limitations of the hBots.

5.7.6 Symmetrical Object Shapes

Due to the way the hBots change between states they cannot distinguish between symmetrical object shapes, or parts of object shapes. In the following example two sets of three hBots are neighbouring similar parts of symmetrical object shapes, figure 5.8. The hBot A neighbours an object shape with the data-chain {1,1,1,2,2,1,2,1,1,1,3,2} and the hBot B neighbours an object shape with the data-chain {1,1,1,2,1,2,2,1,1,1,2,3}. The three hBots including hBot A are on the sub-chain (1,3,2) and the three hBots including hBot B are on the sub-chain (2,3,1). Although these sub-chains are different the resulting state of hBot A and hBot B at state-level 2 will both be the result of referencing the state-rule list with values [3][1][2], state 17.

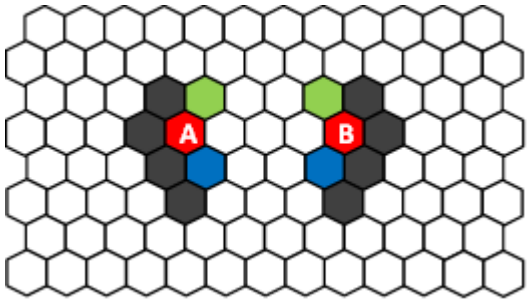


Figure 5.8: hBot A and hBot B will both change to the same state even though their neighbours are the opposite way round. Green is state 1, Blue is state 2, Red is state 3.The resulting states for both hBots A and B is state 17

5.7.7 Synchronised

The SHM is a synchronised platform, where each of the hBots perform their actions in near unison. In a physical platform this would be difficult to implement. A more idealistic simulation would have the hBots act in an asynchronised manner. The choice was made not to do so at this stage to reduce the complexity of the program and therefore the amount of run-time it would take to complete the experiments.

5.7.8 Controlled Movement

It was assumed that the hBots had an ability to control their movement in order to remain stationary with a given probability when neighbouring an object cell. Dependant on how movement would be controlled in a physical system and considering the aim of simplifying the agents' capabilities this could be difficult to implement. As no specific physical platform is currently identified for the system a range of possibilities are explored regarding the probability of movement, from 0 to 1, in section 6.2.2.

5.8 Training the Swarm

It is beneficial if the swarm that is attempting the task to distinguish between two types of object shape does not have to be explicitly told the differences between these types. To do this a method of training is required. A GA was chosen as a suitable technique for the hBots to learn to distinguish between two object shapes which would remove the process of having to identify the object shapes. The specifics of the GA method used and the random method to generate the behaviours that it is compared to are discussed in section 9.2 and section 9.3.

5.9 Summary

A cooperative object recognition task is described. In this task the swarm of agents must distinguish one of the object shapes from the other and depending on the experiment destroy the object shape or move the object shape into the collection zone.

To complete this investigation a research platform named the SHM was created in the Processing programming language. The SHM in general consisted of agents named hBots which moved around a hexagonal shaped arena made from hexagonal cells. Object shapes in the arena were made from binding numerous object cells together. The hBots identiedy the object through the changing of their states which were affected by other agents' states.

The main limitation of the SHM is that it is an abstract representation of a real world environment. As such, the strategies devised cannot be directly transferred to any specific robotic platform. However, the cooperative approach identified will influence the design of strategies for robotic platforms or the robotic platforms themselves in future. The simplicity of the method at this time gave the benefit of allowing many tests to be carried out as they are computationally light.

Chapter 6: Initial Research

This chapter contains the initial multi-agent experimentation using the Simplified Hexagonal Model (SHM) platform. In an arena with a predetermined size and number of hexagonal and triangular object shapes different sized swarms of hBots with varied probabilities of moving in certain situations were given the task of removing only the valid object shapes. The results found were used to guide further experimentation for more complex object recognition tasks where the similarities and distinguishing features of the object shapes used were varied rather than the variation of the hBot agent attributes.

6.1 Methodology

6.1.1 The Arena

For all of the initial experiments the arena contained three triangular object shapes and three hexagonal object shapes. Each of these shapes was placed at the same distance from the collection zone, which is at the centre of the arena measuring eleven cells between opposite corners. The hBots started adjacent to this collection zone. As more hBots were added they continued to spiral outward clockwise from the centre, remaining as close to the collection zone as possible, without overlapping. The arena at time-step zero is shown in figure 6.1.

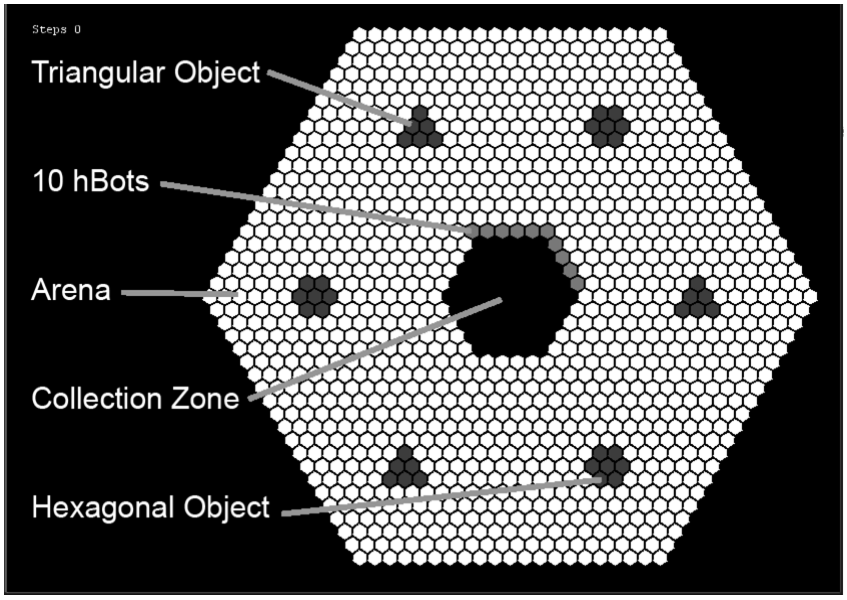


Figure 6.1: The SHM arena with both triangular and hexagonal object shapes placed evenly around the collection zone.

6.1.2 Removing Object Shapes

Object shapes were removed by the hBots when they pushed or pulled them into the collection zone. The hBots themselves could not enter the collection zone and returned to state 0 when in contact with it. When more than 50% of the object shape was in the collection zone it disappeared, as if it had fallen into a hole. To move an object shape required at least four hBots in an identifying state attempting to move it, this distinguished the movement task from the recognition task. The object shape could not be moved if there were other hBots blocking their path or the path of object shape they were moving.

6.1.3 The hBots

In the initial experiments the number of possible states and state-levels were reduced to those necessary for identifying the difference between hexagons and triangles. The defining feature of hexagonal and triangular object shapes are their corners. The method of differentiating between these two types of object shapes considered the relationship of their corners so that any scale hexagonal and triangular object shapes could be distinguished. Figure 6.2 shows different sized hexagonal and triangular object shapes where each bounding empty cell contains the number of object cells it is in contact with. The states that are achievable for any size hexagonal and triangular object shape are considered in table 6.1. From this one identifying state for both shapes is found, state 5 for the triangular object shape, and state 7 for the hexagonal object shape, the other remaining state-level 2 states are not achievable by the agents for these tasks.

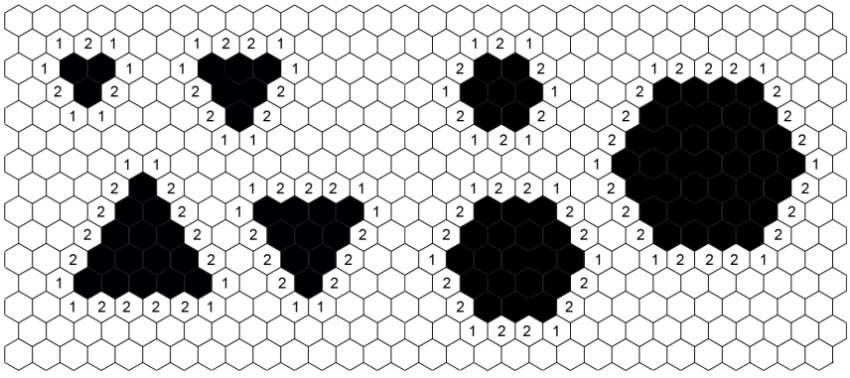


Figure 6.2: Objects with surrounding cells showing number of sides in contact with object.

State-Level	States	Possible for Every Size Hexagon	Possible for Every Size Triangle
1	1	YES	YES
	2	YES	YES
2	[1][1][1] = 4	NO	NO
	[1][1][2] = 5	NO	YES
	[1][2][2] = 7	YES	NO
	[2][1][1] = 10	NO	NO
	[2][1][2] = 11	YES	YES
	[2][2][2] = 13	YES	YES

Table 6.1: The states achievable by hBots interacting with hexagonal and triangular object shapes at state-levels 1 and 2.

One additional behaviour was used in the initial experimentations which is inconsistent with the later research and that is the behaviour of hBot in states 1 or 2, neighbouring a hBot in an identifying state will change to that identifying state. A decision tree is shown in figure 6.3 which illustrates how the hBots change states based on their surroundings for the initial experimentation. The behaviours the hBots exhibit in each of these states is described in table 6.2.

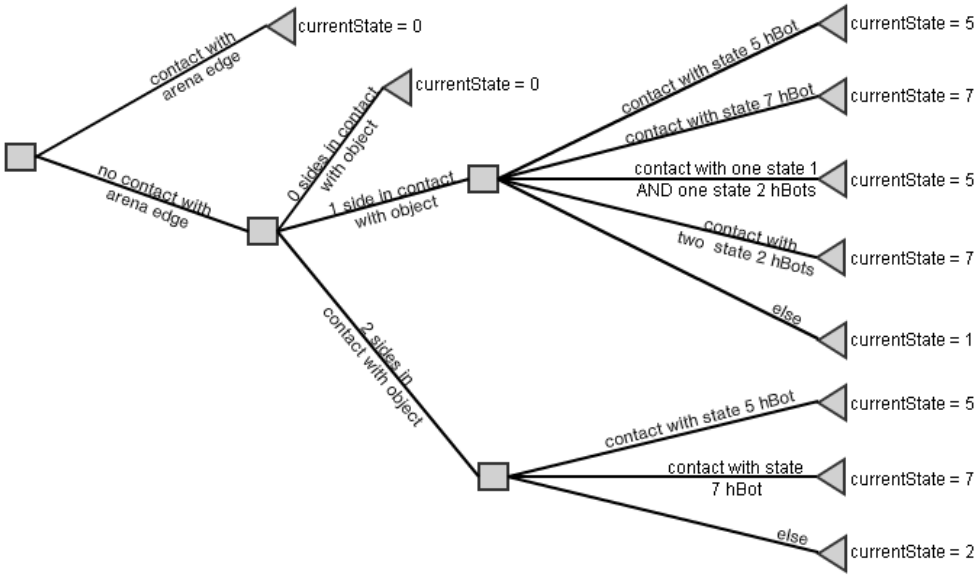


Figure. 6.3: A decision tree explaining how the current states of the hBots are determined based on their sensed surroundings.

State	Behaviour (per time-step)	Colour
0	Move in a random direction. If the direction chosen is not available the hBot remains stationary.	Grey
1	Probability between 0 and 1, determined by specific test, of moving in a random direction. If the direction chosen is not available the hBot remains stationary.	Green
2		Blue
5	Attempt to push/pull object one cell towards the collection zone if that object shape is valid. If other hBots are in the way of the hBots or object shapes they remain stationary.	Red
7		Purple

Table 6.2: The state and behaviours for the hBots.

An example of a group assessing a triangular object shape can be seen in figure 6.3 which shows:

i) Three hBots in contact with object in states 1 for single side contact and state 2 for dual object side contact, three hBots in state 0 are approaching object.

ii) A group of three hBots form at a corner of the triangle with states 1, 1, and 2.

iii) The centre hBot of the first group of three changes to state 5, from state relationship [1][1][2], a second group of three hBots is in contact with the object with states 1, 2, and 2.

iv) The two outer hBots of the first group change to state 5 also, the second group remain in states 1, 2 and 2 as this is common to both hexagonal and triangular objects.

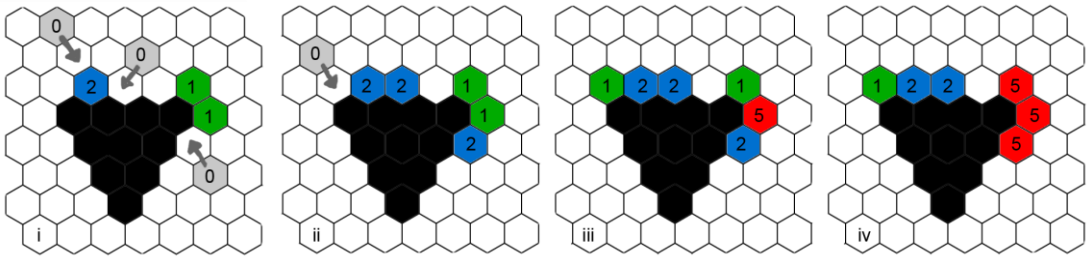


Figure 6.4: Example of hBots using state behaviour to identify a triangular object.

The assessment of a hexagonal object shape is very similar.

The details of the initial SHM program, written in the Processing open source language, used for all the initial experiments in this Chapter is included in Appendix A.

6.1.4 The Variables

A series of tests were carried out to determine the effects of three different factors in the cooperative object recognition task.

- The number of hBots
- The probability of them moving away from an object shape.
- Which object shape was considered valid.

Considering all of these three variables in relationship to each other gave a total of 440 individual tests, each of which were repeated fifty times. The hBots had a maximum of 15000 time-steps to complete the task. The data obtained included: the number of time-steps taken to complete the task; the amount of energy to complete the task and the number of object shapes removed. When all three valid object shapes where not collected the number of time-steps to complete was recorded as 15000.

6.2 Results

6.2.1 Task Completion

In tests with a low numbers of hBots at high probabilities of movement the hBots did not always manage to complete the task of removing all three invalid objects within the given 15000 time-steps. This is likely due to the hBots not remaining near the object shapes long enough to form groups of three around the corner of the object shape in order to differentiate them. As the group size increases this becomes less of a problem as overall there is less empty space in the arena and therefore more interaction around the boundary of the object shapes. The number of successfully removed object shapes for each of the experiments are shown in Appendix B.1 and Appendix B.2.

Considering all swarm group sizes, a probability of 0.1 gave the largest number of completed tasks, this is true whether the valid object shape was a hexagonal or triangular object shape as shown in figures 6.5 and 6.6 respectively. The probability which completed the least amount of tasks overall was 1.0, as shown in figures 6.7 and 6.8. At movement probabilities of 0.7 and above there is a distinct jump visible in the number of completed tasks from 90 to 100 hBots. The reason that this occurs is currently unclear. However, it could related to the amount of free space available for the

hBots to move around in, where at 100 hBots a suitable density for completing the tasks in the given 15000 time-steps is found.

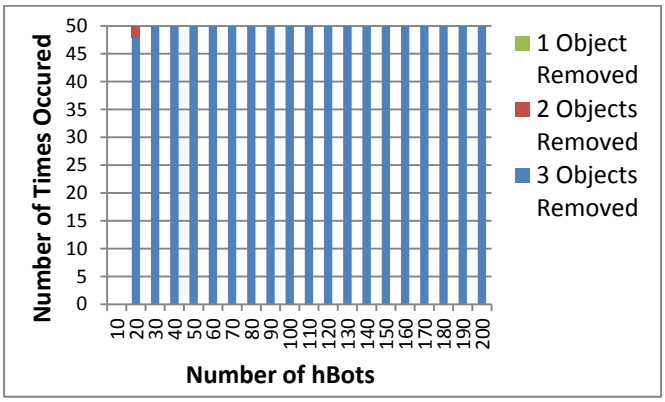


*Figure 6.5: The number of object shapes removed when the probability to move is **0.1** and the **hexagonal** object shape is valid.*

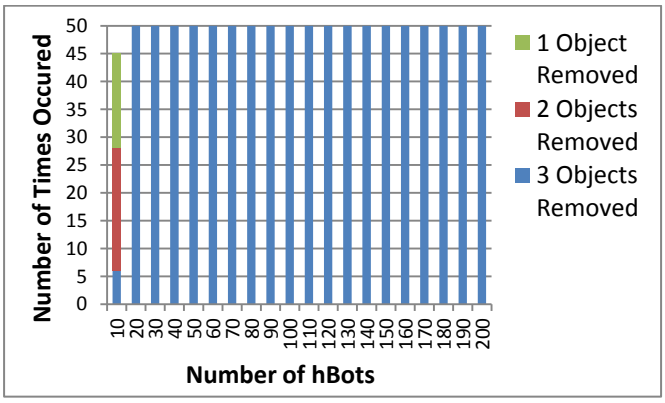


*Figure 6.6: The number of object shapes removed when the probability to move is **0.1** and the **triangular** object shape is valid.*

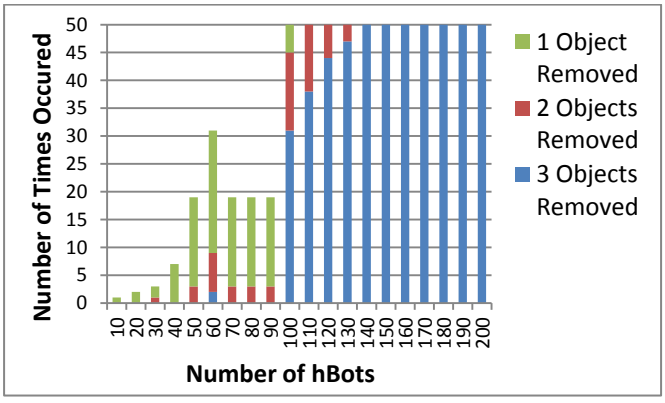


*Figure 6.7: The number of object shapes removed when the probability to move is **1.0** and the **hexagonal** object shape is valid.*

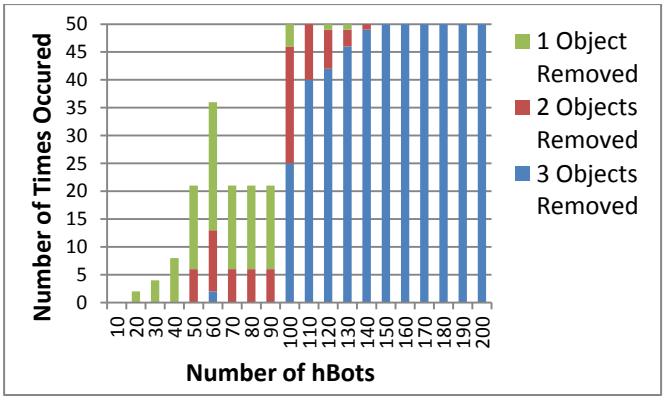


Figure 6.8: The number of object shapes removed when the probability to move is ***1.0*** *and the* ***triangular*** *object shape is valid.*

It was found that a group size of 10 was not large enough to complete the object recognition task for any probability of movement for hBots in states 1 and 2. In general, the higher the probability of the agents moving away from an object shape, the larger the swarm size was required to be in order to complete the task. Swarm sizes of 150 above were always successful at removing all three valid object shapes whether they were hexagonal or triangular. However, the information does not consider the efficiency at which the task was completed.

6.2.2 Swarm Size

As the number of hBots is increased the number of time-steps required to complete the cooperative object recognition task is decreased. This is true whether the valid object shape is a hexagon or a triangle as shown in the heatmaps in figure 6.9 and figure 6.10 respectively. The results of these tests are shown in full in Appendix B.3 and B.4. For low probabilities (0.0 – 0.3) of movement when in states 1 and 2 the number of time-steps it takes initially drops quickly between 10 and 60 hBots, and levels out when the number of hBots reaches 110. This is likely due to the higher probability that a hBot that comes into contact with an object will remain next to the object shape until the object shape is removed. Therefore there is a limited amount of variance between which of the hBots are neighbouring the object shapes. As the number of hBots increases, the number of them interacting directly with the object shapes remains constant, as there is a limited number of spaces around each shape, causing the relationship to level out. In the case of a 0.1 probability of moving there is even a minor increase in the number of time-steps it takes to complete the task when above 150 hBots, this is greater in the case of a triangular valid object figure 6.10.

However, as the probability of movement increases towards 1.0 the relationship between the two variables becomes increasingly linear when plotted on a logarithmic scale, and once a suitable amount of hBots have been reached to successfully complete the task. This change can be seen in more detail in the plots in Appendix B.3 and B.4. This is due to the hBots constantly changing position around the object shape, effectively switching places with each other. As the number of hBots increases there is a greater chance that the hBots as a swarm rather than as an individual will interact with the object shapes, this is due to the overall reduced amount of empty space in the arena in which the hBots can move.

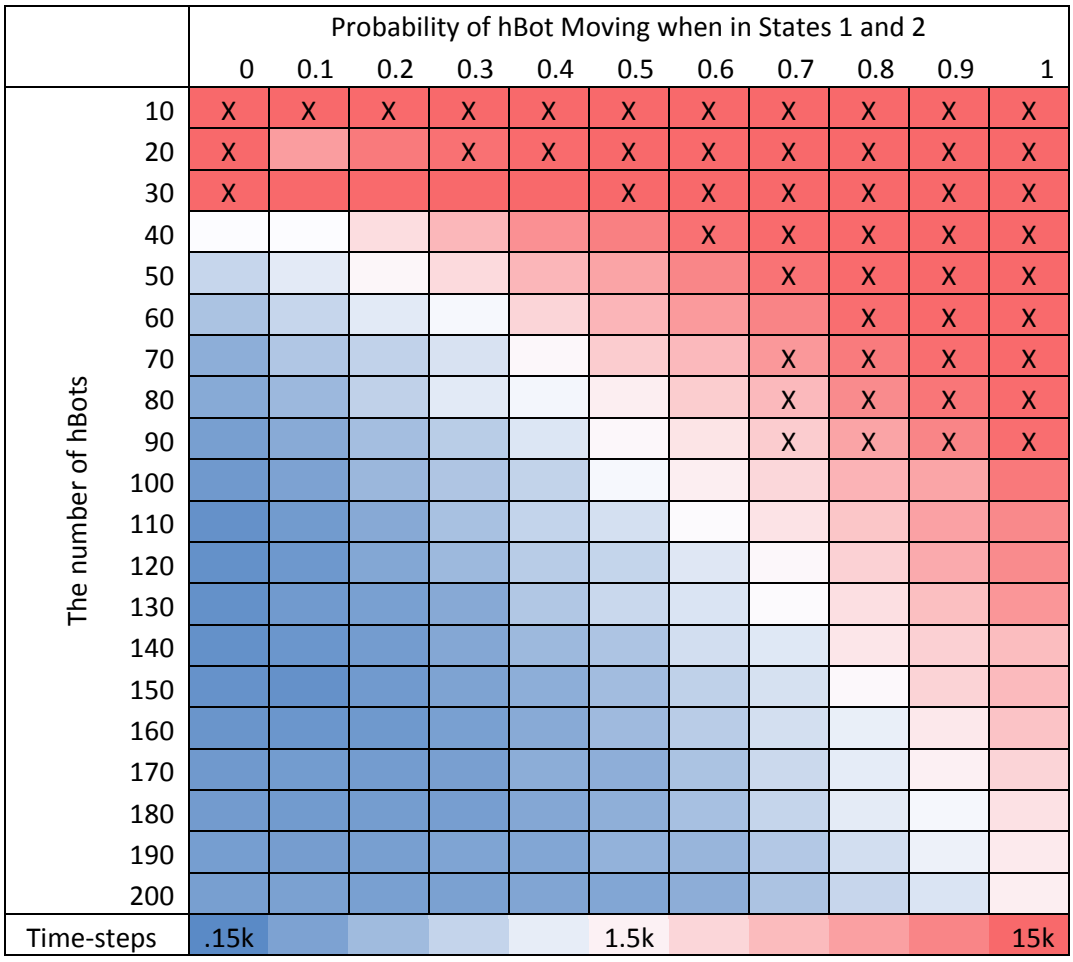


*Figure 6.9: A heatmap of the mean average number of **time-steps** required to complete the task where **hexagonal** object shapes were valid. Xs indicate results with less than 50% completed tasks, where results would have higher values.*

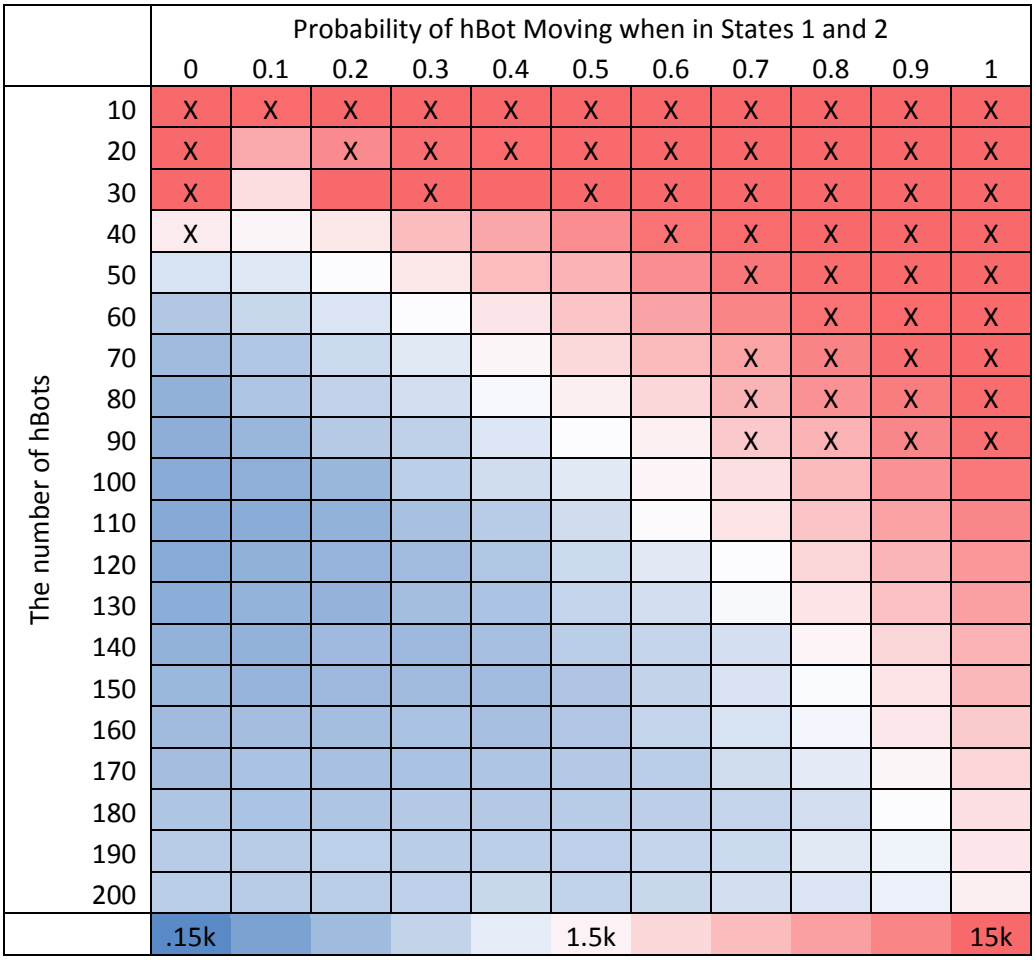


*Figure 6.10: A heatmap of the mean average number of **time-steps** required to complete the task where **triangular** object shapes were valid. Xs indicate results with less than 50% completed tasks, where results would have higher values.*

6.2.3 Energy Consumption

The time-steps it takes to complete the object recognition task gives one perspective of the swarms effectiveness of completing the task. A physical agent would require energy to work. As the number of agents in the swarm increases so does the amount of energy consumed by the swarm for each time-step. Taking this into consideration the energy each swarm group size took to complete the task was calculated for each of the tests. The results are available in full in Appendix B.5 the hexagonal object shape was valid and Appendix B.6 where the triangular object shape was valid.

Comparing the results for the time-steps, figures 6.9 and 6.10, and energy, figures 6.11 and 6.12, over both the number of hBots and their probability of movement when the triangular object shape was valid interesting changes can be noted. The clearest changes are at the extreme values where there are either low numbers or high numbers of hBots

and low and high probabilities of movement. However, at low probabilities there are a greater number of uncompleted tasks, it would be expected that in these cases it would take a larger number of time-steps to complete the task if it is even possible. When considering the energy consumption of the group these values move closer to the mean. The most efficient swarm group had 110 hBots in it and a probability 0 of moving when considering time-steps alone, but when considering the energy consumption the most efficient number of hBots was reduced to 80.

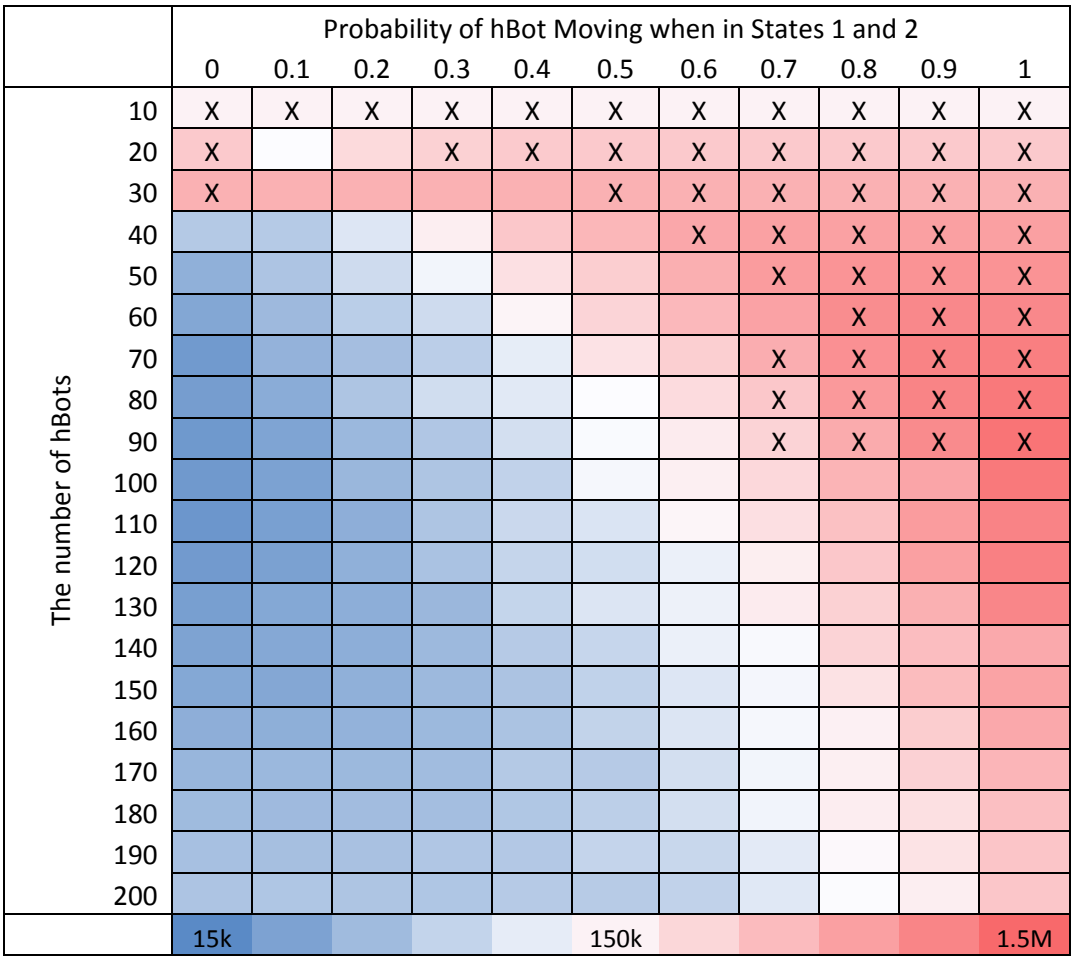


Figure 6.11: A heatmap of the mean average number of ***energy*** *required to complete the task where* ***hexagonal*** *object shapes were valid. Xs indicate results with less than 50% completed tasks, where results would have higher values.*

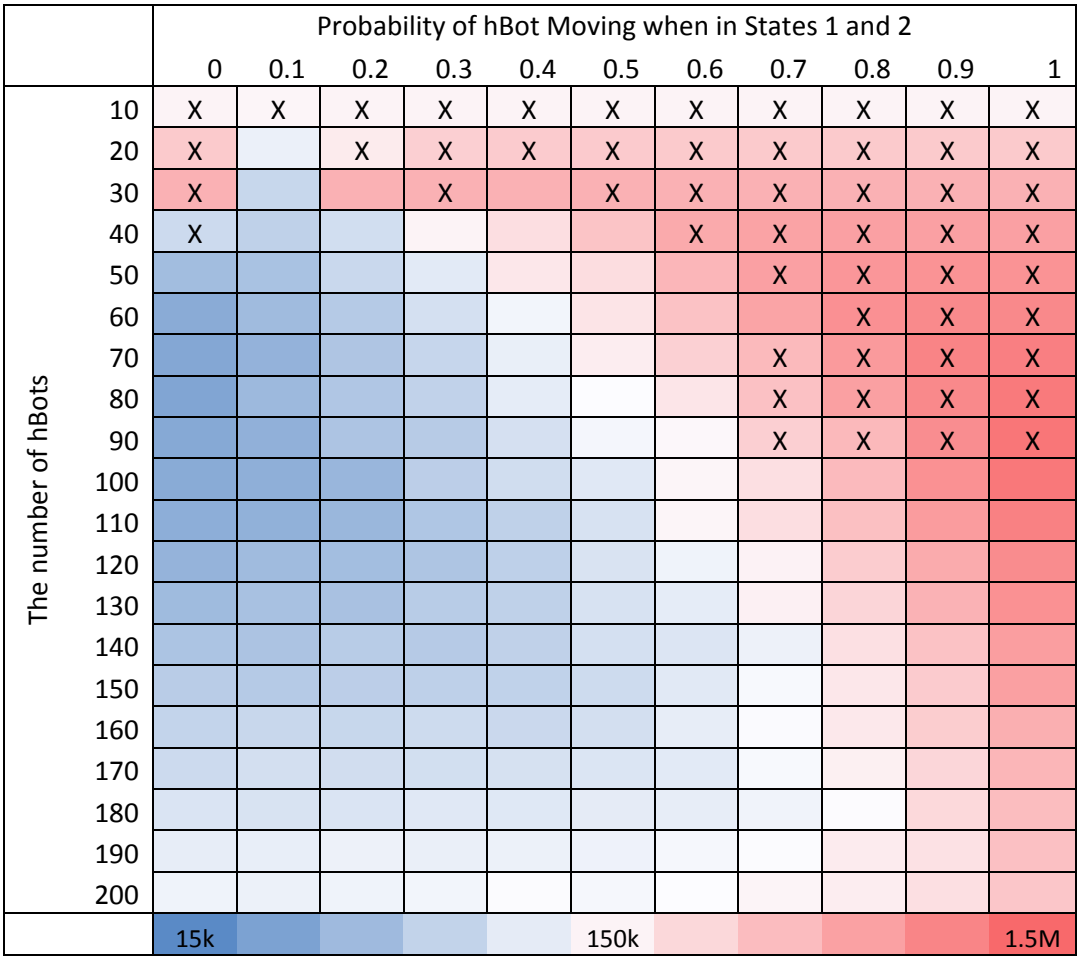


Figure 6.12: A heatmap of the mean average number of ***energy*** *required to complete the task where* ***traingular*** *object shapes were valid. Xs indicate results with less than 50% completed tasks, where results would have higher values.*

6.2.4 Probability of Movement

In general as the probability of the hBots moving away from the object shapes whilst in states 1 and 2 increases so does the amount of energy required to complete the cooperative object recognition task. This variation in energy consumption is diminished when there are either high or low numbers of hBots in the swarm, as shown in figure 6.11 and 6.12 for both when hexagonal and triangular object shapes are valid. In situations where there are higher numbers of hBots in the enclosed arena there will be an increasing number of them that do not interact with the object shape and hinder the swarms progress by either consuming energy or being in the way whilst other hBots are trying to move the valid object shapes to the collection zone. Changing these hBots probability of movement will have a relatively small effect as the arena is congested. The required number of agents quickly surround and identify the object shape and are no longer affected by the probability of movement. In the case of low hBot numbers the

hBots take a longer amount of time to identify the object shapes because they are spread too thin. It is difficult to determine, with the current results, whether if a low number of hBots remaining near the object shape or not is relevant because of the lack of data, due to hBots failing to complete the tasks. However, the trend of the heatmap indicates that the difference will be less prominent than the mid-range, 60 to 120, number of hBots.

6.2.5 Hexagons and Triangles

There was little difference between the results whether the hexagons or the triangles were the valid object shape. This is most likely due to the relationships between the object shapes chosen for these tests themselves. Both the triangular object shape, with six object cells, and hexagonal object shape, with seven object cells, have twelve spaces around them. This means that it is possible for the same number of hBots to be engaged in identifying them. There are also six places out of these twelve that would allow a hBot to each a suitable state to identify them for both situations. Despite the similarities there is a slight difference in results where higher numbers of hBots are used. When the triangular object shape was classed as valid the energy consumed increases at a higher rate for these larger numbers of hBots. This could be explained by the placement of spaces that allow a hBot to reach a suitable identifying state. When dealing with a triangle the placements are in pairs at each of the corners whilst in the case of the hexagonal object shape they are spread out one at each of the six corners. This observation is true for any size hexagon or triangle. As the hexagonal and triangular object shape are always compared to each other and not a third object shape there is little difference when either one is classed as valid and the other is classed as invalid.

6.3 Further Investigations

The initial investigation used only two types of object shapes and it was found that there was little variation between them when either was set as the valid object shape and the other set as the invalid object shape. Further experiments are required which increase the range of object shapes that are used whilst considering the difference between the object shapes themselves. How the difference between the object shapes changes the difficulty of the swarm's cooperative object recognition task is an important factor to consider.

With the focus of further investigation being on different object shape couples the other variables would need to be reduced. In the case of the probability of the hBots moving when in none identifying states, a lower value would appear to more suitable. However, as the complexity in the different object shapes is increased a number of probabilities of movement should be considered.

6.4 Summary

The initial investigation using the SHM with only two type of object shape, the triangle and hexagon, revealed a number of interesting results. Where only time-steps are considered increasing the number of hBots increases the efficiency in which the swarm completes the object recognition task. Increasing the probability that an agent in state 1 or state 2, attempting to identify the object shape, moves decreases the efficiency.

By also considering the amount of energy consumed by the swarm the most efficient set-up shifts towards a lower number of hBots with the exception of very low numbers of hBots. This suggests that with a high number of hBots there is an increasing number of hBots who are not contributing to the task completion but perhaps also hindering it by restricting the movement of the valid object shapes towards the collection zone.

Overall there was little difference between the cases where the hexagonal object shape was valid and the triangular object shape was valid. An increase in the number of object shapes with an increased range of differences between them is necessary for a more thorough study of the cooperative object recognition task.

Chapter 7: Object Shapes

In the initial hBot research, two types of object shapes were used, hexagonal and triangular. These object shapes were chosen because they had both distinct differences from each other as well as similarities. In this chapter an increased range of object shapes are considered through a systematic production of the shapes possible with a hexagonal grid-world. The data-chains that describe these object shapes without needing information of orientation and placement. With minor exceptions, that fall outside the scope of this research, data chains are shown to be unique generators for object shapes.

7.1 Possible Object Shapes

The way chosen to explore hexagonal cell object shapes was to start as simple as possible and gradually add more complexity. The simplest object shape in a hexagonal lattice is a single hexagonal cell. The next object shape, in terms of complexity, has two hexagonal cells next to each other, described as an object shape with a two cell allowance. The next object shapes are those with three cell allowances, and so on. However, once there are more than two cells in the object shape there is more than one possible combination of putting those cells together whilst ignoring rotational symmetry. In the case of object shapes with a three cell allowance there are three possible shapes. Increasing the cell allowance by only one, from three to four, more than triples the possible object shapes to ten. As object shapes are described partially by the number of object cells they contain, object shapes which appear similar but have different scales are not considered to be the same object shape. Figure 7.1 shows all the possible object shapes with one, two, three and four cell allowances as well as the identification number they were assigned for this research. Beyond this point it was increasingly difficult to find and draw the shapes by hand, as shapes were easily missed and the shear amount required would be time consuming. Despite this difficulty it was necessary to understand the nature of object shapes in the hexagonal cell grid.

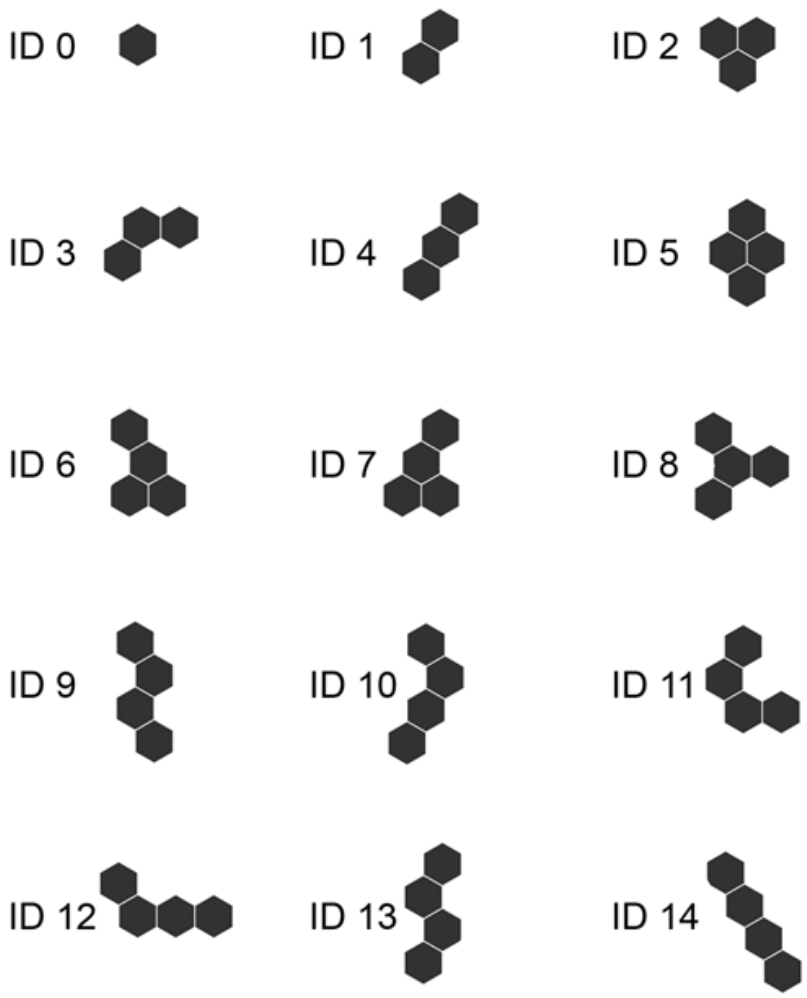


Figure 7.1: Possible object shapes built from neighbouring cells on a hexagonal grid with one, two, three and four object cell allowances. The identification number of the object shapes used by this research are shown.

7.2 Systematically Creating the Object Shapes

A systematic system was designed to search through all the possible shapes starting with a one cell allowance increasing to an *N* cell allowance. To understand how the system works spiral locations first have to be explained.

7.2.1 Spiral Location

A spiral location is a coordinate that describes a location on a grid using a single value, as opposed to a Cartesian coordinate which uses two, x and y. In figure 7.2 a cell in hexagonal grid is shown with the relative coordinates of its neighbours. Starting at the Cartesian coordinate (0,0) it is possible to spiral outwards systematically covering each cell, figure 7.3.

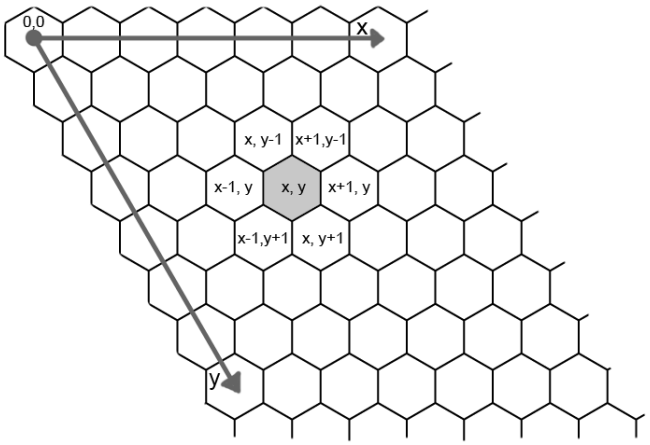


Figure 7.2: A cell and its neighbours with their relative Cartesian coordinates for a hexagonal lattice.

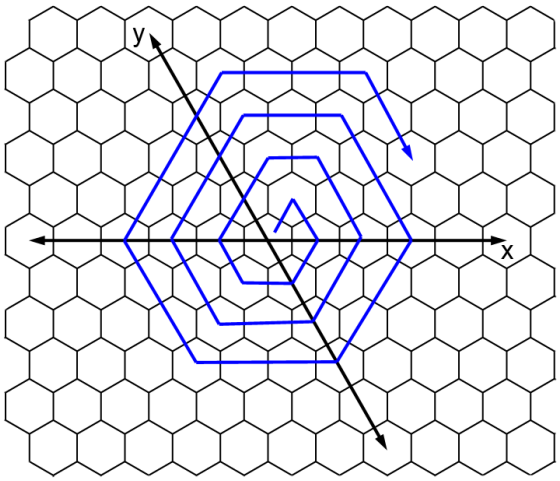


Figure 7.3: A spiralling search system shown on a hexagonal cell grid starting from (0,0) continuing outwards in a clockwise fashion.

By numbering each cell in order starting from 0 the spiral location of that cell is found. Figure 7.4 describes the spiral location coordinate for each cell and their relative ring-number is marked in red. Where, the ring number describes the minimum distance that any cell on the ring is from the centre coordinate. For example, spiral location 0 is on ring number 0, spiral locations 1-6 are on ring number 1, spiral locations 7-18 are on ring number 2.

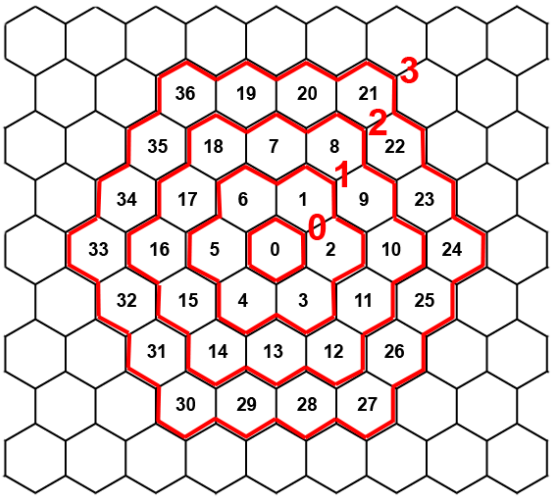


Figure 7.4: The spiral location coordinates and their relative ring-numbers. The minimum distance between the centre cell and any cell on a relative ring is equal to its ring number.

The maximum spiral location coordinate for any given ring could found be found with the following formula, where r is the ring number:

$$\boldsymbol{spiral\ location}_{\max\ for\ ring} = \mathbf{3}(\boldsymbol{r}^{\mathbf{2}}) + \mathbf{3}(\boldsymbol{r})$$

7.2.2 Producing the object shapes

To start the identification of the object shapes the first object cell is added at spiral location (0) and the shape is recorded as ID0. This becomes the first base shape for other shapes to be created from. After all n objects containing c object cells have been found these are recorded as IDp to ID($p + n - 1$) for some integer p, IDp becomes the next base shape. Then add another object cell to the base shape placing it first at spiral location (0) and then moving it incrementally through allspiral locations. At each location the resulting shape created is only classed as an authentic object shape if:

- The new object cell does not overlap an existing object cell from the base shape.
- The new object cell is in contact with at least one of the currently existing object cells.
- The object shape created is not hollow.
- The object shape created does not already exist.

Each authentic object shape is recorded including its identification number and the spiral location coordinates of its object cells. Once all possible object shapes are found

using the base shape IDp, the base shape is updated to ID(p+1) and a new object cell is added and the process repeated.

Theoretically, in an infinite grid, this method could then be repeated until all possible object shapes containing a given number of cells are found. In practice the system was limited to the size of the grid used.

7.2.3 Checking the Object Shape is Authentic

Of the four checks to determine if the object shape is authentic the first two, considering the position of the new object cell, were determined by comparing the location of all the current object cells. Hollow object shapes were identified using a flooding algorithm, where the four empty corner cells of the grid were initially marked and any empty cells touching those also became marked. These four starting cell locations were chosen to increase the spread rate, whilst not inhibiting the space the object shapes were created in at the centre. Once the flooding algorithm had reached a stable size the arena was checked for empty cells, if there were any the object shape would not be classed as authentic. Figure 7.5 shows an example of the flooding algorithm in practice with a hollow and solid object shape. The method used to remove hollow shapes is only suitable when using a finite grid space.

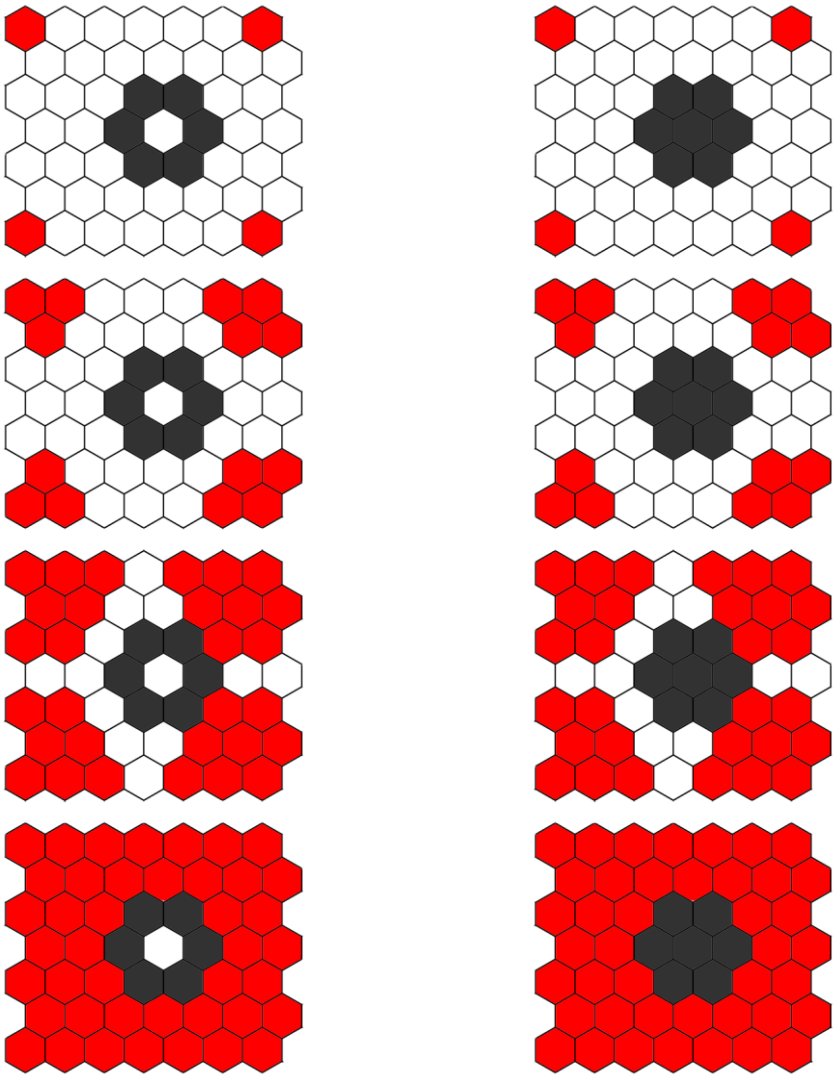


Figure 7.5: On the left: A grid containing a hollow shape is flooded until it becomes stable, there are some cells which are empty and un-marked therefore the shape is hollow and not a new object shape.

On the right: A grid containing a solid shape is flooded until it becomes stable, there are no empty cells therefore the shape is solid and potentially a new object shape.

The final check needed to determine that the new object shape was not identical to one that was already recorded with an ID number. As it is the shape of the object that is important and not the position of the object cells a method was required for coding the object shape which would remain unchanged whatever its position or orientation. The development of such a coding system was built from the concept of object shapes as binary images.

7.3 Object Shapes as Binary Images

Geometric shapes in both two-dimensional and three-dimensional space can be distinguished by their boundary regions. These differences are determined by how many sides or faces they have and how these sides and faces relate to each other. Research exists considering the relationships and similarities between different two-dimensional shapes. This research compares the shapes of objects or images to find similarities between them. Latecki and Lakämper (2002) used a process of digital curve evolution to simplify the shapes and reduce the amount of noise there is in order to compare them. The process allows for the shape similarity measure to be used with shapes in their database. In comparison Sajjanhar and Lu (1997) normalised shapes by position, scale and orientation in order for them to be compared to each other.

The object shapes in the SHM could be considered binary images where the object is the image and the arena is the background. Shape coding is used to store these types of binary images in a range of different ways (Katsaggelos et al. 1998; Zhang and Lu 2004). Most relevant of these techniques is chain coding. Chain coding maps the relative positions of the neighbouring pixels at the boundary of a shape (Katsaggelos et al., 1998). It is usually done on a square grid but has also been done on a hexagonal lattice (Scholten and Wilson, 1983). Lossless chain coding copies the exact information required to recreate the image, whilst lossy chain coding reduces the data required at the cost of making an approximation of the image. Examples of two types of lossless pixel based chain coding on a square grid using 4-neighbours and 8-neighbours are shown in figure 7.6. Here the starting position is in the top left and the code determines the position of the next cell. Once the progression returns to its starting cell the boundary of the shape is mapped, allowing the shape to be reproduced. Using 8-neighbour chain coding decreases the length of the chain-code but at the cost of increasing the amount of variables for each bit of the chain-code.

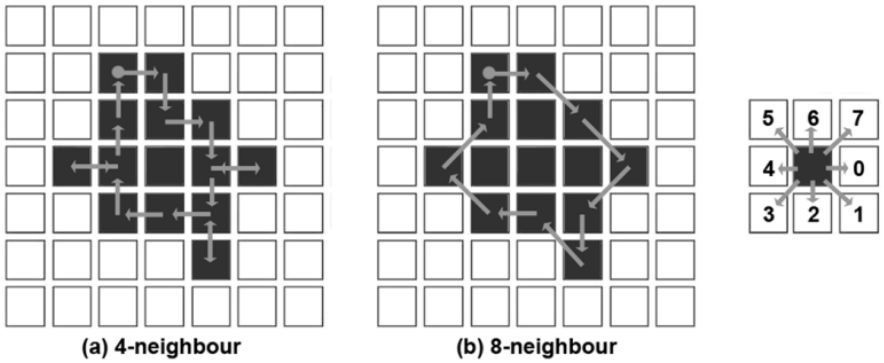


Figure 7.6: (a) 4-Neighbour and (b) 8-Neighbour contour chain coding. The start point is shown with a spot (top left). The code for (a) 0,2,0,2,0,4,2,2,6,4,4,6,4,0,6,6 and (b) 0,1,1,3,2,5,4,5,7,6.

Chain-coding is similar in theory to how a geometric shape can be represented by the length and curve of its sides and the angles they intersect. These contours can follow the pixels at the inside or the outside of the shape as seen in figure 7.7. An alternative method has been considered where the chain code considers the relationships of the inter-pixel edge links (Nunes et al., 2000; Park, Martin and Yu, 2008) as seen in figure 7.8.

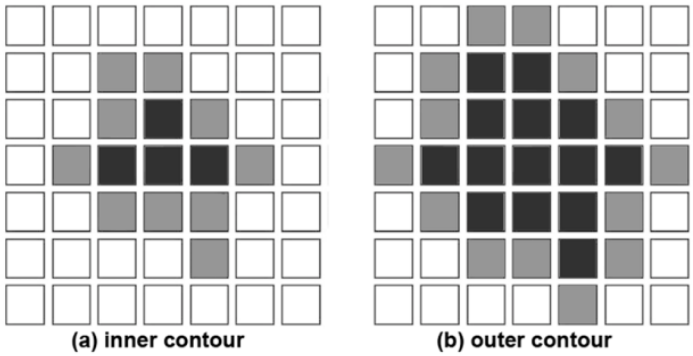


Figure 7.7: Pixels used for the contour coding are either pixels within the shape (a) inner contour or those neighbouring the shape (b) outer contour.

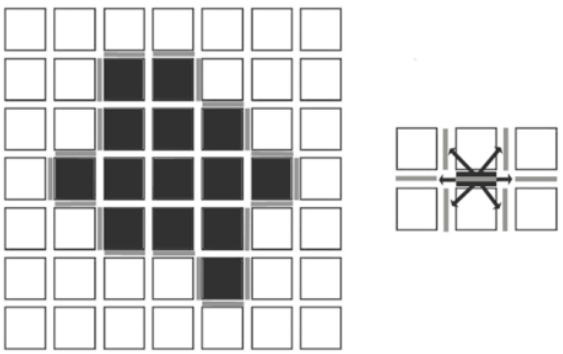


Figure 7.8: In edge based contour coding it is the relative edges of the pixels that are coded rather than the pixels themselves.

There are lossy techniques, where estimations are used to increase efficiency by decreasing the accuracy of the chain-code. These techniques aim to minimise the amount of information needed to store and transfer images, usually in videos where many images are needed for this final product (Nunes et al., 2000; Park, Martin and Yu, 2008). The issue with lossy techniques is the loss of quality when the images are required to be remade accurately.

For the case of the shapes in the SHM it was important that the representations of the objects were lossless, a near approximate would not suffice. Due to the issue of needing to know where to display the image, in the case of lossless contour and chain coding, the information needs to contain a starting point. This starting point also requires either the initial orientation of the first piece of data to be known, a common orientation or every piece of information containing an orientation. Also, an image that is rotated by 90 degrees will have a different chain-code to an image that has not been rotated. For these reasons this type of existing contour or chain coding was not suitable for the object shape investigation and a new one needed to be developed

7.3.1 Describing Object Shapes without Location or Rotation

Relative to the object shapes the hBots exist in the negative space around them, similar to outer-contour chain coding. By considering the negative space around the objects and the way the hBots perceive the object cells, it is possible to produce an array of information, termed the data-chain, which, with exceptions that fall outside the remit of this research discussed in section 7.5.3., will be shown to be unique to each object shape irrespective of placement or orientation and hence can be used in describing the object shapes used within the research.

Every empty cell that touches an object shape cell can be given a value determined by how many different object cells it touches. In figure 7.9 the first fifteen object shapes are shown with the empty cells surrounding them containing the number of object shape cells they are in contact with. This information is similar to how the hBots view the object shapes and therefore provides additional benefits when comparing the data-chains of object shapes to each other.

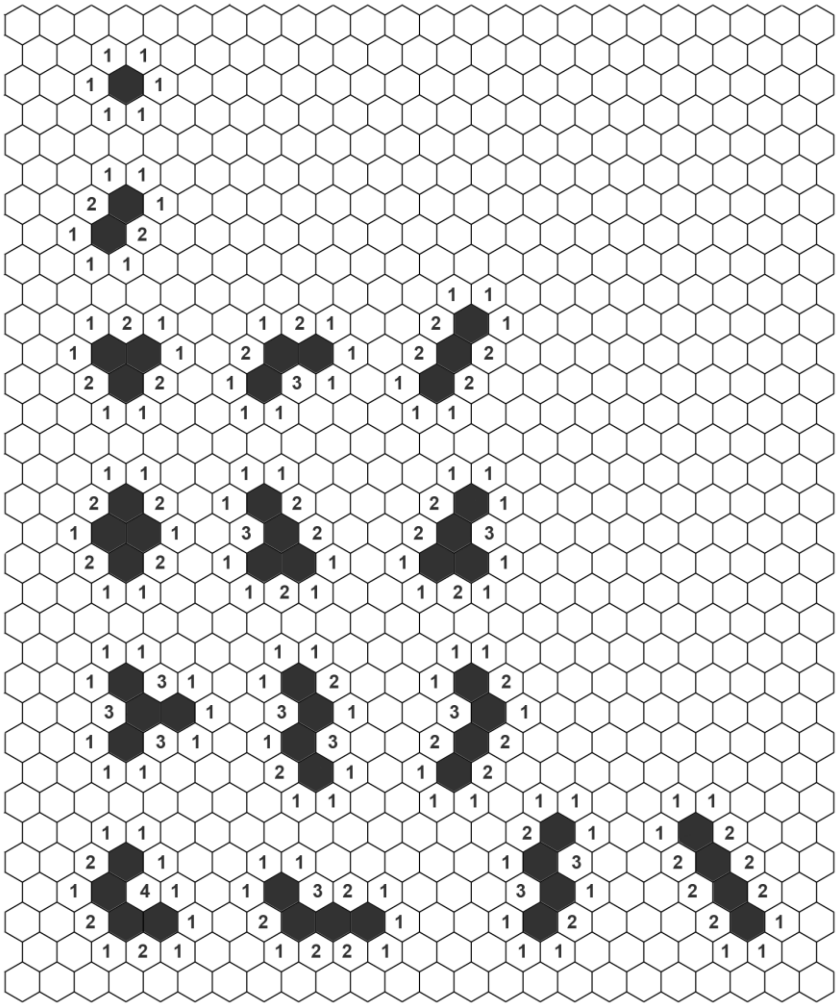


Figure 7.9: The first fifteen object shapes whose neighbouring empty cells containing the number of object shape cells they are in contact with.

This arrangement of cell values forms what is termed the data-chain for the object. Traversing the chain clockwise produces a cyclic sequence of numbers, that is a sequence $a_0, a_1, \ldots. a_{n-1}$ that can be read starting at any number a_k $0 \leq k < n$ and reading consecutively modulus n to a_{k-1} . If these reading are listed separately in lexicographical order then one will be first, this list will be used to represent the data-chain and will be referred to as the data-chain of the object. For example the data-chain from the object shape with the two cell allowance is {1,1,1,2,1,1,1,2}. In table 7.2 the data-chains for each of the first forty-eight shapes are described. The identification number describes the order in which the object shapes were found. The base shape details the identification number of the object shape that the additional cell was added to in order to make that shape in the object shape creation process.

Identification	Number of Cells	Length of Data-Chain	Data-chain	Base Shape
0	1	6	{1,1,1,1,1,1}	-
1	2	8	{1,1,2,1,1,1,2,1}	0
2	3	9	{1,1,2,1,1,2,1,1,2}	1
3	3	10	{1,1,1,2,1,2,1,1,1,3}	1
4	3	10	{1,1,1,2,2,1,1,1,2,2}	1
5	4	10	{1,1,2,1,2,1,1,2,1,2}	2
6	4	11	{1,1,1,2,2,1,1,2,1,1,3}	2
7	4	11	{1,1,1,3,1,1,2,1,1,2,2}	2
8	4	12	{1,1,1,3,1,1,1,3,1,1,1,3}	3
9	4	12	{1,1,1,2,1,3,1,1,12,1,3}	3
10	4	12	{1,1,1,2,1,2,2,1,1,1,2,3}	3
11	4	12	{1,1,1,2,1,2,1, 2,1,1,1,4,}	3
12	4	12	{1,1,1,2,2,1,2,1,1,1,3,2}	3
13	4	12	{1,1,1,3,1,2,1,1,1,3,1,2}	3
14	4	12	{1,1,1,2,2,2,1,1,1,2,2,2}	4
15	5	11	{1,1,2,1,2,1,2,1,1,2,2 }	5
16	5	12	{1,1,1,3,1,1,2,1,2,1,1,3}	5
17	5	12	{1,1,1,2,2,1,2,1,1,2,1,3}	5
18	5	12	{1,1,1,3,1,2,1,1,2,1,2,2}	5
19	5	12	{1,1,2,1,1,3,1,1,2,1,1,3}	6
20	5	13	{1,1,12,2,1,1,3,1,1,1,2,3}	6
21	5	13	{1,1,1,2,2,2,1,1,1,3,1,1,3}	6
22	5	13	{1,1,1,2,2,1,1,2,2,1,1,1,4}	6
23	5	13	{1,1,1,2,1,2,2,1,1,2,1,1,4}	6
24	5	13	{1,1,1,2,2,2,1,1,2,1,1,3,2}	6
25	5	13	{1,1,1,3,2,1,1,2,1,1,3,1,2}	6
26	5	13	{1,1,1,3,1,1,2,2,1,1,1,3,2}	7
27	5	13	{1,1,1,2,1,3,1,1,2,1,1,2,3}	7
28	5	13	{1,1,1,2,3,11,2,1,1,2,2,2}	7
29	5	13	{1,1,1,4,1,1,2,1,1,2,2,1,2}	7
30	5	14	{1,1,1,2,1,3,1,1,1,3,1,1,1,4}	8
31	5	14	{1,1,1,2,3,1,1,1,3,1,1,1,3,2}	8
32	5	14	{1,1,1,3,1,1,1,3,1,2,1,1,1,4}	8
33	5	14	{1,1,1,2,1,2,1,3,1,1,1,2,1,4}	9
34	5	14	{1,1,1,2,1,3,2,1,1,1,2,2,1,3}	9
35	5	14	{1,1,1,2,1,3,1,2,1,1,1,3,1,3}	9
36	5	14	{1,1,1,2,1,2,1,2,2,1,1,1,2,4}	10
37	5	14	{1,1,1,2,2,1,2,2,1,1,1,2,3,2}	10
38	5	14	{1,1,1,2,3,1,2,1,1,1,3,1,2,2}	10
39	5	14	{1,1,1,2,1,2,3,1,1,1,2,1,2,3}	10
40	5	14	{1,1,1,2,1,2,2,2,1,1,1,2,2,3}	10
41	5	14	{1,1,1,2,1,2,2,1,2,1,1,1,3,3}	10
42	5	13	{1,1,2,1,2,1,2,1,2,1,1,2,5}	11
43	5	14	{1,1,1,2,2,1,2,1,2,1,1,1,4,2}	11
44	5	14	{1,1,1,3,1,2,1,2,1,1,1,4,1,2}	11
45	5	14	{1,1,1,2,2,2,1,2,1,1,1,3,2,2}	12
46	5	14	{1,1,1,3,2,1,2,1,1,1,3,2,1,2}	12
47	5	14	{1,1,1,2,2,2,2,1,1,1,2,2,2,2}	14

Table 7.2: The identification number of the object shape and the corresponding, number of cells in the shape and the data-chain, length of data-chain and the base shape that the new object shape was built upon.

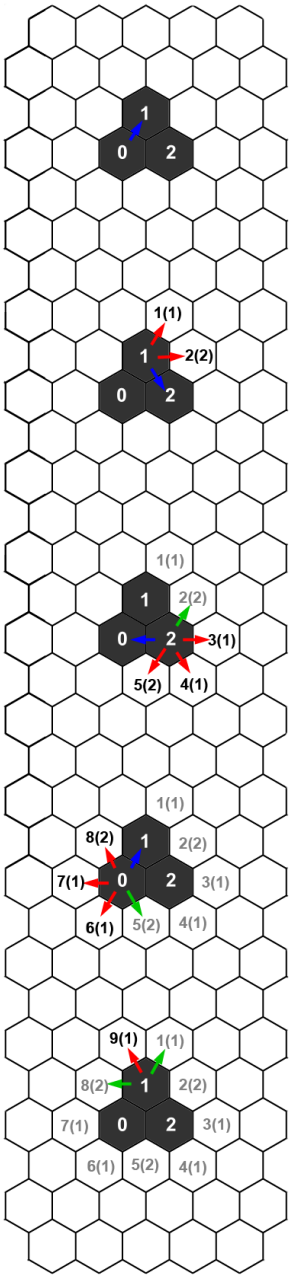
7.4 From Object Shapes to Data-Chains

A method was developed to identify the data-chain of each object shape found. As the first object shape is a single object cell at spiral location coordinate (0) the tracking pointer was placed here first. The pointer then determined if the cell at the tracking pointer's relative spiral location (1) was an empty cell or an object cell. If it was an object cell the pointer then moved to this coordinate and repeated the process until it checked and found an empty cell. This empty cell was marked as the first in the data-chain and given a value equal to the number of object cells it was in contact with. The pointer then checked its next spiral location, relative to itself, in this case (2). If this cell is also empty it was marked as the next position in the data-chain and given a value equal to the number of object cells that surround it. However, if it was an object cell the tracking pointer moved to its position. When the tracking pointer moved from one object cell location to another it needed to change which relative spiral location direction it checked first.

- Given the previous pointer search direction was the relative spiral location *P*. The current pointer search direction was *P+1 modulus 6*.

In the case that any of the empty cells had already been searched the relative search spiral location was increased by one. This process was repeated until all the surrounding cells had been marked, ordered and given a value relative to the number of object cells they were in contact with. An example of this method being used to find the data-chain for object shape ID 2 which has 3 cells is explained in figure 7.10.

Whilst it may at first appear that a data-chain is a unique representation of an object shape section 7.5.3 provides an example that this is not true for all data-chains. Section 7.5 as a whole, however, demonstrates that data-chains are a sufficient and unique representation for the object shapes used within the research and hence when such an object shape is constructed its data-chain can be compared to all the previous listed data-chains to determine if the object shape already exists.



The pointer starts at spiral location coordinate (0) and checks the cell at spiral location (1) relative to the pointer. As this is an object cell the pointer moves to this cell.

The pointer is now in spiral location (1), it checks the cell at its relative spiral location (1), it is empty. This empty cell is marked its order 1 is noted and the number of neighbouring object cells is noted in brackets, also 1. The pointer checks at relative spiral location (2), it is also empty. This is the second empty cell with 2 neighbouring object cells. At relative spiral location (3) is an object cell so the pointer moves to this position.

The pointer is now at spiral location (2). The last relative spiral location checked was 3. (3+4)%6 = 1. Checking at spiral location (1) relative to pointer, the cell is empty but has already been marked. Empty cells checked in order 3, 4 and 5 have number of neighbouring object cells 1, 1 and 2 respectively. Spiral location (5) relative to the pointer is an object cell, the pointer moves to this cell.

The last relative spiral location checked was 5. (5+4)%6 = 3. The first spiral location checked relative to the pointer, now at location (0) is 3. Order 6, 7 and 8 are found to have neighbouring objects cells 1, 1 and 2 respectively. Pointer moves to spiral location (1).

Empty cell with order 9 found to have 1 neighbouring object cell. All surrounding cells checked. Data-chain has a length 9 and is {1,1,2,1,1,2,1,1,2} found from the values from cells with orders 1 to 9 {1,2,1,1,2,1,1,2,1} which are made into a cyclic sequence from which the first in lexicographic order is determined.

Figure 7.10: Example of method to find the data-chain for object shape ID 2 with 3 cells. Blue arrows show check and move pointer to new position, red arrows show identification of unmarked empty cell and green arrows show previously identified empty cells. The spiral locations of the objects cells are marked in white text the order the empty cell is found in is marked in the empty cell and the number of object neighbours is marked inside the brackets.

7.4.1 Simple and Complex Object Shapes

When finding an object shape's data-chain each of the surrounding cells can be visited one or more times. Where every surrounding cell is visited only once the object shape is considered simple. Figure 7.11 shows two examples of simple object shapes with data-chains {1,1,1,2,1,2,1,2,1,1,1,4} and {1,1,1,3,1,2,1,1,1,3,1,2}.

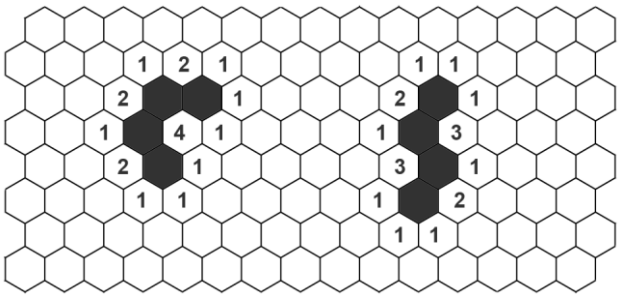


Figure 7.11 Two simple object shapes with data-chains {1,1,1,2,1,2,1,2,1,1,1,4} and {1,1,1,3,1,2,1,1,1,3,1,2}.

Where this is not true, and any number of these surrounding cells are visited more than once, this is considered a complex object shape. In the case of the complex object shapes, these cells that are used more than once to form the data-chain can be considered 'shared-links'. In the majority of cases these shared-links are used twice, but can be used up to three times in the same data-chain. These shared-links are the equivalent of a number of overlapping cells each with their own number of neighbouring cells which are added to each other to give the final value. An example of a complex object shape with a shared-link used twice is {1,1,2,1,2,1,2,1,2,1,1,**2**,5,**2**} where the bold numbers represent the shared-link. In comparison, an object shape with a shared-link used three times {1,2,1,2,1,2,1,2,1,2,1,2,1,2,1,**2**,**3**,5,**3**,5,**3**,**2**} where the bold numbers represent the twice shared-link and the underlined numbers represent the thrice shared-link. Both of these object shapes are shown in figure 7.12.

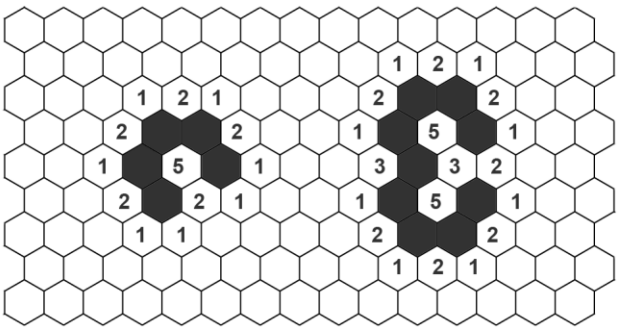


*Figure 7.12: Example of complex shapes with shared-links. Data-chains {1,1,2,1,2,1,2,1,2,1,1,**2**,5,**2**} and {1,2,1,2,1,2,1,2,1,2,1,2,1,2,1,**2**,**3**,5,**3**,5,**3**,**2**} where the bold numbers represent the twice shared-link and the underlined numbers represent the thrice shared-link.*

Complex object shapes represent the real world situation where the shapes being explored by a swarm contain cavities, tunnels or entrances of the same order of size as individual agents within the swarm. In this situation there would be increased probability of agents getting in the way of each other or blocking parts of the shape, so that the swarm would be acting at the cusp of its capability. Should a greater level of detail be necessary in the object recognition task then then either a smaller design or increased observational ability would need to be considered for the agents.

For this research cavities, tunnels and entrances were designed to be of sufficient size to be beyond the cusp of capability of the swarm. The actual issue that can occur at the cusp of capability are examined in sections 7.5.2 and 7.5.3

7.5 From Data-Chains to Object Shapes

It is necessary to show when it is possible to use the comparison of data-chains in order to identify previously found object-shapes. In other words when is a data-chain a unique representation of an object shape and the relationship between the two equivalent? A data-chain can be formed into a trace of the object shapes boundary, described by the number of object cells it is in contact with. Using this information within a data-chain it seems likely that the object shape giving rise to a data-shape could be reconstructed from that data-chain.

7.5.1 Tracing a Data-Chain

Having obtained a data-chain from an object shape a general method was devised for constructing an object shape from it. Firstly consider tracing the boundary of an object shape where the trace may be considered as a series of steps. Each step involves a move

from one vertex of a hexagon along one edge to an adjacent vertex, figure 7.13. There are six directions that the trace can follow, D_0, D_1, D_2, D_3, D_4, and D_5 shown in figure 7.14. These can be considered pairs of opposing directions, D_0 and D_3, D_1 and D_4, D_2 and D_5. The traces along these directions D_0, D_1, D_2, D_3, D_4, and D_5 can be represented by the displacement vectors (1,0,0), (0,1,0), (0,0,1), (-1,0,0), (0,-1,0) and (0,0,-1) respectively. Starting with a displacement of (0,0,0) the on-going displacement can be measured during the trace of the boundary. A finite sequence of directions will produce the boundary of an object shape if the final displacement is (0,0,0) and no two sub-displacement totals are identical. The latter case indicating the trace has crossed over itself.

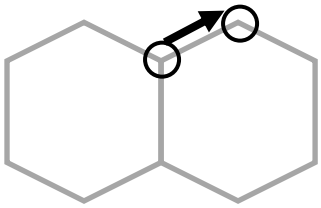


Figure 7.13: One step of tracing a data-chain, moving along an edge from one vertex to an adjacent vertex.

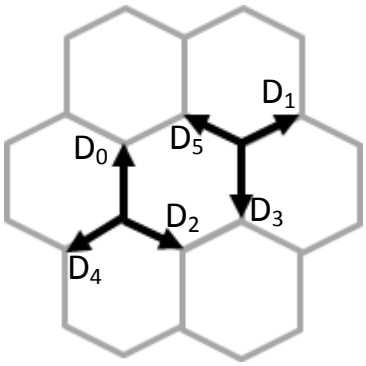


Figure 7.14: The six possible trace directions on a hexagonal lattice.

For a data-chain arising from a simple object shape the links in a data-chain can be used in order to determine a sequence of directions from which the construction of an object shape can be attempted. This is achieved using a combination of two functions:

- $F_{positive}$: Trace in current direction Dc, then c = (c+1) modulus 6.
- $F_{negative}$: Trace in current direction Dc, then c = (c-1) modulus 6.

as follows

- For a link of 1 in the data-chain: $F_{positive}$
- For a link of 2 in the data-chain: $F_{negative}$, $F_{positive}$
- For a link of 3 in the data-chain: $F_{negative}$, $F_{negative}$, $F_{positive}$
- For a link of 4 in the data-chain: $F_{negative}$, $F_{negative}$, $F_{negative}$, $F_{positive}$
- For a link of 5 in the data-chain: $F_{negative}$, $F_{negative}$, $F_{negative}$, $F_{negative}$, $F_{positive}$

For example the simple object shape data-chain {1,1,1,2,1,2,1,1,1,3} is tested following these instructions in table 7.3. Each function is carried out in order determined by the data-chain and the values of displacements are calculated. At each stage the displacement is checked against previous displacements. As no displacement other than the starting displacement is repeated the data-chain has constructed an object shape.

Value of link in proto-chain	Function	Displacements	Same as Previous Opposing Value	Current Direction
-	-	[0,0,0]	FALSE	D_0
1	$F_{positive}$	[1,0,0]	FALSE	D_1
1	$F_{positive}$	[1,1,0]	FALSE	D_2
1	$F_{positive}$	[1,1,1]	FALSE	D_3
2	$F_{negative}$	[0,1,1]	FALSE	D_2
	$F_{positive}$	[0,1,2]	FALSE	D_3
1	$F_{positive}$	[-1,1,2]	FALSE	D_4
2	$F_{negative}$	[-1,0,2]	FALSE	D_3
	$F_{positive}$	[-2,0,2]	FALSE	D_4
1	$F_{positive}$	[-2,-1,2]	FALSE	D_5
1	$F_{positive}$	[-2,-1,1]	FALSE	D_0
1	$F_{positive}$	[-1,-1,1]	FALSE	D_1
3	$F_{negative}$	[-1,0,1]	FALSE	D_0
	$F_{negative}$	[0,0,1]	FALSE	D_6
	$F_{positive}$	[0,0,0]	TRUE	D_0

Table 7.3: Tracing the data-chain {1,1,1,2,1,2,1,1,1,3} to determine that it is a data-chain representing an object shape.

For simple object shapes the method of construction from the data-chain ensures that the object shape constructed has the same data-chain as that used during its construction. Hence for simple object shapes the object shape and its representation as a cyclic sequence are isomorphic. So it is legitimate to use the representation when checking to see if simple object shapes have already been identified during their systematic production.

7.5.2 Tracing Complex Object Shapes

For complex object shapes a refinement of the data-chain is required since each 2, 3 or 4 in the data-chain could be either a shared-link or not. Figure 7.15 shows the different positions of object cells arising from the same value in a data-chain.

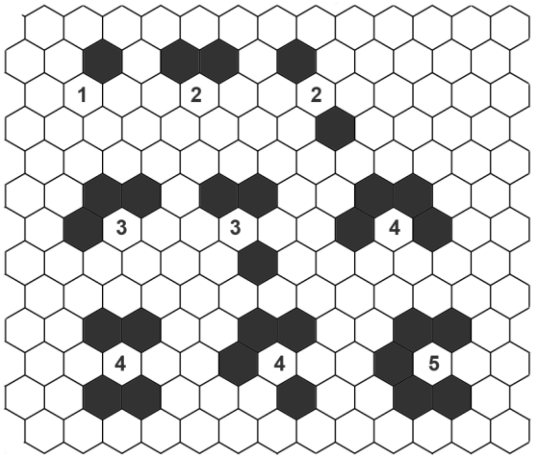


Figure 7.15: All relative object cell placement options, not considering rotation, that produce 1, 2, 3, 4 or 5 in a data-chain.

This means that for each link in the data-chain there multiple traces to be considered for each of these circumstances. The number of possible traces is affected by the value of the link in the data-chain, as follows:

- A link of 1 in the data-chain gives a single trace option.
- A link of 2 in the data-chain could be 2 or 1, giving two trace options.
- A link of 3 in the data-chain could be 3, 2 or 1, giving three trace options.
- A link of 4 in the data-chain could be 4, 3, 2 or 1, giving four trace options.
- A number 5 in the data-chain gives a single trace option.

It is possible to determine the total number of options that need to be tested when attempting to construct an object shape from a data-chain using the following rules:

- Each 2 in the data-chain, doubles the number of potential traces.
- Each 3 in the data-chain, triples the number of potential traces.
- Each 4 in the data-chain, quadruples the number of potential traces.

Each searched option produces a representative data-chain, or rep-chain, which refines the data-chain by splitting the shared-links into the appropriate values for the component traces. For example the data-chain {1,1,2,1,2,1,2,1,2,1,1,**2**,5,**2**} would give a total of sixty-four rep-chains each represented by a different route along the branches as shown in figure 7.16.

For each of the rep-chains found an attempt to construct an object shape can be undertaken using the functions in section 7.5.1. Each attempt can be tested in the same way as before to check for crossing points and for closure. False rep-chains, if traced out, will cross at least a single point more than once or not return to the starting position. In the case of the data-chain {1,1,2,1,2,1,2,1,2,1,1,**2**,5,**2**} the rep-chain that finds the solution is {1,1,2,1,2,1,2,1,2,1,1,**1**,5,**1**}. Notice how it is the two shared-links that have changed value from the data-chain to the rep-chain. Unlike the data-chains from simple object shapes which are traced once, the necessary branching of data-chains from complex object shapes may produce more than one rep-chain that construct object shapes. Since these rep-chains are different the systematic algorithm of section 7.4 will identify the associated object shapes as different though both will have the same data chain.

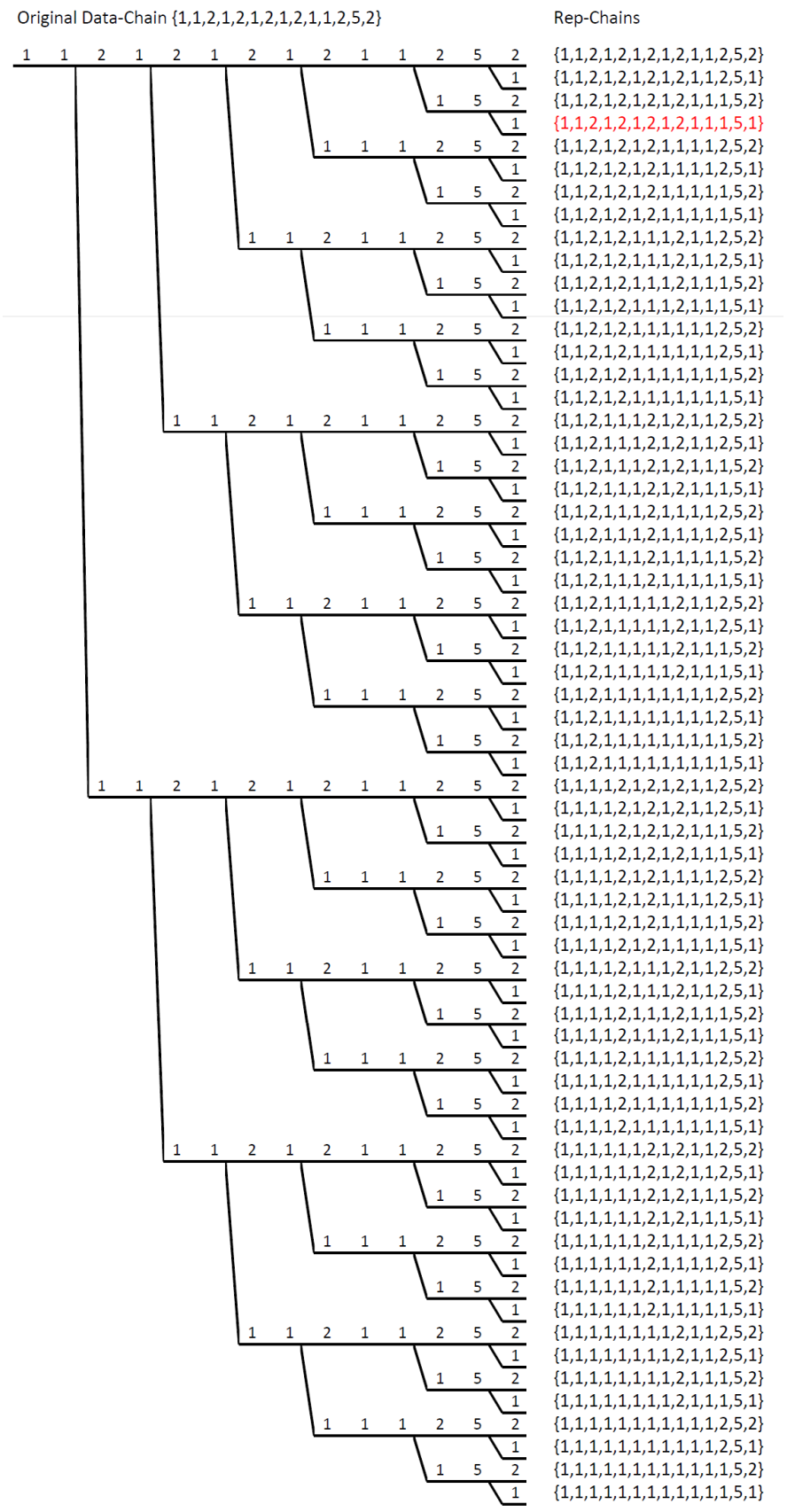


Figure 7.16: Search for rep-chains of the data-chain {1,1,2,1,2,1,2,1,2,1,1,2,5,2} which is shown to be a true object shape by the rep-chain {1,1,2,1,2,1,2,1,2,1,1,1,5,1}, highlighted red in the diagram.

7.5.3 Single Data-Chains with Multiple Object Shapes

There are instances of data-chains describing more than one object shape. This is due to the way the data-chains are produced. The first example of a data-chain that produces two different object shapes, has eight object cells and the data-chain {1,1,2,1,2,1,2,2,1,2,1,2,1,1,**3**,5,**3**}, where the shared-link is highlighted bold. The two object shapes it represents are shown in figure 7.17. The reason the object shapes have the same data-chain is due to the narrow tunnel. In one object shape this tunnel goes one way and in the other it goes the other way. This subtle change is not noticeable with the current data-chain notation system.

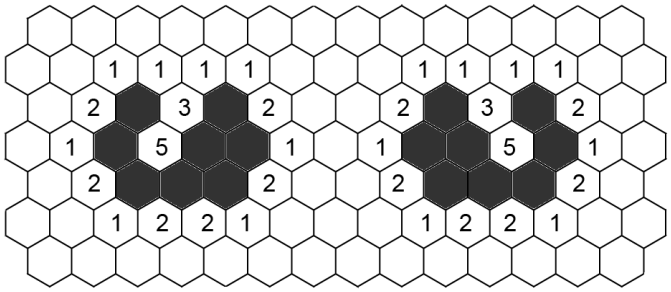


Figure 7.17: Two different object shapes with the same data-chain {1,1,2,1,2,1,2,2,1,2,1,2,1,1,3,5,3}.

Where the rep-chains are found for these types of data-chain, they produce numerous viable results. In this example the rep-chains {1,1,2,1,2,1,2,2,1,2,1,2,1,1,**1**,5,**2**} and {1,1,2,1,2,1,2,2,1,2,1,2,1,1,**2**,5,**1**} are found providing a clear distinction between the two object shapes, where the shared-link is split into its component parts. Although these object shapes are symmetrical it is not necessary for this to be true for shared data-chains to occur. This can be proven by adding another object cell at the furthest left extremity of both shapes which in both cases gives the data-chain {1,1,1,3,1,1,**3**,5,**3,** 1,1,2,1,2,1,2,2,1,3}.

The requirement for rep-chains occurs at the cusp of capability of the swarm due the order of size of the hBots being close to that of the size of the narrow tunnelling within some complex object shapes. The reason for the rep-chains not being used as the standard representation for object shapes is that within the scope of this research a hBot counts the number of object cells it neighbours without considering their position. This is a limitation of the hBot and data-chain systems where neither can distinguish between object shapes that have a similar shape other than the direction of a narrow

tunnel. To add further observational requirements to a hBot would remove its simplicity without adding a huge amount of benefit. The capabilities of a swarm in completing the cooperative recognition task can at these initial stages be adequately determined by choosing simple object shapes within the experiments.

7.6 Types of Object Shapes Found

The object shape search method, details in Appendix C, was run up to and including object shapes with a cell allowance of eight. The number of possible object shapes found is described in table 7.4. As the number of object cells used increases the number of different object shapes for each cell allowance increases. The first complex object shape occurs when there are five object cells and is shown in figure 7.12, the first pair of object shapes with the same data-chain occurs when there are eight object cells, shown in figure 7.17.

Number of Object Cells in Object Shape	Number of Object Shapes	Number of Complex Object Shapes	Number of Data-Chains with Multiple Object Shapes
1	1	0	0
2	1	0	0
3	3	0	0
4	10	0	0
5	33	1	0
6	146	11	0
7	618	85	0
8	2802	561	1

Table 7.4: Increasing the cell allowance for the object shapes increases the amount of possible object shapes that can be made.

7.6.1 Computational Complexity

The limiting factor for the complexity of identifying object shapes is checking that the new object shape has not already been found. As this is done by comparing the data-chain of the object shape to all the previously identified data-chains, for the number of object cells allowed this is an exponential time complexity, ($O(e^n)$).

The number of locations the new current cell has to be tested in increases in relation to the ring size, figure 7.4, required for containing the object shape. This is true for testing if the latest spiralling search object cell is overlapping a current base-shape cell and testing if it is neighbouring one of the current base-shape cells. Due to the way the object shapes are found the ring number is at most one greater than the current number

of object cells. The ring numbers have a quadratic relationship with regards to the number of cells contained within them, section 7.2.1. Therefore these methods both have a quadratic time complexity, ($O(n^2)$).

The flooding algorithm, section 7.2.3, decreases in the time it takes to stabilise as the number of object cells in the object shape increases, as there are less cells to complete. In its current configuration the arena size is required to be at least one ring size larger than the maximum number of object cells to allow placement of the initial flooding cells. Therefore it also has a quadratic time complexity, ($O(n^2)$).

Martins and Simoni (2009) discuss an alternate method for determining the different possible combinations of hexagonal cells whilst considering metamorphic robots. Their system involves identifying the number of distinct orbits a shape has in order to determine the number of places a new cell can be added. Although the system they use may be more efficient in identifying different configurations of hexagonal cells, the method used here involves data-chains which not only describe the object shape but can be used to determine a difference value between two object shapes, as completed in Chapter 8.

7.7 Summary

By increasing the number of object cells in an object shape, an ever increasing number of object shapes can be found. A systematic method is described for finding these object shapes starting with a single cell object shape ID0, and using this as a bases for finding new object shapes, which in turn are used to find further object shapes with increasing numbers of object cells. To be considered a new object shape, the object shape must not be hollow nor already exist.

To determine if the object shapes created have already been noted a data-chain is created for each object shape. The data-chain describes the boundary region of the object shape in a similar manner to chain-coding, however it is done in such a way that it considers how a hBot would view the object shape. It is also possible to determine if any given data-chain can reconstruct its object shape by tracing directions mapped from the links within a chain. For some object shapes there was a requirement to consider the variety of object cell placements for data-chain links with the values 2, 3, or 4. This produces rep-chains only some of which construct object-shapes checked by determining if their own traces return to the starting position without crossing any other

vertex on that trace more than once. It was found that in some instances data-chains describe more than one object shape. This is a limitation of the method used to determine the data-chain of the object shape which itself is modelled on how the hBots view and interact with the object shape and therefore share this same limitation.

Chapter 8: Comparing Object Shapes

The technique of comparing complete data-chains to distinguish between object shapes does not by itself provide a metric of how different two object shapes are from each other when compared to another pair of object shapes. To do so requires looking at sections of the data-chains, named sub-chains to determine numerical variations between two object shapes. The values of these variations are dependent on the length of sub-chain that is considered. A general method indicating how much information is needed about an object shape to distinguish it from the other is described first. Here sub-chain sequences of fixed length for two object shapes are compared term by term to see if they match and the results used to calculate a distinguishing value. A metric, the difference value, more specific to the cooperative recognition task is then produced based on the way that hBots interact with the object shapes. In this case the method converts the terms of a sub-chain into the states a hBot could achieve and the difference values are calculated from the state values. A comparison between the calculated difference values is made with experimental results which measured the number of time-steps it takes a swarm of hBots to complete the object recognition task with the same object shapes.

8.1 General Method for Comparing the Object Shapes

To determine a value of difficulty for distinguishing the differences in pairs of object shapes when only given a limited knowledge of that object shape a series of similarity tests were carried out on a number of object shapes. The limited knowledge about the object shape is represented by partial information about the entire data-chain. This partial information, or sub-chains, are taken from consecutive linear sequences of links from the data-chain. The length of a data-chain is the number of cell bordering an object shape which is the same as the number of links in the representation of the data-chain. In theory these sub-chains can vary in length from one to any size number, as a data-chain is a cyclic sequence, for example (2,1,1,2,1,1,2,1,1,2,1,1) is a sub-chain of length 12 from the data-chain {1,1,2,1,1,2,1,1,2} of length 9. Sub-chains with length greater than the length of the data-chain will start repeating from term data-chain length plus one. This repetition of information, to some extent, also reflects how hBots, due to their limited capability, interact with object shapes through their state changes. A specific sub-chain describes a feature of an object shape at a given size which can be

compared to features of another object shape to determine if they are identical or not. Examining sub-chains of different lengths gives an impression of how capable different size groups of neighbouring agents would be at discerning different object shapes. In this general case each term of a sub-chain is considered in the same order as the data-chain. A number, the distinguishing value, can be calculated between two object shapes at a given sub-chain length as follows.

All sub-chains of the same fixed length for both object shapes are found and the non-matching ones counted.. For example all the possible sub-chains of length three for the data-chains of objects shapes ID 2 and ID 1 are given in table 8.1

ID 2: {1,1,2,1,1,2,1,1,2}	**ID 1: {1,1,1,2,1,1,1,2}**
(1,1,2)	(1,1,1)
(1,2,1)	(1,1,2)
(2,1,1)	(1,2,1)
(1,1,2)	(2,1,1)
(1,2,1)	(1,1,1)
(2,1,1)	(1,1,2)
(1,1,2)	(1,2,1)
(1,2,1)	(2,1,1)
(2,1,1)	

Table 8.1: The different sub-chains with three data-links possible for object shapes ID 2 and ID 1's data-chains. Sub-chains of three that do not occur in the other shape are highlighted red.

The number of possible sub-chains of a given length is equal to length of the data-chain. The same patterns of sub-chains can occur more than once within a single object shape's data-chain. These repetitions represent all the different parts of the object shapes, which are similar to each other. In the above example all the sub-chains length three in ID 2 are also in ID 1. However, in the reverse case, object shape ID 1 has a sub-chain length three (1,1,1) that object shape ID 2 does not have. Therefore object shape ID 2 is not distinguishable from ID 1 but ID 1 is distinguishable from ID 2 with sub-chain length three. In general the distinguishing value when comparing object shape A to object shape B is different to that found when comparing B to A. The distinguishing value in comparing A to B is the number of sub-chains of given length that are found in A but not in B divided by the length of the data-chain for A. Distinguishing values will always be from zero to one. A measure of how object shape P differs from object shapes Q is given by dividing the number of sub-chains of given length of P not found in Q by the min-length of P.

In the above example the distinguishing value for ID 2 compared to ID 1 is 0 and for ID 1 compared to ID 2 is 0.25. These solutions were found as there are no sub-chains in ID 2 that are not in ID 1 and two cases of (1,1,1) in ID 1, which are different from any sub-chains in ID 2, and ID 1 has a data-chain length of 8, 2 divided by 8 is 0.25. This demonstrates that inspecting how different object shape ID 1 is to object shape ID 2 is not the same to inspecting how different object shape ID 2 is to object shape ID 1. This is true of all the different combination pairs of object shapes.

Distinguishing values can be found for all sub-chain lengths. Increasing the sub-chain length to four, as shown in table 8.2, a different set of distinguishing values are found. The sub-chains which are not in the comparison ID's set of sub-chain, at that particular length, are highlighted red. In this case the distinguishing value for ID 2 compared to ID 1 is 0.33' and for ID 1 compared to ID 2 is 0.5.

ID 2: {1,1,2,1,1,2,1,1,2}	**ID 1: {1,1,1,2,1,1,1,2}**
(1,1,2,1)	(1,1,1,2)
(1,2,1,1)	(1,1,2,1)
(2,1,1,2)	(1,2,1,1)
(1,1,2,1)	(2,1,1,1)
(1,2,1,1)	(1,1,1,2)
(2,1,1,2)	(1,1,2,1)
(1,1,2,1)	(1,2,1,1)
(1,2,1,1)	(2,1,1,1)
(2,1,1,2)	

Table 8.2:The different sub-chains with four data-links possible for object shapes ID 2 and ID 1's data-chains. Sub-chains of length four that do not appear in the other shape are highlighted red.

8.2 Results from General Method of Object Shape Comparison

The distinguishing values for all pairs of all object shapes with four object cells or less, ID 0 to ID 15. was recorded for different length sub-chains of the data-chains, ranging from 1 to 10. For each sub-chain length the average distinguishing value was calculated by adding all the distinguishing values of each ordered pair of object shapes found for that sub-chain length, excluding those that compared against themselves, and dividing it by the total number of comparisons made. As the sub-chain size increased so did the average distinguishing value between all the object shapes when compared to each other, as shown in figure 8.1. This outcome was expected as the comparison of longer sub-chains within the data-chains would lead to a lower likelihood of similar patterns arising, increasing the distinguishing values to their maximum value of one.. The graph

in figure 8.1 shows that for the first fifteen object shapes, ID 0 – ID 14, once a sub-chain length of 8 is considered all the object shape pairs can be distinguished from each other considering any position in the data-chain, where the longest data-chain has length twelve.

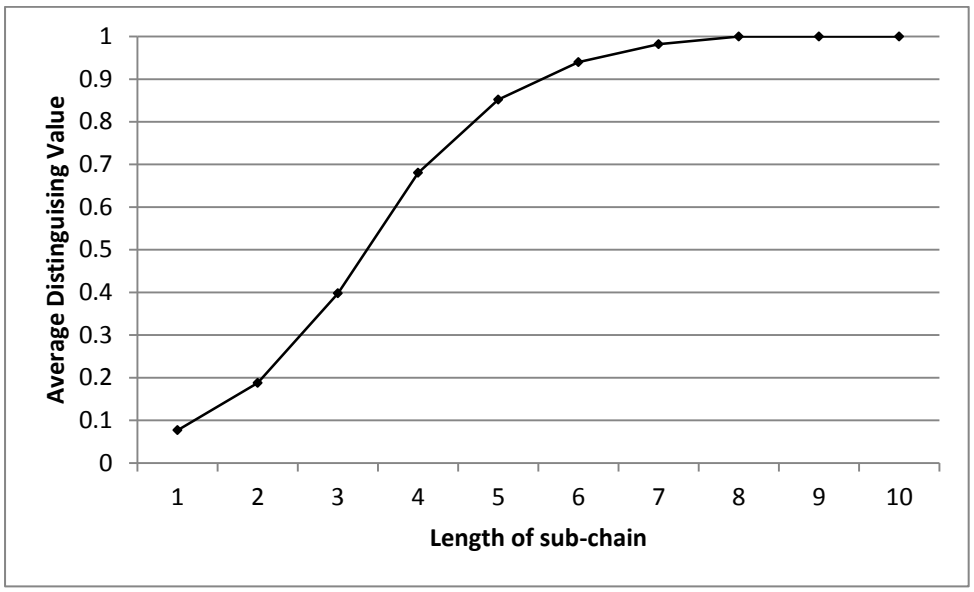


Figure 8.1: The average distinguishing value for all of the first 15 object shapes when comparing sub-chain sizes between 1 and 10.

To better understand the relationships between the differences and similarities of the object shapes heat-maps were drawn for each of the different sub-chain lengths comparing all the possible pairs of object shapes to each other. The heat-maps for ID 0 – ID 47 with sub-chain lengths ranging from 1 – 10 are shown in order in figures 8.2 and 8.3 where the object shape in each row is compared to the object shapes in each column such that every intersection describes the distinguishing value when the row object shape is compared to the column object shape. The diagonal where each object shape is compared to itself always has a distinguishing value of 0. The values are not symmetrical about this line as due to the non-commutative nature of distinguishing value calculation. This increase was expected as the comparison of larger segments of the data-chain would lead to lower likely hood of similar patterns arising.

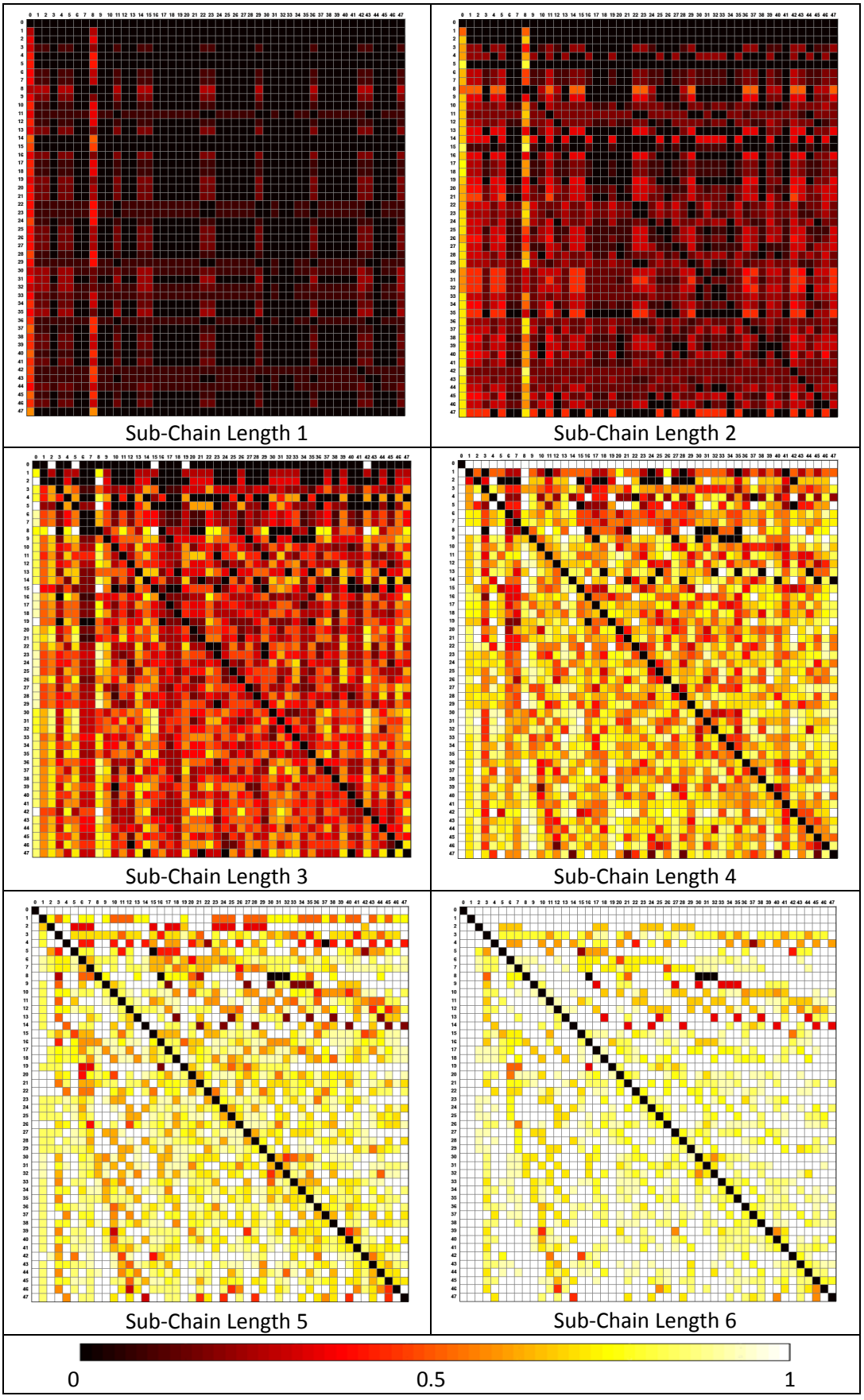


Figure 8.2: Heat-maps showing the distinguishing value of the row object shape from the column object shape for the first 48 objects shapes when compared to each other with a range of sub-chain lengths.

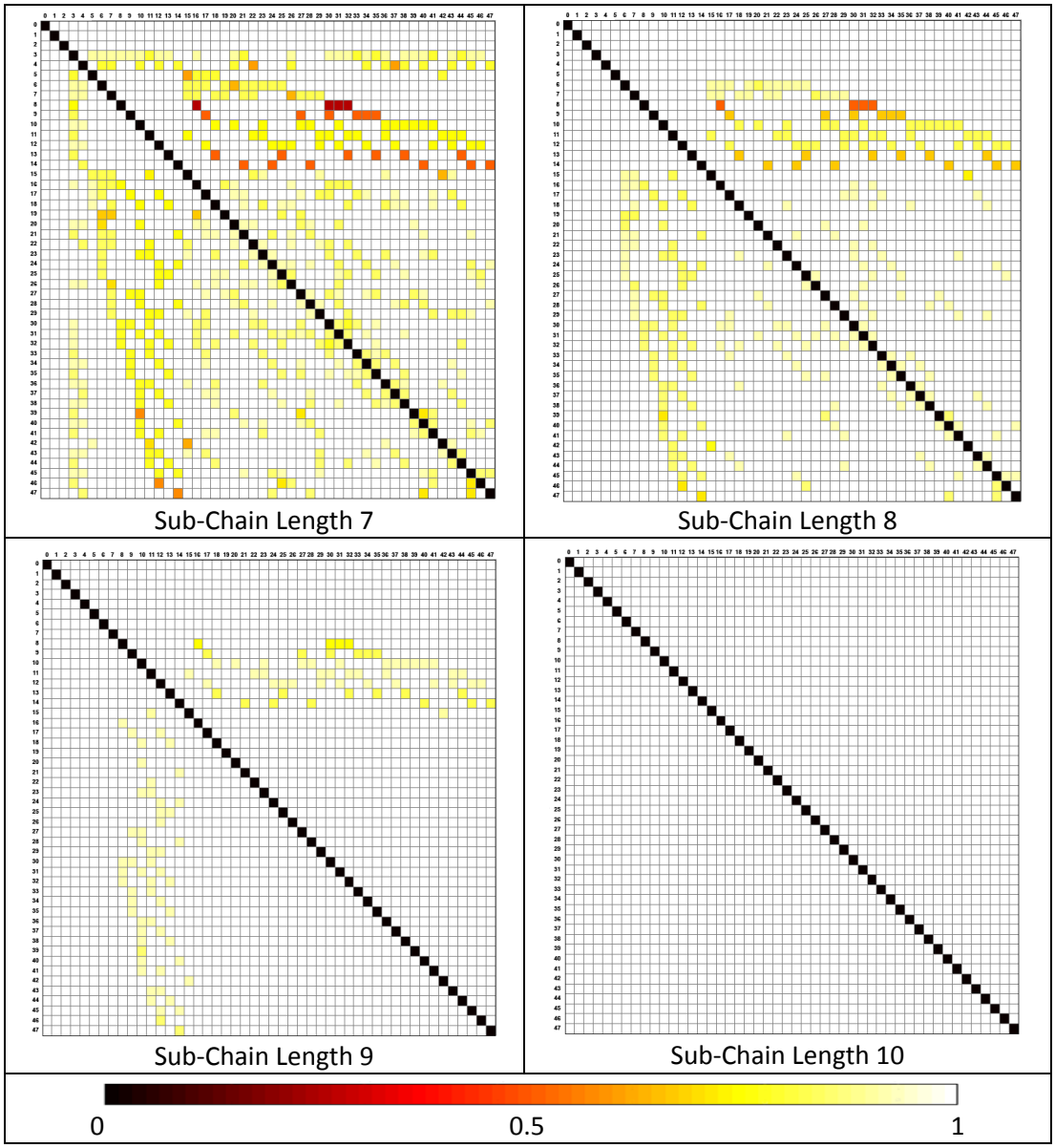


Figure 8.3: Heat-maps showing the distinguishing value of the row object shape from the column object shape for the first 48 objects shapes when compared to each other with a range of sub-chain lengths.

Further inspection of heat-maps, especially those depicting sub-chain lengths five to eight, showed interesting groupings of distinguishing values which appear to step down or across. These groupings were symmetrical in position but not in exact value. These relationships can be seen more clearly in figure 8.4, where the groups are highlighted from the top right of the heat-map in figure 8.3 which shows the clearest of these relations for a sub-chain length of seven. The reason these grouping occurred was because these groups share the same base shape. This result makes sense as the object shapes will have a lot in similar with each other as there is only a single cell difference

between them, which with larger shapes, with longer data-chains, make less of a difference to the sub-chains available.

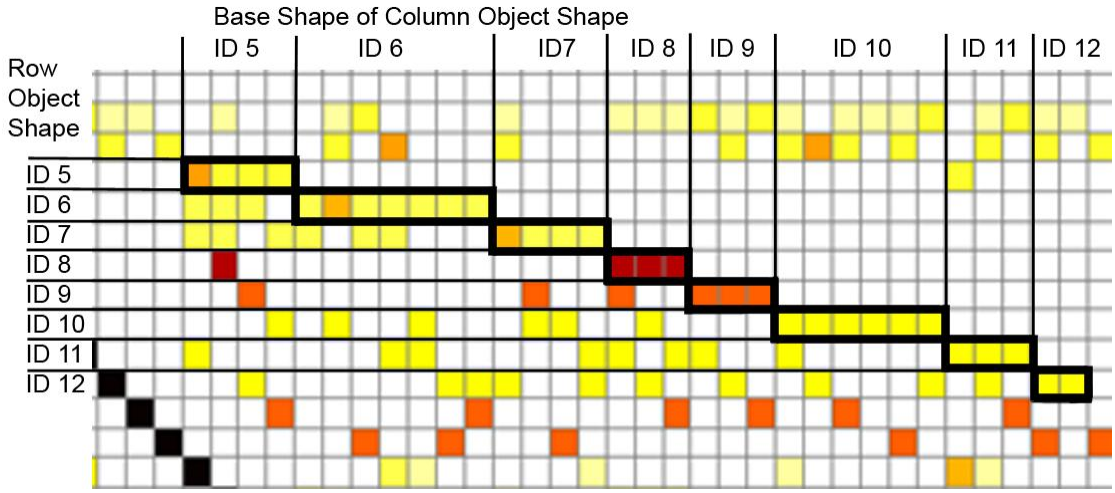


Figure 8.4: Close up of results from sub-chain length 7. The object shapes with the same base shape have similar distinguishing values when compared to the object shape with the same ID as the base shape.

8.2.1 Special Instances of Object Shapes

The heat-maps found give a representation of all the distinguishing values for each sub-chain length. The effect of changing the sub-chain length on the average distinguishing values of each of the object shapes can be seen in figure 8.5. As the sub-chain length increased from 3 to 10 the spread of the values decreased. From the figure outlying results can be found, these indicate where an object shape has a significant average distinguishing value from the rest of the object shapes and therefore identifies interesting object shapes.

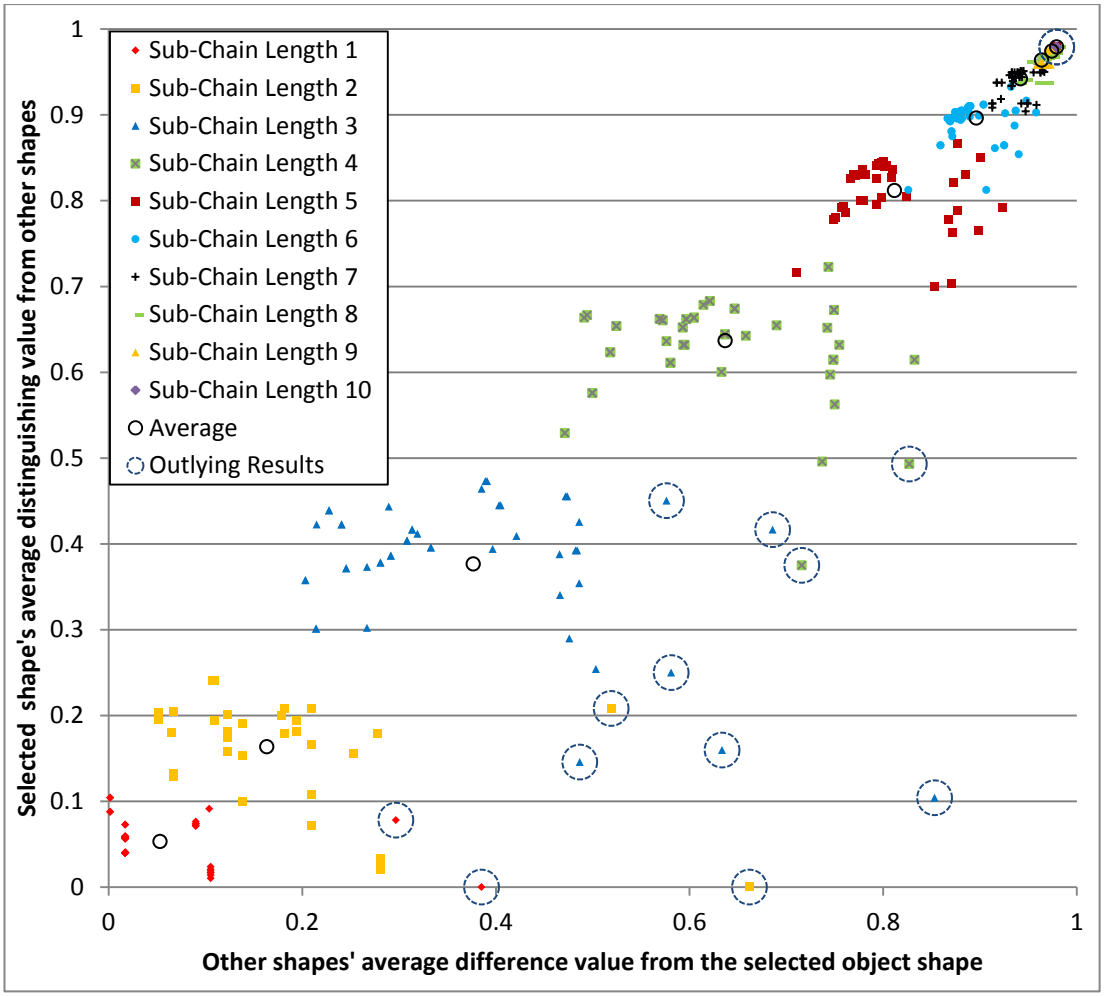


Figure 8.5: The average distinguishing value of an object shape from the other object shapes vs. the average distinguishing value of the other object shapes to that object shape for sub-chain lengths 1 to 10. The outlying results (those 0.2 or further away from the average results of the sub-chain) are highlighted with a ring.

There are a number of object shapes that had distinct distinguishing values from the rest of the object shapes with the same sub-chain length. These outlying results are highlighted by dashed circles in figure 8.5. All of these results were at least a radial distance of 0.2 from that specific sub-chain length group average. These results are all further right than the average. This trend indicates that these object shapes have mostly features which are common in other object shapes and have few or no features of their own that distinguish them from other object shapes. The reverse of this trend would require an object shape that has few or no features which other object shapes have whilst having rare features. From the measurements made this reverse case does not appear to occur in the same scale of magnitude. The reason for this is that the object

shapes will generally have common features when considering relatively low sub-chain lengths and the limited number of values for the links in those sub-chains.

The object shapes which are outlying results are:

- Sub-chain lengths 1 and 2 both have object shapes ID 0 and ID 8 outside of this 0.2 radius.
- Sub-chain length 3 has object shapes ID 0, ID 1, ID 2, ID 5, ID 8 and ID 42 outside this radius.
- Sub-chain length 4 has object shapes ID 0, ID 1 and ID 2 outside this radius.
- Sub-chain length 5 has only object shape ID 0 outside this radius.

All of these outlying object shapes are shown in figure 7.1 in section 7.1 with the exception of object shape ID 42 which is shown in figure 7.12 in section 7.4.1. The reason that these object shapes were outlying from the average is due to the distinct sequence and patterns of numbers that makes up their data-chains.

- Object shape ID 0 was the only shape with only the number 1 in its data-chain.
- Object shapes ID 1, ID 2 and ID 5 had short data-chains, which are repeated through the shape showing rotational symmetry.
- Object shape ID 8 was the only shape with just the numbers 1 and 3 in its data-chain.
- Object shape ID 42 was the only shape with a number 5 in its data-chain.

The number of outlying object shapes for different sub-chain lengths gives further indication to the variation of the distinguishing values. The grouping of the distinguishing values were initially packed close together with sub-chain length 1, they then expanded through to sub-chain length 4 before becoming gradually compacted to a single point at sub-chain length 10.

8.3 Differentiating Object Shapes for hBots

The method used to calculate the distinguishing value between two object shapes considered the terms of the sub-chains in consecutive order. However, this is not how the hBots sense their surroundings. The hBots do not consider the position of their neighbours, but only that it has up to two hBot neighbours each with their own state. Therefore to find a relevant metric, the difference value, consistent with the observations of the hBots in the cooperative object recognition tasks the previous method had to be refined.

8.3.1 Calculating Difference Values for hBots

In comparison with the calculation of distinguishing values for the general object shape comparison the hBot comparison makes two distinct changes. First of these changes was the need to consider only sub-chain lengths of 1, 3 and 5. These lengths represent the required number of hBots to change between the three state-levels, 1, 2 and 3 and provide enough information for the hBots to distinguish most object shapes. To clarify, a hBot at state-level 1 will be occupying the equivalent of one link in the data-chain, a hBot at state-level 2 will have two neighbours and therefore have knowledge of three links in the data-chain, and a hBot at state-level 3 will be neighbouring two hBots that are also neighbouring another two hBots and it will have knowledge about five links in the data-chain. Secondly rather than the terms in the sub-chains being compared directly the sub-chain is used to determine the highest state that a hBot could reach if it was interacting with the part of the object shape that is represented by that sub-chain.

Given a data-chain of an existing object shape it is possible to determine which states at which state-levels are achievable for a group of hBots by examining the sub-chains and using tables 5.1 and 5.2.

- At sub-chain length one the single number in the sub-chain is equal to that of the state of the hBot in contact with the object shape at that same location.
    - (A) maps to state A.
- At sub-chain length three the terms are converted into possible relative hBot states using the shorthand notation of section 5.5.1.
    - (A,B,C) maps to [B][lowest of A and C][highest of A and C] which returns the new current state for the hBot at position B.

- At sub-chain length five two stages of conversion are required. First three, three length sub-chains must be found, producing three states from which the final state can be determined.
    - (A,B,C,D,E) is broken into three (A,B,C) , (B,C,D) and (C,D,E).
        - (A,B,C) maps to [B][lowest of A or C][highest of A or C] giving state P.
        - (B,C,D) maps to [C][lowest of B or D][highest of B or D] giving state Q.
        - (C,D,E) maps to [D][lowest of C or E][highest of C or E] giving state R.
    - Taking P,Q,R in order
        - The triple (P,Q,R) maps to [Q][lowest of P or R][highest of P or R] which gives the final state for the hBot at position C.

For example in Table 8.3 the data-chain of object shape ID 5, {1,1,2,1,2,1,1,2,1,2}, is examined to find the hBots' achievable states at state-level 1, 2 and 3 when interacting with this object shape. These achievable states are compared to those of object shape ID 6, {1,1,1,2,2,1,1,2,1,1,3}, in Table 8.4 where the difference value is found by dividing the number of achievable states that are not present in the other object shape by the length of the data-chain. This represents the number of positions that a hBot could determine that it is next to one object shape and not the other.

Data-Chain	State-Level 1	State-Level 2	State-Level 3
1	1	[1][1][2] = 5	[5][5][10] = 36
1	1	[1][1][2] = 5	[5][5][10] = 36
2	2	[2][1][1] = 10	[10][5][7] = 104
1	1	[1][2][2] = 7	[7][10][10] = 70
2	2	[2][1][1] = 10	[10][5][7] = 104
1	1	[1][1][2] = 5	[5][5][10] = 36
1	1	[1][1][2] = 5	[5][5][10] = 36
2	2	[2][1][1] = 10	[10][5][7] = 104
1	1	[1][2][2] = 7	[7][10][10] = 70
2	2	[2][1][1] = 10	[10][5][7] = 104

Table 8.3. Determining the achievable states for object shape ID 5 {1,1,2,1,2,1,1,2,1,2} at State-Levels 1, 2 and 3.

State-Level 1:		State-Level 2:		State-Level 3:	
Object Shape ID 0	**Object Shape ID 1**	**Object Shape ID 0**	**Object Shape ID 1**	**Object Shape ID 0**	**Object Shape ID 1**
1	1	5	6	36	52
1	1	5	4	36	26
2	1	10	5	104	32
1	2	7	11	70	112
2	2	10	11	104	112
1	1	5	5	36	37
1	1	5	5	36	36
2	2	10	10	104	103
1	1	7	5	70	40
2	1	10	6	104	57
	3		16		184
Difference Values (relative to alternate object shape)					
0/10	1/11	2/10	4/11	6/10	10/11

Table 8.4: Determining the difference values of object shape ID 5 and object shape ID 6. The achievable values which are not possible in the alternate object shape are highlighted red. The difference value realitve to the opposing object shape is calculated by dividing the number of these types of values by the length of the data-chain.

8.3.2 The Difference Values

When considering how hBots interact with object shapes difference values for only the first fifteen object shapes (ID 0 – ID 14) were compared to each other with the exception of object shape ID 11. The reason for not including object shape ID 11 is that this object shape is the only one of the object shapes that contains a number 4 in its data-chain. Including a single object shape with a number 4 in its data-chain would increase the number of state relationships required to be considered at each state-level and therefore increase the complexity of the problem by a factor that would outweigh the benefits of including a single extra object shape in the experiment. Given the current object shapes it was necessary to determine the possible state-relationships that could occur for the hBots, their state-levels and their relationships to each other.

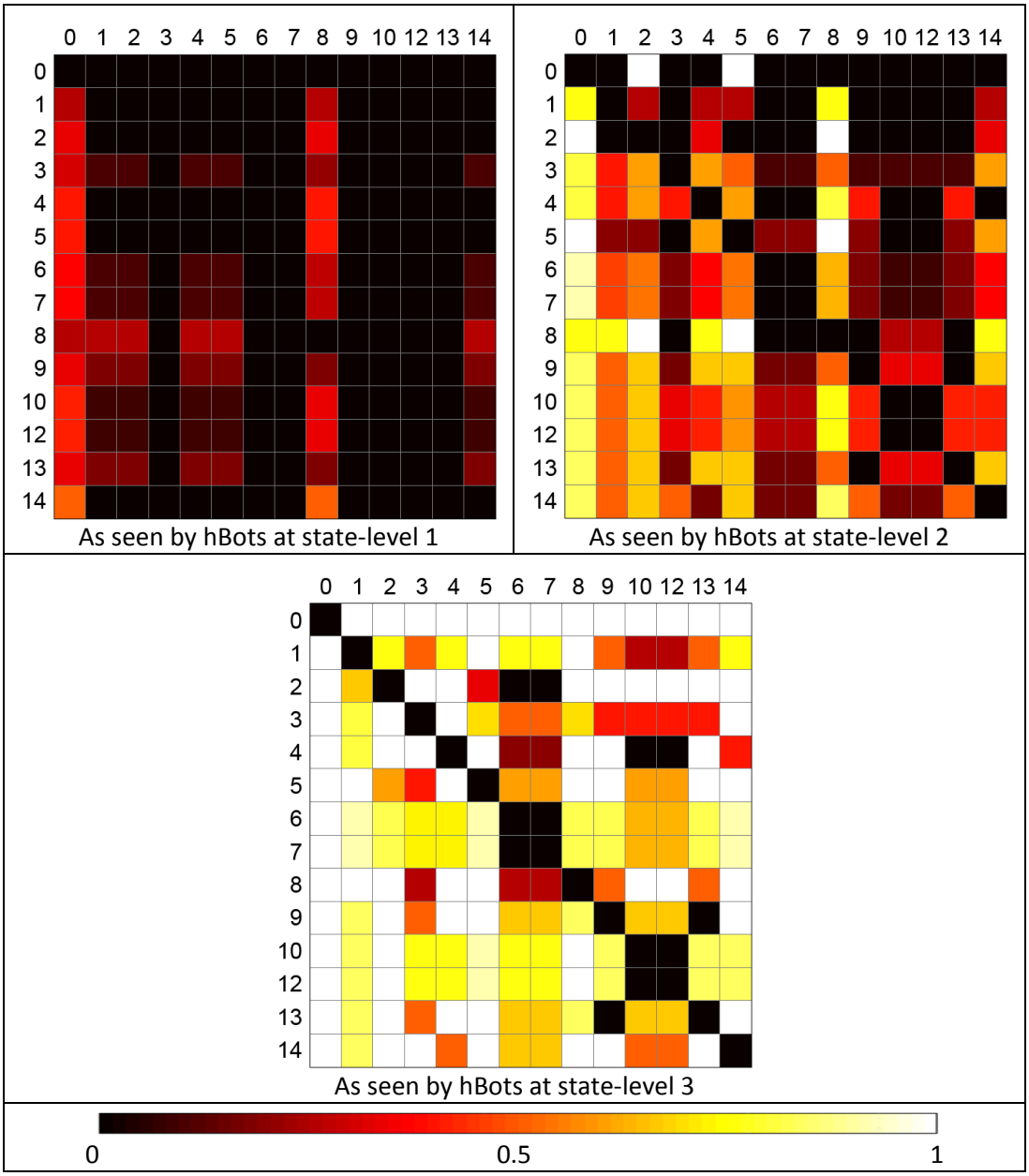


Figure 8.6: Difference Values of row object shape to column object shape including object shapes ID 0 – ID 14 (excluding ID 11). Repeated at three different length sub-chains (1,3,5) representing different hBot state-levels (1,2,3). Black represents no difference between the object shapes and white represents a complete different.

The more information a hBot has about its surroundings, through cooperation, the easier it should be for it to distinguish one object shape type from another. This effect is shown in figure 8.6, the average difference values over the entire heat-map becomes higher as the relative state-level of the hBot increases. Even at state-level 3 there are some object shapes that could not be distinguished from each other. In the majority of these cases this observation was because the object shapes were mirror images of each

other (ID 6 and ID 7; ID 10 and ID 12; ID 9 and ID 13), this can be seen in figure 6.3 where the black boxes which are symmetrical along the diagonal as well as by the identical results for both their rows and columns. It was also not possible to distinguish object shape ID 2 from object shapes ID 6 and ID 7 nor was it possible to distinguish object shape ID 4 from object shapes ID 10 and ID 12 by reaching state-level 3. However, the reverse for all cases was not true. At state-level 1, 2 and 3 it was easier to distinguish most of the other object shapes from object shapes ID 0 and ID 8. This ease of differentiation was because object shape ID 0 had only 1s in its data-chain making any shape that has any other number (all other object shapes) easy to distinguish from it. Object shape ID 8 had no 2s in its data chain and therefore was easy to be distinguished from.

8.4 Comparison to hBot Experiments

To test if the hypothesis from the difference values provided a true indication of how difficult a swarm of hBots would find it to complete a specific task scenario it was necessary to complete a series of simulations. In these experiments object shapes were not moved by the hBots, whether they were valid or invalid. Instead once a hBot reached a state where it had identified the valid object shape, the valid object shape it was in contact with is removed from the arena instantly, as if it had been dissolved or destroyed. The difficulty of the task would be expressed by the amount of time-steps it took to identify all of the valid object shapes whilst ignoring the others. To make this experimentation possible the actions a hBot took for each of its state needed to be determined for each pair of object shapes. There were three potential actions for a hBot to take when in each state, although only one of these actions were assigned to each state:

1. **Identify object shape:** Object shape is instantly removed from arena when an hBot in this state is touching it.

2. **High probability of moving away from object shape:** The hBot has 0.9 chance of changing to state 0 and moving if possible when in this state.

3. **Low probability of moving away from object shape:** The hBot has 0.1 chance of changing to state 0 and moving if possible when in this state.

This number of actions was increased from two possible actions as seen in the initial experimentations (Chapter 6) where there was only one probability, per test, of moving away from an object shape that was being identified. The duel probability levels were added to the hBots in order to further avoid stagnation and for the agents to adapt to different states, where they are more or less likely to move to a higher state-level which could potentially distinguish the object shape. There was also no requirement for the hBots to move any of the object shapes.

These three rules of behaviour were linked to each of the 264 possible states of the hBot determined by the two object shapes used in the scenario and according to which of the object shapes were valid and should be removed, as if dissolved, and which were invalid and should be ignored. To find the suitable behaviour for each of the states a list of all the possible states achievable by a swarm of hBots when interacting with each individual object shape was constructed. Cross referencing two object shapes and the hBots' possible states, whilst considering which was object shape valid, governed the rules chosen for each specific state for a given scenario.

The pseudo code in figure 8.7 explains how the actions for each state was determined when considering two object shapes; one of which valid, the other invalid. In the case that a state is not achievable for either of the object shapes in the scenario, its state-behaviour is not important as it is never acted upon.

At this point it was not clear whether moving with a low probability was the best action where both object shapes have a reachable state in common. It seemed logical in the general case to allow the hBots to arrive at a higher state-level, by having a higher likelihood of staying still, as this is more likely in general to give different distinguishing states. There could be exceptions where the shape to be found has a reachable state-level 1 state (for example state 3) that does not appear in the other shapes data-chain for example ID 10 and ID 14 as shown in figure 8.8. In this case, whether or not it is best for the hBots to move with a high probability unless in behaviour 3 when they remove the shape is unclear. A list of all the states that a hBot could reach at any of the first fifteen object shapes, excluding object shape ID 11, is included in Appendix D.

```
for(int i = 0; i < noOfStates; i++){
    valid[i] = false;   // is state achievable for valid or
    invalid[i] = false; // invalid object shape

    if(state i is a possible state for valid object shape){
        valid[i] = true;
    }
    if(state i is a possible state for invalid object shape){
        invalid[i] = true;
    }

    if(valid[i] == invalid[i]){
        if(valid[i] == true){
            // state achievable for both object shapes
            if(state level[i] == 3){
                // cannot reach a higher state-level
                action[i] = High probability of movement
            }
            else{
                // can reach a higher state-level
                action[i] = Low probability of movement
            }
        }
        else{
            // state not achievable for either object shape
            action[i] = High probability of movement
        }
    }
    else{
        if(valid[i] == true){
            // state only achievable for valid object shape
            action[i] = Remove object shape
        }
        else{
             // state only achievable for invalid object shape
            action[i] = High probability of movement
        }
    }
}
```

Figure 8.7: Pseudo code for determining the action to take for a given state given the possible states achievable for an object shape that is valid and one that invalid.

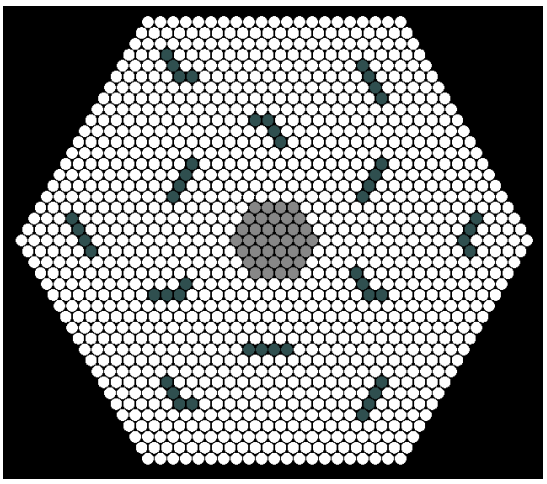


Figure 8.8: The object shape ID 10 with a 3 in its data-chain is valid whilst object shape ID 14 without a 3 in its data-chain is invalid, 37 hBots are in their starting position at the centre of the arena.

8.4.1 The Difference Value and the Number of Time-Steps

Each of the thirty combinations of object shapes with exactly four object cells were repeated fifty times with thirty-seven hBots and the average number of time-steps to complete the task of removing all six required object shapes was recorded. The object shapes were placed in two rings of six, each alternating between valid and invalid object shapes as in figure 8.8. There was no need to consider energy used as the number of agents remained constant, there was also no upper time limit for the experiments. The average of the fifty tests was found and plotted against the average difference value, found from the mean of the three difference values at the sub-chain lengths of 1, 3 and 5. The results from this comparison are shown in figure 8.9, which show a clear correlation between the two sets of data. As expected a pair of object shapes with a higher average difference value required more time-steps before the task was completed. As the difference value increased beyond 0.5 the effect on the number of time-step to complete the task decreased. This relationship between the difference values was further examined, figure 8.10, where the difference value at each of the different sub-chain length 1, 3 and 5, which represent the hBots at state-levels 1, 2 and 3 respectively, was considered. The area under the graphs correlates to the average difference value.

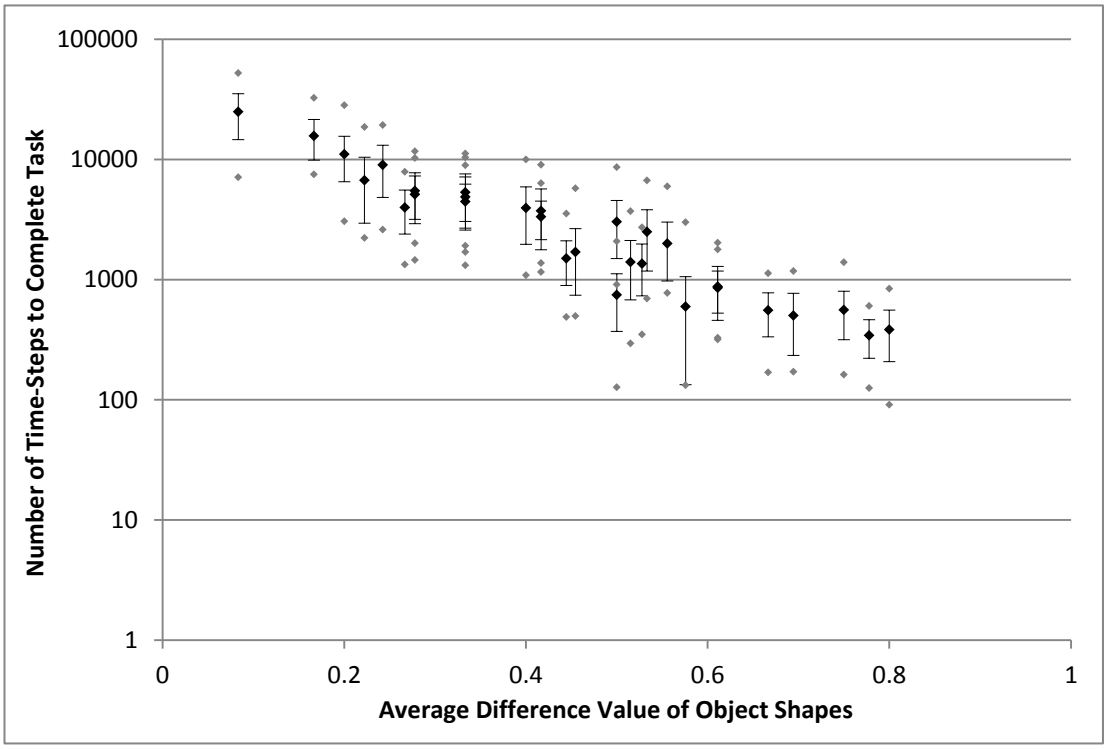


Figure 8.9: Average difference values for each pair of object shapes (ID 5 – ID 14, excluding ID 11) compared with the average number of time-steps for them to complete the task of identifying six required object shapes. The standard deviation and maximum and minimum values are shown.

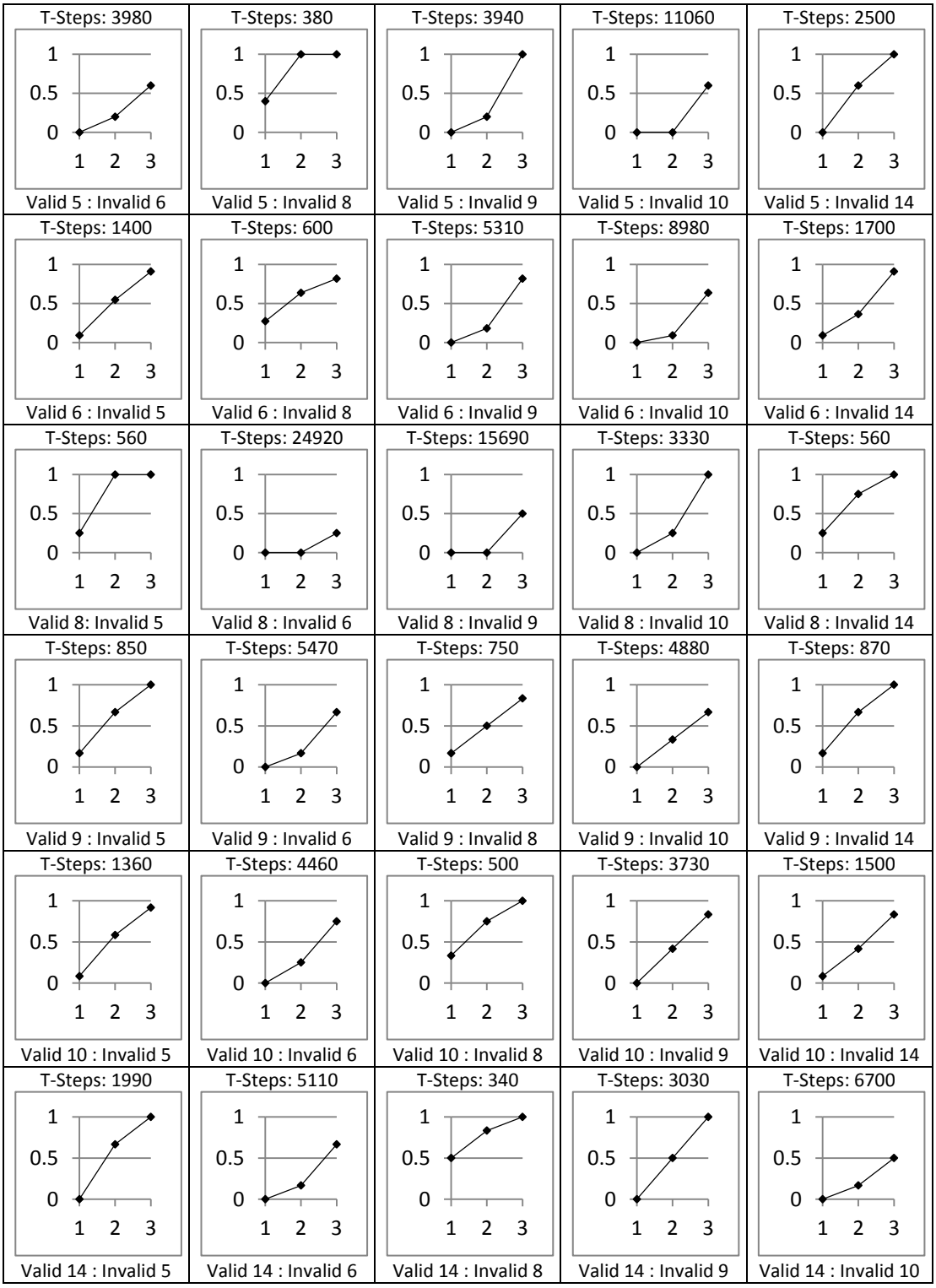


Figure 8.10: The difference value for each equivalent state-level of the hBots (1,2,3) with the average number of time-steps taken to complete the task, to the nearest 50. The area under the graph represents of the average difference value over the three state-levels.

8.5 Summary

A method was devised for determining the difference between two object shapes by considering each sub-chain of a certain length in their respective data-chains. Each sub-chain that one object shape had and the other did not indicated a feature that was present in that object shape and not the other. Increasing the length of the sub-chains, increased the distinguishing value between the object shapes. This increase is because more of the total object shapes boundaries are being considered and therefore there is more information to distinguish between them. Interestingly comparing object shape A to object shape B does not necessarily give the same distinguishing value as the reverse. This is due to the relationship of common and distinguishing features between the object shapes, where one may have a distinguishing feature but the other not.

A number of distinct shapes were found which have the property that they are relatively difficult to be distinguished from other object shapes but easy for other object shapes to be distinguished from them. These object shapes individually tended to have many repeated features (ID 0, ID 1, ID 5, ID 8) or had distinct features, for example being the only shape to have a certain number in their data-chain (ID 42) or not have a certain number in their data-chain (ID 8).

By adapting this system to consider the states a hBot could reach in sub-chains of lengths 1, 3 and 5, and comparing those achievable states to each other rather than the value of the links in the data-chain a metric was devised which gives the difference value between one object shape and another. This metric was shown to be accurate by running a series of experiments which compared the difference values found to the number of time-steps it took to complete that specific object recognition task with a swarm of thirty-seven hBots.

Chapter 9: Training Methodologies

A range of eleven object shape recognition scenarios were selected from those used in section 8.4 to compare two methods for training a swarm of hBots to distinguish between the objects without them initially being aware of the differences in the object shapes. The two methods compared were a genetic algorithm and a random search. Both methods search and test a number of candidate solutions, which detail the behaviours the hBot exhibit at each state. The capabilities of fittest solutions are related to the general solutions derived in section 8.4. The two methods are detailed here with their assessment criteria.

9.1 The Object Shape Recognition Scenarios

Previously, the state-behaviours for the hBots had been determined through a separate analysis of the object shapes they were cooperating to distinguish between. By deriving a method of self-training for the hBots this separate analysis of the object shapes can be removed from the process. The hBots, however, required feedback on their performance in order to train for that scenario. This feedback, in the form of fitness values, was measured by the types of object shapes, valid and invalid, they removed and when they removed them. The scenarios were selected to analyse how the methods deal with tasks with a range of different difficulties.

9.1.1 Scenario Selection

The scenarios chosen were limited by the time available to complete the research whilst maintaining a reasonable degree of complexity. It was found that a maximum of 7000 time-steps would be acceptable per candidate run to give the required number of test repeats, detailed in section 9.2.7. The actual number of time-steps required for different scenarios to complete, with pre-determined state-behaviours, can be found in section 8.4.1. All of the scenarios that have a maximum time-steps to complete the task of less than 8500 and an average below 4000 but above 1300 where chosen. This choice limits the initial study to the eleven scenarios which fall in the middle of the difficulty spectrum. There are nine scenarios which in terms of time-steps taken to complete can be considered to be easier and ten which can be considered more difficult. A summary of the chosen eleven scenarios and the maximum and average run-times are shown in table 9.1. These scenarios show a range of difficulty, where the average number of time-steps taken for the slowest scenario to complete took more than double the fastest.

ID of object shape to be...		The maximum number of time-steps to complete task.	The average number of time-steps to complete task.
Removed	Ignored		
5	6	7877	3980
5	9	9982	3645
5	14	6678	2392
6	5	3712	1400
6	14	5744	1699
8	10	6334	3331
10	5	2724	1355
10	9	9015	3728
10	14	3545	1501
14	5	5969	1993
14	9	8623	2686

Table 9.1: The maximum and average number of time-steps to complete for the eleven chosen scenarios for the GA.

9.1.2 The Cooperative Object Recognition Task used in Training

The arena was set identically to the experiments in section 8.4, with, six valid object shapes and six invalid object shapes. The valid object shapes required removing and the invalid should be ignored. The number of hBots remained constant throughout and was set as thirty-seven. As before, any object shape neighbouring a hBot that performs a remove object shape behaviour is removed instantly as if destroyed. However, unlike previously, during training it was possible for the hBots' state-behaviours to be incorrectly determined and for them to remove some invalid object shapes, mistakes being quantified to provide feedback.

9.1.3 The Training Methods

Each candidate solution, of both methods, would be tested fifteen times in order for averages of the number of the valid and invalid object shapes removed to be calculated as well as the number of time-steps required to remove those object shapes. These averages would be used to determine the fitness values of each candidate solution, detailed in section 8.4. For each scenario both methods were attempted three times to reduce the chance of anomalous results. Ideally this would be much higher, however time limitations permitted only this amount.

The successfulness of the methods would be determined by the fitness values achieved by the best performing candidates and the number of suitable candidate solutions found. This will indicate how easy suitable solutions were to find and the capability of those solutions.

9.1.4 Calculation of Fitness for Candidate Solutions

An evaluation function gives each candidate solution its fitness value, which represents how well it has performed at the scenario. These fitness values can then be used to compare the relative capabilities of the candidate solutions to one other and in the GA method this evaluation function helps determine parent selection and survivor selection over generations of a population.

The purpose of the agents in the cooperative object identification tasks was to distinguish one of the object shapes from the other and remove it from the arena. The object shape type that is to be identified and removed by the agents and the one that is to be ignored by the agents are not known to the agents. These shapes are known by the overarching control system which measures the candidate solutions performance. The fitness value for each candidate solution is determined in the following way.

For each of the fifteen tests on a candidate solution, the number of valid and invalid object shapes removed and the number of time-steps taken to remove the first and last are each recorded.

$$V = number\ of\ valid\ object\ shapes\ removed$$

$$I = number\ of\ invalid\ object\ shapes\ removed$$

$$T = timesteps$$

In the situation that no object shapes are removed from the valid set of object shapes, the time for the first removed is set to the maximum number of time-steps for the experiment and the time for the last removed was set to zero.

$$V_{first} = \begin{cases} T_{max} & if\ no\ valid\ object\ removed \\ T\ to\ remove\ 1st\ valid\ object & otherwise \end{cases}$$

$$V_{last} = \begin{cases} 0 & if\ no\ valid\ object\ removed \\ T\ to\ remove\ last\ valid\ object & otherwise \end{cases}$$

$$I_{first} = \begin{cases} T_{max} & if\ no\ inalid\ object\ removed \\ T\ to\ remove\ 1st\ invalid\ object & otherwise \end{cases}$$

$$I_{last} = \begin{cases} 0 & if\ no\ invalid\ object\ removed \\ T\ to\ remove\ last\ invalid\ object & otherwise \end{cases}$$

As the behaviours of the hBots were not deterministic each candidate solution swarm required multiple runs in order to give a clear indication of their capabilities. The results from these runs were averaged. In the case of the number of object shapes removed, of each type, a median number was found, noted by a ~ above the variable. This method followed the advice given by Bahçeci and Şahin (2005) regarding the avoidance of optimistic functions which combine the performance values obtained from different runs. For the number of steps taken for each of the four variables the mean value was calculated, noted by the — above the variable.

The overall fitness, normalised between 0 and 1, for a candidate solution was found with the following formula:

$$fitness = \frac{154C + 28\left(\tilde{V} - \tilde{I}\right) + \frac{S}{1000} + 336}{672}$$

Where:

$$C = \begin{cases} 1 & if\ \tilde{V} > 0\ and\ \tilde{I} = 0 \\ -1 & if\ \tilde{V} = 0\ and\ \tilde{I} < 0 \\ 0 & otherwise \end{cases}$$

$$S = \left(\bar{I}_{first} + \bar{I}_{last}\right) - \left(\bar{V}_{first} + \bar{V}_{last}\right)$$

Note: $\tilde{V}$is the median of V and $\bar{V}$is the mean of V.

The values chosen for the fitness values are such that:

- Removing only the correct object shapes is more important than the difference in the number of each type of object shape removed.
- The number of each type of object shape removed is more important than the number of time-steps it took to remove them.
- The number of time-steps taken to remove the object shape is the least important factor.

This will allow the GA to rank the candidate solutions by their capability of completing the task.

9.2 Genetic Algorithm Determined State-Behaviours

The first method used to determine the state-behaviours of the hBots was a genetic algorithm or GA. There are a number of key components to creating a GA that can be utilised to solve any given problem. These components needed to be considered carefully in order for there to be a suitable correspondence between the way the GA searches the problem space for the optimal solution. In a GA the phenotype describes how the solution reacts in the simulated world, whilst the genotype encodes the phenotype so that the operations within the GA can be performed on it. The components of a GA include; population, representation, recombination, mutation, evaluation function, parent selection mechanism, survivor selection mechanism, initialisation and termination condition.

9.2.1 Representation of Candidate Solutions

Each candidate solution in the GA represents a swarm of homogeneous hBots, which all have identical state-behaviours determined by the genome. Other aspects of the hBots control were not affected by the GA, including, how they moved and how they switched between the different states dependant on their neighbouring hBots' states.

The problem being solved by the GA is, which of the three possible behaviours should the hBot perform when in a given state. The bits of the genome represent the behaviour performed at each of the available states. Although there are a maximum of 264 states, only 41 states need to be considered in the genotype as the other states are not possible to achieve with the object shapes in the scenarios used, the reasons for this choice are discussed further in section 9.2.3. As the hBots were not initially aware of any of the differences between the valid and invalid object shapes all three behaviours need determining by the training method. The three possible behaviours the hBot can perform were:

1. Remove the object shape
2. A 0.9 chance of moving away from object shape.
3. A 0.1 chance of moving away from object shape.

In summary, the GA genome has an integer representation with a restricted set of {1,2,3} where each candidate solution represents the state-behaviour relationships for each agent of an entire swarm of hBots.

9.2.2 Recombination

Recombination is a variation operator which merges the genetic information of two parent genomes in order to produce offspring. The offspring shares some of the features of both the parents in the hope that it will be better suited to its environment, although this is not always the case.

In N-point crossover there is a distinct point in which the information is no longer taken from one parent's genome but is taken from the other to form the offspring's genome. The crossover point is selected at random, but will not happen before the first bit or after the last bit. The N in N-point crossover represents the number of times that this switch occurs. In the case that N is 2, two random points are selected along the length of the genomes at which point the information is taken from the other parent. An example of two-point crossover is given in figure 9.1.

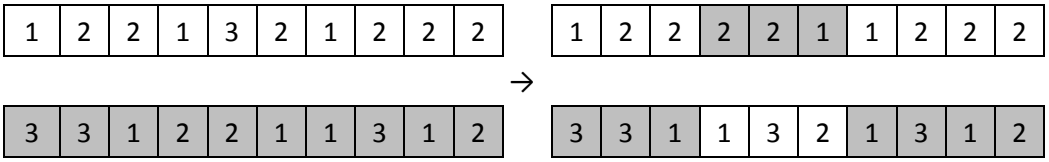


Figure 9.1: An example of Two-point crossover for an integer representation genetic algorithm.

The GA for this task used a two-point crossover as appose to any other crossover methods available such as uniform crossover. This method was chosen as it more closely represents the relationships of the state-rules which also split into three sections due to state-level 1 having three states. This would increase the likelihood that related state-rules would be kept together. However, to make this relationship apparent it was necessary to reorder the states to represent the relationships they had to each other.

There were a total of 264 states (excluding state 0, which was searching for an object). A hBot in state 1 could only potentially move to states 4 to 9 and states 22 to 102, the relationships of the states and state-levels are shown in figure 9.2. Further those hBots currently in state 6 could only potentially move to states 49 to 54. The order the states were represented in the genotype were determined by doing a depth first search of the state-rule relationships. By doing this the related states were placed together in the genotype . The order would be as follows where triple dots represent missing numbers:

- 1, 4, 22, 23, 24, 25, 26, 27, 5, 28, 29 … 47, 48, 6, 49, 50, 51, 52, 53, 54, 7, 55, 56 … 74, 75 and so on.

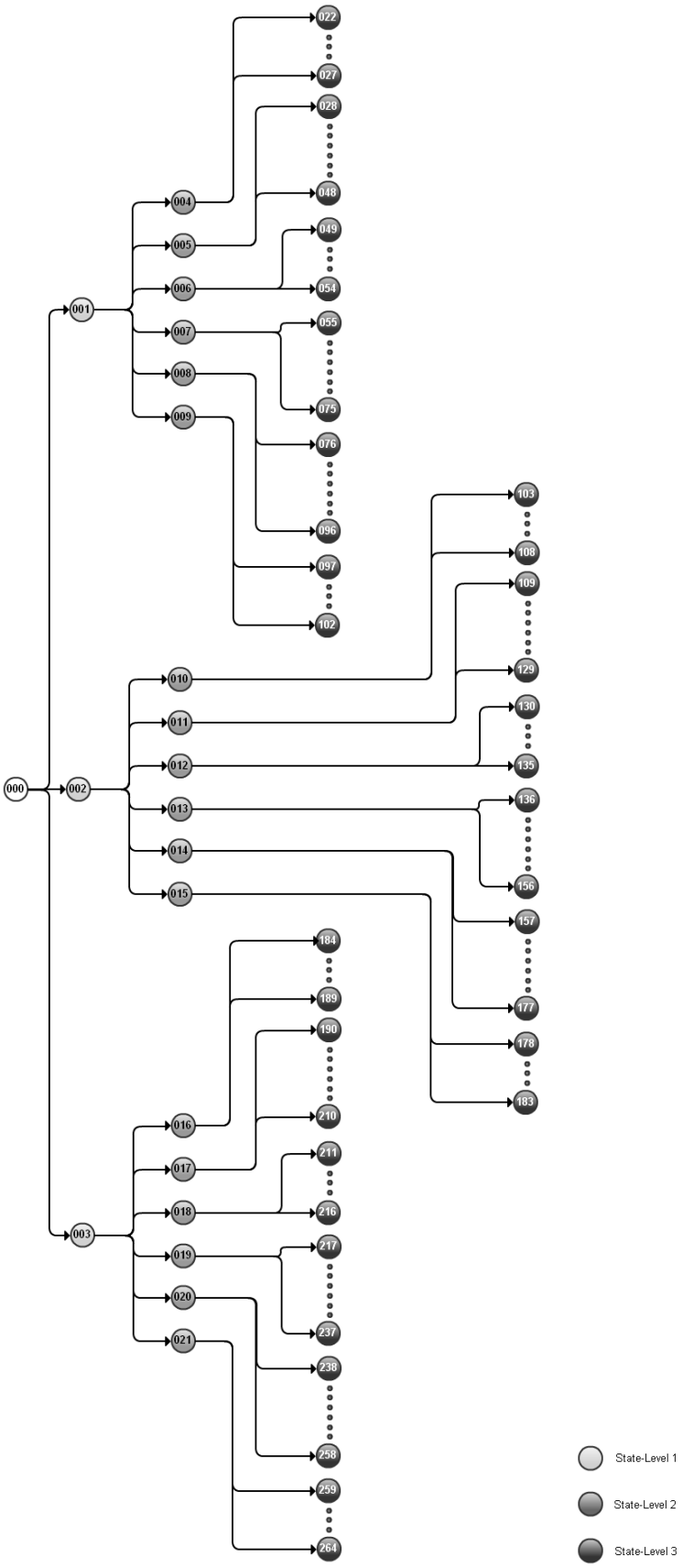


Figure 9.2: The state-rules relationship. A hBot in any state can only move to a higher state it is connected to in the direction of the arrow.

9.2.3 Mutation Operator

Mutation is the second variation operator, it changes individual bits of a candidate solution's genome with a given probability.

As an integer representation was chosen the mutation operator chosen was a variation of the random resetting operator, where any bit selected for mutation will not return to its current state. In this GA there are three possible values for each bit [1,2,3] in the genome which each represent a different action. Therefore if a bit was at value 2 and was selected to be mutated it will change to either 1 or 3 with equal probability. This choice was made to increase the variance, as the standard mutation rate is decreased by the unachievable states for a given scenario.

The probability that any bit will change is known as the mutation rate. The mutation rate is commonly set as 1/L where L is the length of the genome. There was a potential issue with this value of mutation in the case of the state-behaviour rule GA in that not all the states and therefore parts of the genome would be used for each scenario. If the states are not used as they cannot be reached by a hBot trying to identify either of the object shapes, in that scenario, any mutation that then happens at the relevant bit of the genome becomes the equivalent of doing nothing. Therefore the mutation rate would be artificially decreased by a small amount in general, and would technically vary from scenario to scenario dependant on the object shapes in the scenario. However, this method was still determined to be the best method of choosing the mutation rate. The effect of this problem was reduced for the purposes of the experiment by only using states in the genome that could be reached by the object shapes ID 0 to ID 14, excluding object shape ID 11 as it is not used. This decision reduced the total number of states from 264 to 41. In future research it would be appropriate to determine what cause this change in mutation rate might do to affect the evolutionary process. The 41 states in the order they were represented in the genome were:

- 1, 4, 22, 25, 26, 27, 5, 31, 32, 33, 36, 37, 40, 6, 52, 53, 57, 7, 70, 71, 8, 79, 2, 10, 103, 104, 105, 11, 112, 113, 117, 12, 133, 13, 151, 3, 16, 184, 185, 17, 193

9.2.4 Population Model

The population for any specific generation acts as a record of the current, existing genomes on which an evaluation function takes place. Populations are generally defined by their size and members, where individuals in the population are static the population itself varies. For the object recognition task the population is formed from a range of candidate solutions. Some populations also contain spatial structures where the position of the members of the population play a role in determining their fitness, Cantú-Paz (1998) discusses this in more detail.

The population model describes the way in which the current population of parents and offspring is reduced to the required population size after the recombination process. It was determined that the generation model would be used for the state-behaviour rule selection GA. This choice increased the simplicity of the survivor selection model as all parents are replaced by an equal number of offspring in each generation.

9.2.5 Parent Selection

Parent selection is the method by which the parents of the next generation of candidate solutions are selected by comparing their fitness values.

The method chosen for this GA was tournament selection, with size 4. The size of a tournament selection indicates the number of candidates that are taken from the population and ranked against each other. The highest ranked candidate is added to the parent pool and the remainder are returned to the potential pool so they may get picked again. This process is repeated until there are enough parents in the parent pool for producing offspring. In the state-behaviour GA the parent pool was equal to half the population.

Tournament selection was chosen over fitness proportional selection as it allowed candidate fitness' to be compared without the magnitude of the difference in their values effecting the outcome. This choice was important due to the way fitness is measured in the system, section 9.1.4.

9.2.7 Initialisation and Termination Condition

The final aspects of the GA that needed determining were both the initialisation and the termination condition. Random seeding was selected for the state-behaviour GA as the most common initialisation condition and it represents the lack of initial information the hBots have about the scenario.

The termination condition can be set in a number of different ways. The evolutionary algorithm can continue until the following cases as provided by Eiben and Smith (2007, p. 24):

- The maximally allowed CPU time elapses
- The total number of fitness calculations reaches a given limit.
- The fitness improvement remains under a threshold value for a given period of time (i.e. a number of generations or fitness evaluations).
- The population diversity drops under a given threshold.

The upper boundary for the number of time-steps in a test was determined by considering the total amount of run time available for the experiment and how long it took for the scenarios to be completed in the previous experiment, section 8.4.1. The largest limiting factor was the time it would take to complete a single complete run of the GA for a number of different scenarios which would each require repeating themselves.

To calculate the time it would take to run the GA a number of factors were considered; how long the program took to run a single time-step, the maximum number of time-steps in a run, the number of runs each candidate solution of the population gets before averaging, the population size, the maximum number of generations. The number of different scenarios tested and the number of repeats for each scenario were not crucial to the calculation of time, but limited by the number of tests that could be run in parallel.

Each of these factors had to be balanced in unison in order to give the GA enough freedom to explore the possible solutions whilst not running for an unsuitable amount of time. It was determined that the parameters suitable for running a GA scenario were as in table 9.2.

Population Size	30
Tests per member of population	15
Maximum number of time-steps per test	7000
Maximum number of generations	30
Total number of time-steps	94500000

Table 9.2: The parameters for the genetic algorithm test.

9.3 Randomly Determined State-Behaviours

In general a GA is a time consuming way of solving problems, in some cases the time it can take outweigh the benefits of utilisation. If there are other less computationally heavy ways of finding the same solution or similarly capable solutions to the problem they should be used as a matter of efficiency. One of the simplest ways to potentially derive a suitable solution is to randomly generate them. This method is equivalent of having a large initial population of candidate solutions in a GA but only having a single generation, so there is no iterative improvement through the recombination and mutation of members of the population.

In this series of tests, each of the eleven scenarios was also tested with 900 randomly generated candidate solutions, this is equivalent to the GA which had 30 generations from a population of 30 candidate solutions. Each of the candidate solutions had fifteen repeated runs with the object shapes reset, as with the GA, to allow for the variance in results and their fitness calculated using the same formula, section 9.1.4, so the two methods could be compared. Each of the eleven scenarios were repeated three times with a new set of candidate solutions.

9.4 Comparison of Methods

To discover whether both the GA solutions and the randomly derived solutions were in fact suitable they needed to be compared to the general solution that was derived by a series of generic rules for the experiment in section 8.4. These were run again with the same experimental set up as the GA and the randomly derived experiments in order to give a comparable fitness value.

As an analyses of the object shape pairs for the different scenarios had been found, section 8.3.2, predictions on the difficulty for the GA or the random procedure could be made. Table 9.3 describes the predicted difficulty of the tasks as determined by:

- The difference values of the data-chains of the object shapes at sub-chain lengths 1, 3 and 5.
- The differences in the number of identifying states for the valid object shape and the level of those states.

For example, where the valid shape has a low difference value to the invalid shape it will be more difficult to identify and remove the valid object shape. This number is related to but not the same as the number of identifying states. In the case of the number of identifying states, the number represents how many different potential states could be used to distinguish the valid object shapes from the invalid object shapes out of the 41 states. A lower number is more difficult as there are less potential solutions. The higher the level required also increased the difficulty as to reach those higher level states requires the correct number of hBots to interact with each other. state-level 1 requires a single hBot, state level 2 requires three neighbouring hBots and state-level 3 requires five neighbouring hBots.

Object Shape		**Difference Value of sub-chain lengths**				**Number of Identifying States per all states at each State-Level**				**Difference in identifying states between valid and invalid object shape**			
Find	**Ignore**	**1**	**2**	**3**	**MEAN**	**Lvl 1**	**Lvl 2**	**Lvl 3**	**Total**	**Lvl 1**	**Lvl 2**	**Lvl 3**	**Total**
5	6	0.00	0.20	0.60	0.26	0.00	0.02	0.05	0.07	-0.02	-0.07	-0.17	-0.27
5	9	0.00	0.20	1.00	0.40	0.00	0.02	0.07	0.10	-0.02	-0.07	-0.07	-0.17
5	14	0.00	0.60	1.00	0.53	0.00	0.05	0.07	0.12	0.00	-0.02	-0.02	-0.05
6	5	0.09	0.55	0.90	0.52	0.02	0.10	0.22	0.34	0.02	0.07	0.17	0.27
6	14	0.09	0.36	0.90	0.45	0.02	0.07	0.22	0.32	0.02	0.05	0.15	0.22
8	10	0.00	0.25	1.00	0.42	0.00	0.02	0.07	0.10	-0.02	-0.12	-0.22	-0.37
10	5	0.08	0.58	0.90	0.53	0.02	0.12	0.27	0.41	0.02	0.12	0.22	0.37
10	14	0.08	0.42	0.83	0.44	0.02	0.12	0.24	0.39	0.00	0.07	0.12	0.27
10	9	0.00	0.42	0.83	0.41	0.00	0.10	0.24	0.34	0.02	0.07	0.15	0.24
14	5	0.00	0.67	1.00	0.56	0.00	0.07	0.10	0.17	0.00	0.02	0.03	0.05
14	9	0.00	0.50	1.00	0.50	0.00	0.05	0.10	0.15	-0.02	-0.05	-0.05	-0.12

Table 9.3: The difference values for three sub-chain lengths' and the number of identifying states at each state-level for the object to be found. Higher values suggest an easier scenario to find a solution to.

9.5 Summary

From the experiment in section 8.4 the difficulty of completing cooperative object recognition scenarios was assessed by the number of time-steps taken to complete them. Eleven of these scenarios were chosen which showed a range of difficulty based on that assessment. Two methods were devised for determining the behaviours of the hBots at each state so they may learn to distinguish between two object shapes without initially knowing their differences. This was done by giving the hBots feedback on how well they had completed the task. The first method is that of a GA. Each candidate solution

is a swarm of homogeneous hBots. The genome is an integer representation with a restricted set, using 2-point crossover recombination and a variation of random resetting for the mutation operator. A tournament selection with size 4 is used, with a generation model. There are thirty candidate solutions run for thirty generations. The second method is that of randomly determined state-behaviours. This is equivalent to a GA with random seeding, a single generation and 900 candidate solutions. Both these methods will be compared by utilising a set of fitness values determined by the generic rule system for each of the scenarios to determine their capability to determine suitable state-behaviours.

Chapter 10: Analyses of Training Methods

The results for both the GA and random training methods are analysed and compared to determine their suitability. This analyses takes into account the difficulty of finding the suitable solutions to all eleven selected scenarios as well as the overall capability of the solutions found. The best performing candidate solutions for each method were re-tested to determine how they perform over an extended duration. Specific consideration is given to the results showing how the hBots interact with the invalid object shape.

10.1 Results of the Genetic Algorithm

The GA, detailed in Chapter 9 and written in full in Appendix E, was run three times for each of the eleven scenarios. The following notation is used for each scenario:

- F5I6A: Find and remove valid object shape ID5 whilst ignoring invalid object shape ID6 experiment run A.

The fitness results of all the experiments are presented in figures 10.1, 10.2 and 10.3, where the blue box shows the first and third quartile and the red line the median, the whiskers indicate the maximum and minimum fitness values for that generation. Each generation has thirty candidate solutions. A suitable solution to the scenario is declared as one where, over the average of the fifteen repeated candidate runs, that candidate solution removed all of the six valid object shapes whilst not removing the invalid object shapes within the allotted 7000 time-steps. This requires the candidate solution to have a fitness value of above 0.9792.

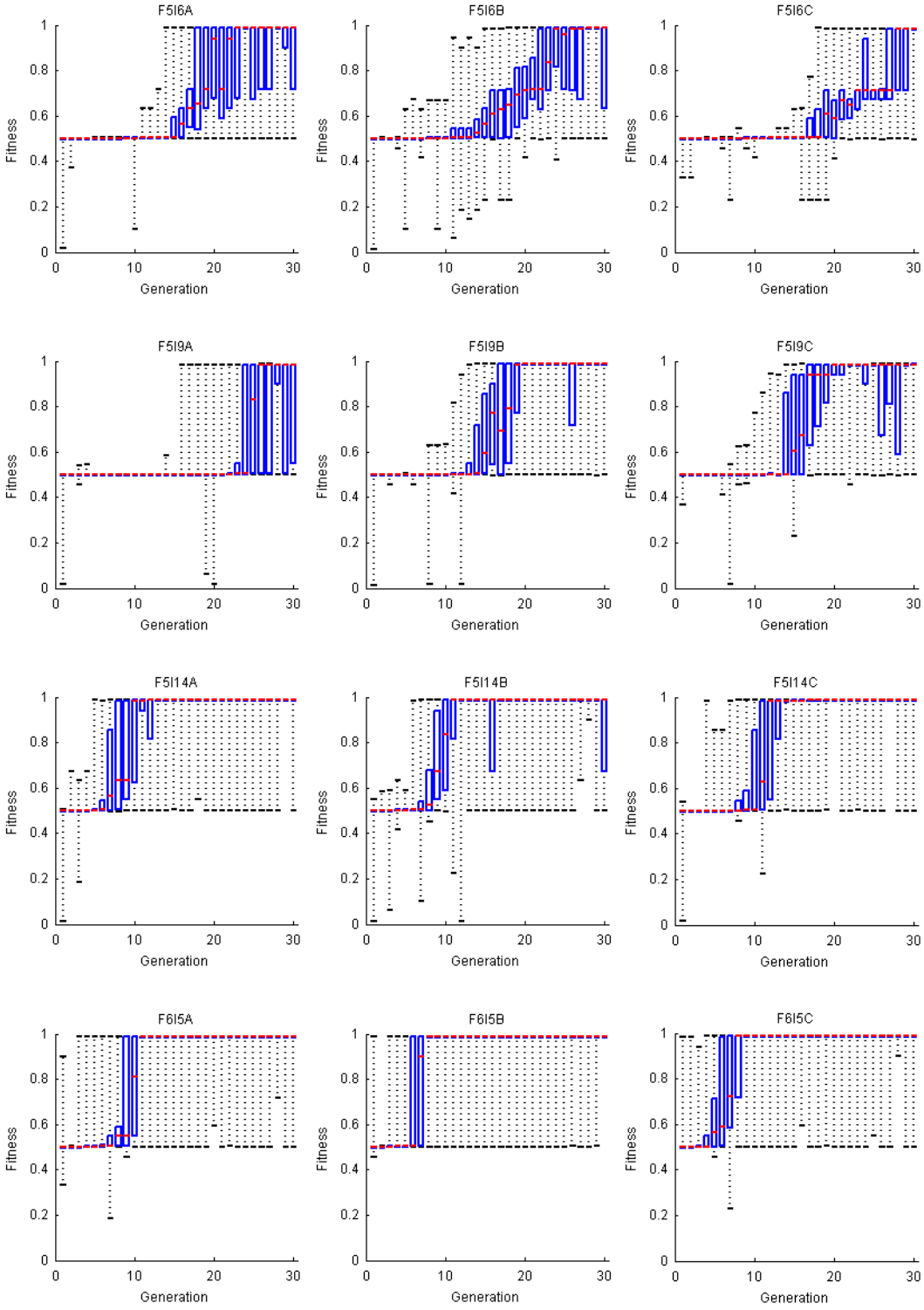


Figure 10.1: Boxplot of fitness values for subsequent generation population for scenarios F5I6, F5I9, F5I14 and F6I5, with whiskers showing the minimum and maximum fitness values.

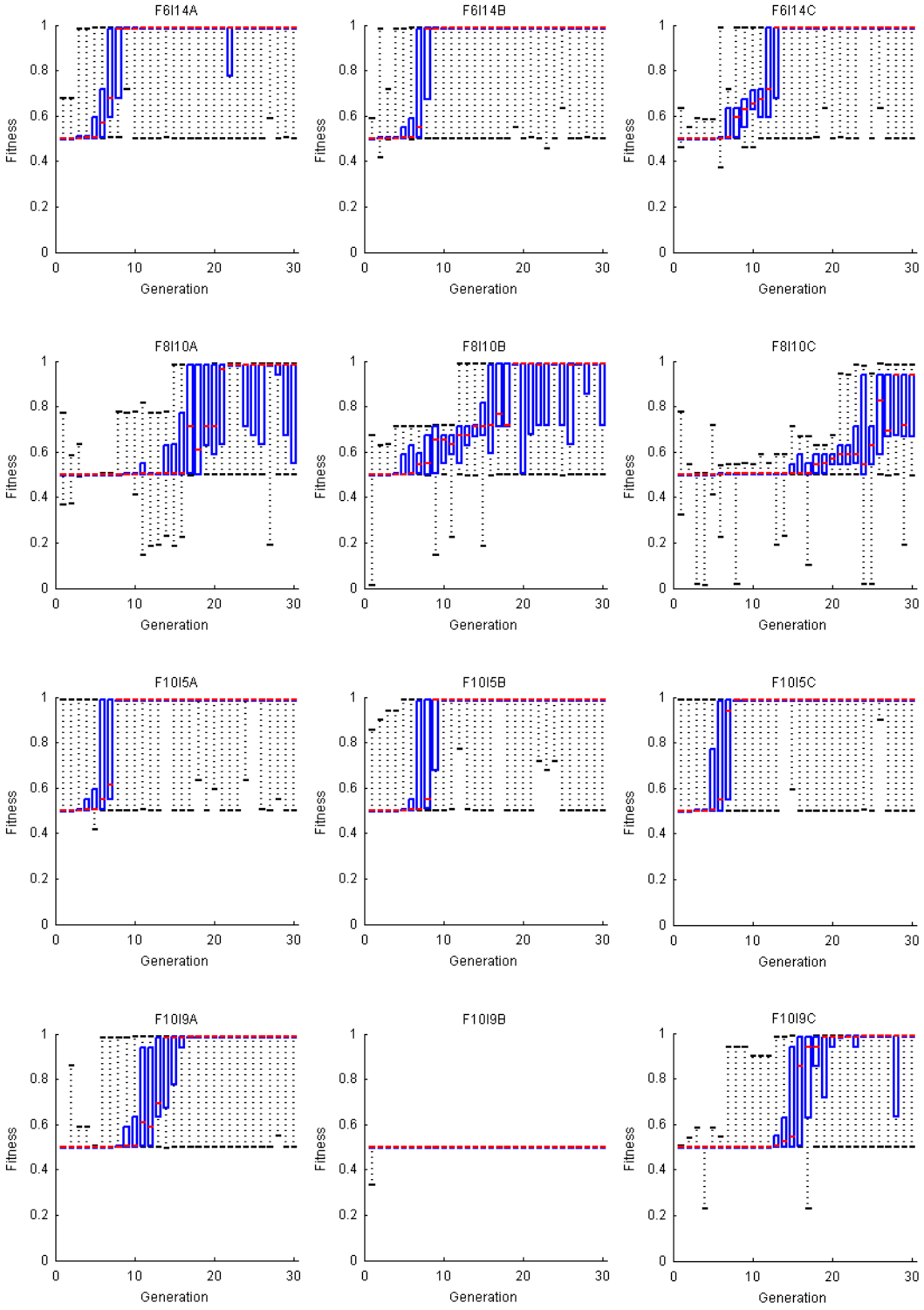


Figure 10.2: Boxplot of fitness values for subsequent generation population for scenarios F6I14, F8I10, F10I5 and F10I9, with whiskers showing the minimum and maximum fitness values.

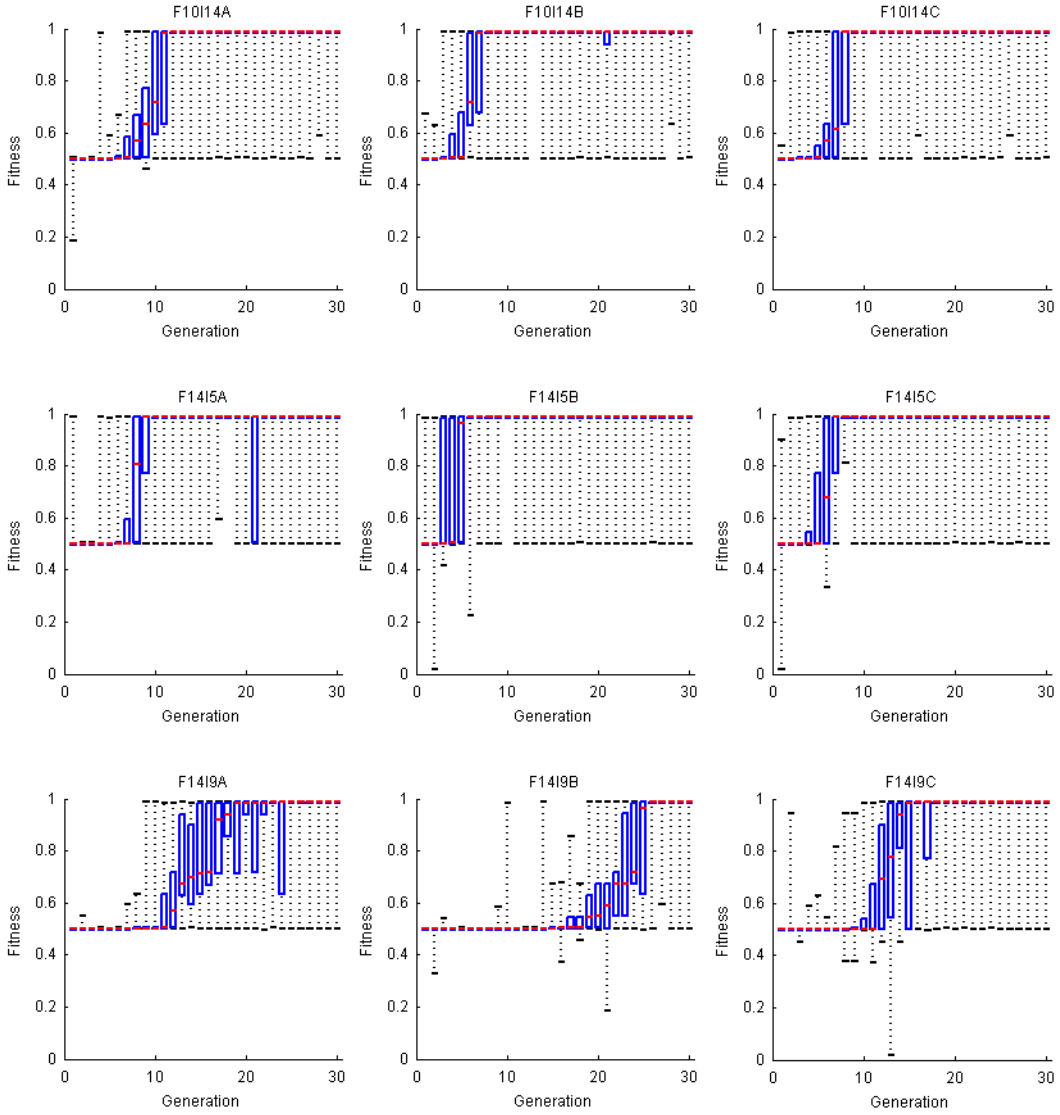


Figure 10.3: Boxplot of fitness values for subsequent generation population for scenarios F10I14, F14I5, and F14I9, with whiskers showing the minimum and maximum fitness values.

For all but one of the scenarios a suitable solution was found for each of the three repeated runs of the GA. The exception to this is F10I9B, the box plot of which is shown in figure 10.2, where there is no visible improvement in the population.

From closer inspections of this scenario's population it was found that for F10I9B each of the candidate solutions removed all of both types of object shape on average, with the exception of one solution that removed less of the valid object shapes for the first generation. However, there was marginal improvement in the maximum values which indicates that although all the object shapes were removed, the valid ones were being removed earlier in subsequent generations and the invalid ones later. Despite the single GA run where a suitable solution was not found the other two runs for F10I9 indicate a

scenario which is neither simple or difficult relative to the other scenarios, based on the shape of the boxplots alone. This suggests that the random seeding of the first generation or the amount of randomness due to the recombination, mutation and the tournament selection methods can affect the outcome of the GA.

The difficulty of the scenarios can be approximated by the shape of the boxplots over the three repeated runs. If the populations' fitness increased rapidly early it would suggest an easier problem to solve. For example F6I14 and F10I5, shown in figure 10.2. Both scenarios have similar results across the three repeated runs, where a suitable candidate solution is found within the first five generations.

Examples of scenarios which suitable solutions where more difficult to find are F5I6 and F8I10, as shown in figures 10.1and 10.2 respectively. In both of these scenarios the improvement in the population between generations is more gradual when compared to the scenarios which appear simpler. A suitable candidate solution that removes all six valid object shapes is not found until approximately generation fifteen in scenarios F5I6 and F8I10.

10.1.1 Comparison to Predicted Difficulty Metric of Scenarios

In section 9.4, a range of predicted difficulties were determined for the eleven scenarios. These difference values, number of identifying states and the number of time-steps to compete when given the generic solution all relate to the difficulty of the scenario. From examining the results of the GA it was identified that F6I14 and F10I5 are examples of simpler scenarios and, F5I6 and F8I10 are examples of more difficult scenarios. The predicted relative difficulties are shown for F6I14 and F10I5 are shown in table 10.1 and the values for F5I6 and F8I10 are shown in table 10.2.

Object Shape		**Difference Value of sub-chains lengths**				**Fraction of Identifying States at each State-Level**				**Difference in fraction of valid and invalid identifying states**			
Find	**Ignore**	**1**	**2**	**3**	**MEAN**	**Lvl 1**	**Lvl 2**	**Lvl 3**	**Total**	**Lvl 1**	**Lvl 2**	**Lvl 3**	**Total**
6	14	0.09	0.36	0.90	0.45	0.02	0.07	0.22	0.32	0.02	0.05	0.15	0.22
10	5	0.08	0.58	0.90	0.53	0.02	0.12	0.27	0.41	0.02	0.12	0.22	0.37

Table 10.1: Two scenarios that are predicted to be relatively simple to solve due to the high number of identifying states and the higher number of difference of sub-chain lengths for the find and ignore data chains.

Object Shape		Difference Value of sub-chains lengths				Fraction of Identifying States at each State-Level				Difference in fraction of valid and invalid identifying states			
Find	Ignore	1	2	3	MEAN	Lvl 1	Lvl 2	Lvl 3	Total	Lvl 1	Lvl 2	Lvl 3	Total
5	6	0.00	0.20	0.60	0.26	0.00	0.02	0.05	0.07	-0.02	-0.07	-0.17	-0.27
8	10	0.00	0.25	1.00	0.42	0.00	0.02	0.07	0.10	-0.02	-0.12	-0.22	-0.37

Table 10.2: Two scenarios that are predicted to be relatively difficult to solve due to the high number of identifying states and the higher number of difference of sub-chain lengths for the find and ignore data chains.

The predicted difficulties, where lower numbers are considered indicators of a more difficult scenario, correlate to the results from the GA runs. There is one exception where the value is contrary to the rest at sub-chain length 3, F8I10, has a difference value of 1.0. This shows that once a hBot reaches state-level 3 it will definitely be able to determine if it is next to object shape ID8 or not. However, this is the most difficult state-level to reach as it requires five hBots interacting with each other, making it less likely to be a factor compared to state-level 1 and state-level 2.

These factors for predicting the difficulty of finding a suitable solution were examined in more detail across the eleven scenarios. This examination was to determine the strength of correlation between the predicted difficulty values and the difficulty perceived for the GA to find the solutions. The difficulty that the GA had in determining a suitable solution was measured by the average number of successful candidate solutions of the three runs of the GA. This gives an indication of how quickly the solutions were found and the consistency of those solutions.

This perceived difficulty from the results was compared to the varied predicted difficulties of the scenario, where the predicted difficulty was measured in four ways:

- The average difference values of the sub-chain lengths, 1, 2 and 3, figure 10.4.
- The total number of identifying states which distinguish the valid object from the object to be ignored divided by the total number of states, figure 10.5.
- The average number of time-steps it took to complete the same scenario when there was no restriction on the amount of time-steps, figure 10.6.
- The difference in the total number of identifying states for the valid and the invalid object shape, divided by the total number of states, figure 10.7.

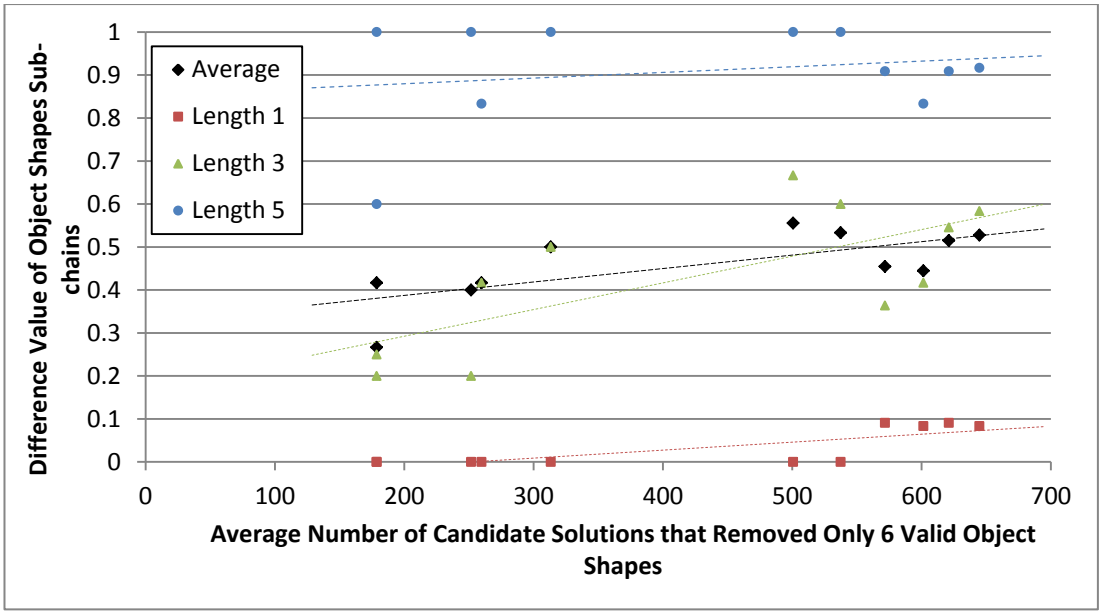


Figure 10.4: The predicted difficulty due to the average difference in the sub-chain length against the measured difficulty of the genetic algorithm to solve the relevant scenario.

Considering the average calculated difference values of a pair of object shapes for all three different length sub-chains a coefficient of correlation equal to 0.70 ($p<0.02$) was found, figure 10.4. This correlation suggests that a lower difference value for the sub-chains of the data-chain present a scenario which is more difficult to find a suitable solution for. The correlation was more significant for sub-chain length 1, 0.79 ($p<0.01$), and decreases for sub-chain length 3 and 5, where the correlation coefficient is 0.71 ($p<0.02$) and 0.20 respectively. These results show that sub-chain lengths 1 and 3 have the greatest influence on the difficulty of the task. There are only four of the eleven scenarios where the object shapes can be differentiated with a single hBot, which is relative to sub-chain length 1. In these four cases the data-chain of the valid object shapes, ID6 and ID10, have a single 3 in their data-chain, where invalid object shapes do not. All eleven scenarios can be solved with three hBots in the correct place, represented by the difference value at sub-chain length 3. As there is no need to have five hBots cooperating to solve the scenarios there is no significant correlation between the perceived difficulty and the difference value at sub-chain length 5.

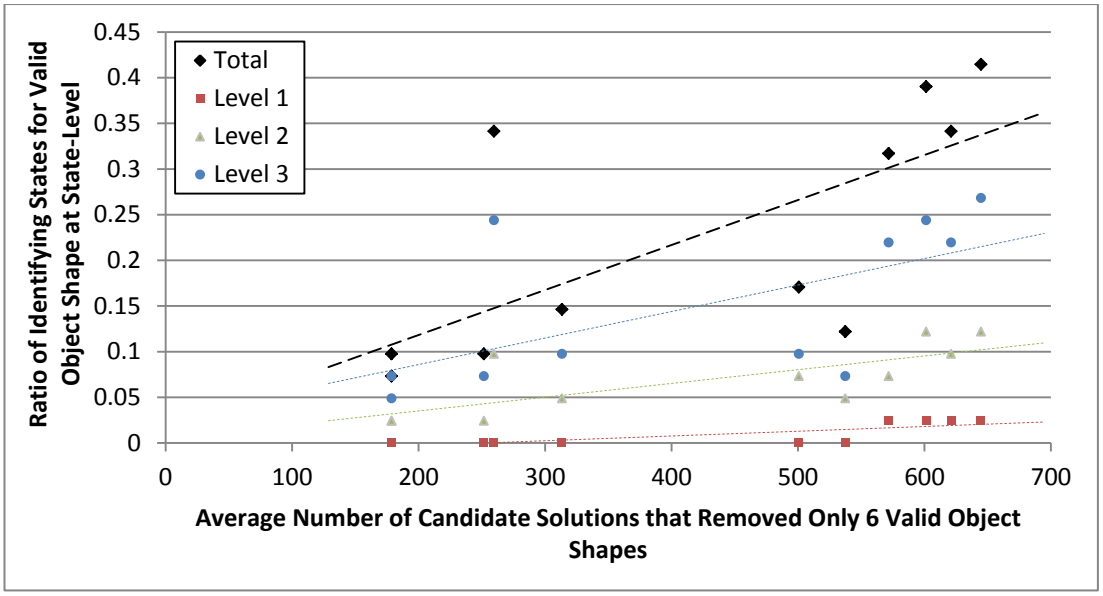


Figure 10.5: The expected difficulty due to the total number of distinguishing states at each state-level against the measured difficulty of the genetic algorithm to solve the relevant scenario.

The number of identifying states for the valid object shape to be found at the different state-levels gave a more significant correlation with the perceived difficulty of finding the solutions to the scenarios. Considering all state-levels at once gave a correlation coefficient equal to 0.69 ($p<.02$), shown in figure 10.5. As the number of identifying states decreases the number of generations required to solve the scenario increases. At state-levels 1, 2 and 3 the correlations are 0.79 ($p<.01$), 0.75 ($p<.01$) and 0.63 ($p<.05$) respectively. These results relate to those of the sub-chain length difference values, where the state-levels represent how many hBots are required for that level to be achieved. state-levels 1 and 2 are more significant than state-level 3, this is due to the decreased probability of reaching state-level 3 as the object shape may already have been removed or due to the interaction required by five hBots.

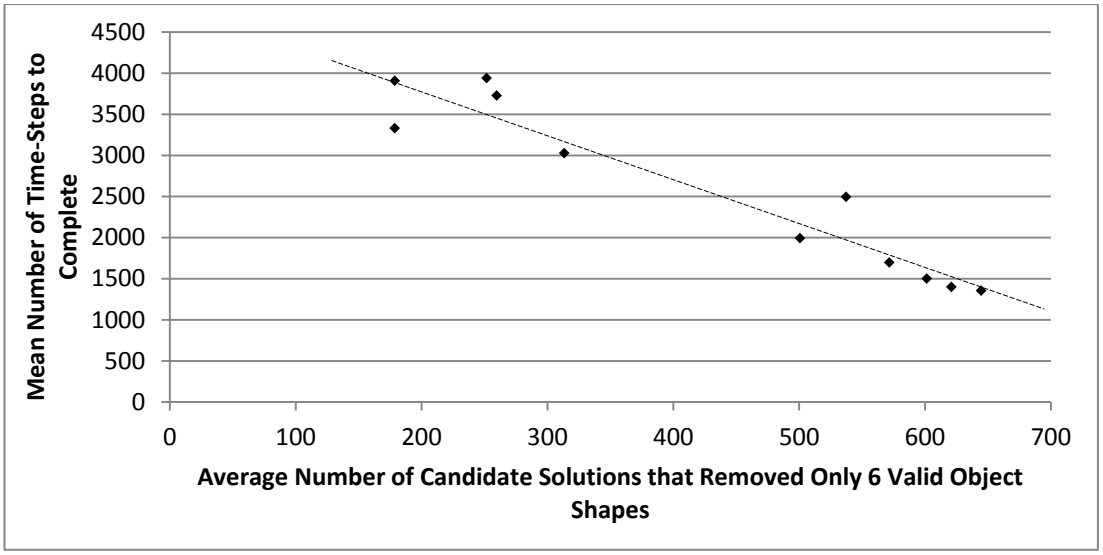


Figure 10.6: The expected difficulty due to average number of time-steps to complete the scenarios in an unbounded test against the measured difficulty of the genetic algorithm to solve the relevant scenario.

The relative difficulty of completing the task scenario given the solution was known has a correlation with the difficulty of finding the solution with a value of -0.96 ($p<.01$), figure 10.6. This correlation is likely because a more difficult task will take longer to complete, therefore there is less time for the hBots to reach the higher state-levels required, through their interactions with each other. If the candidate solution was not ideal the hBots may require more time-steps than allotted in the test to complete the task. This would lead to less distinction between the fitness values of the solutions that could potentially perform well, given more time, and those that are unable to distinguish between the object shapes. As long as suitable solutions can be found within the allotted amount of time-steps this is less of an issue.

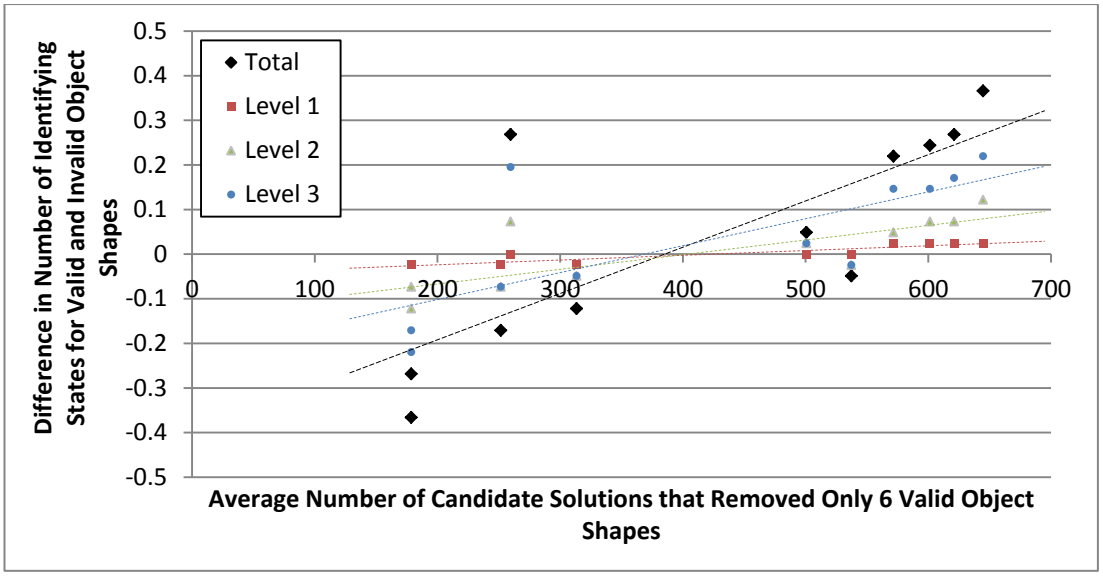


Figure 10.7: The expected difficulty due to the difference in number of distinguishing states at each state-level against the measured difficulty of the genetic algorithm to solve the relevant scenario.

By considering the difference in the number of identifying states for both the valid and invalid object shape a clearer relationship of the difficulty of finding a suitable solution relative to finding a solution that removes the invalid object shapes instead. Figure 10.7 shows the difference between the identifying states of both object shapes for different scenarios at each level against the measured difficulty of finding a suitable solution through a GA.

A significant correlation coefficient of 0.78 ($p<.01$) is found when considering the difference in the number of identifying states over all three state-levels. A similar trend to the previous difficulty predictors is found where the lower level states have more significant correlation coefficients is found for this method too. The coefficients are 0.91 ($p<.01$), 0.78 ($p<.01$) and 0.74 ($p<.01$) for the differences of state-level 1, 2 and 3 respectively. These are the most significant correlation across the measures with the exception of the number of time-steps taken.

A further discussion of all the correlation coefficients for both the GA and the random methods is included in section 10.3.

10.2 The Results for the Random Search Method

For each of the eleven scenarios 900 randomly generated candidate solutions were tested, in the same manner as the GA. The results of the eleven test scenarios are shown as boxplots in figure 10.8, where the blue box shows the first and third quartile and the red line the median, the whiskers indicate the maximum and minimum fitness values for that test scenario. There is little variation between the first, second and third quartile, showing that the majority of the results were similar for each of the scenarios. The same definition of a suitable scenario is used as in the GA method.

Considering all three tests runs, suitable solutions were found for eight of the eleven scenarios. There were three candidate solutions where a suitable solution was not found: F5I6, F5I9 and F8I10.

The candidate solution that has the maximum fitness value for scenario F5I6 removed all of the valid object shapes but also removed a mean of 1.13 invalid object shapes. Similarly, for scenario F5I9 the best performing candidate solution removed all six valid object shapes but removed a mean of 1.67 invalid object shapes. In the case of scenario F8I10 the candidate solution with the highest fitness value, removed a mean of 1.13 valid object shapes whilst removing a mean of 0.4 invalid object shapes. In these three scenarios candidate solutions were found where the opposite task was solved, the valid shape was left untouched whilst all the invalid object shapes were removed.

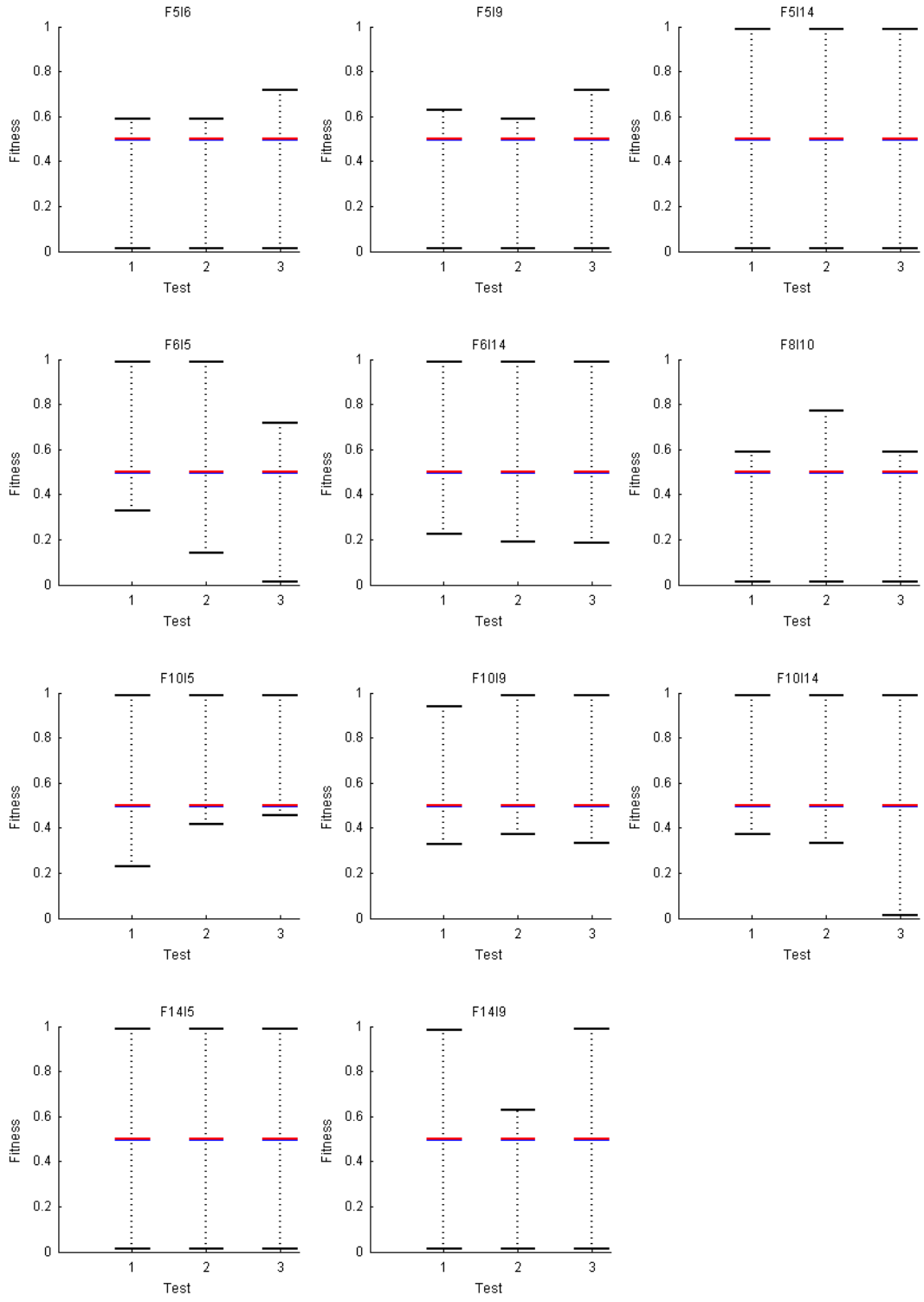


Figure 10.8: Box and whisker plots of the fitness values calculated for each of the 11 scenarios when a random search method was used to find 900 candidate solutions. Each scenario was repeated three times. The maximum and minimum values are shown by the whiskers.

10.2.1 Comparison to Predicated Difficulty Metric of Scenario

To determine the observed difficulty the random method had in finding a suitable solution from the experiments the following method was utilised:

- The average number of successful candidate solutions over the three repeats for each of the eleven scenarios.

The same three predicted difficulty factors were considered as in the GA case. These included:

- The average difference values of the sub-chain lengths, 1, 2 and 3, figure 10.9.
- The total number of identifying states which distinguish the valid object to be found from the object to be ignored divided by the total number of states, figure 10.10.
- The average number of time-steps it took to complete the same scenario when there was no restriction on the amount of time-steps, figure 10.11.
- The difference in the total number of identifying states for both the valid object shape and the invalid object shape, divided by the total number of states, figure 10.12.

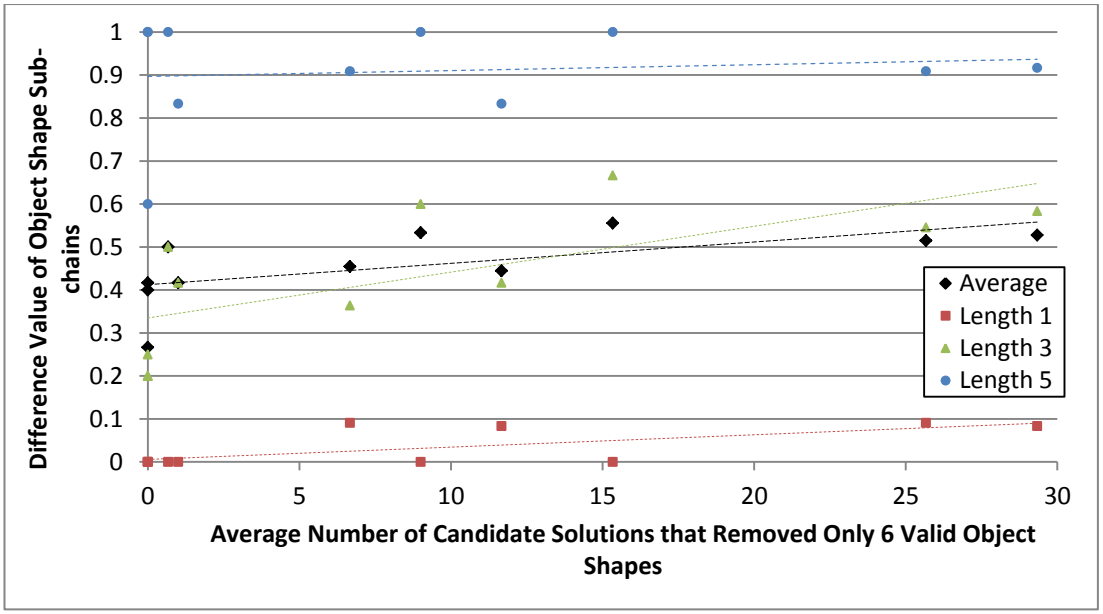


Figure 10.9: The expected difficulty due to the average difference in the sub-chain length against the measured difficulty of the random method to find a solution to the relevant scenario, measured by the average fitness value.

Considering the different lengths of sub-chain and their relative difference values in relation to the average fitness of the random results for the different scenarios showed a general positive correlation, figure 10.9. The results started with significant correlation coefficient of 0.69 ($p<.02$) where the sub-chain length is 1 or 2 and this decreased as the sub-chain length increased to length 5 which had correlation coefficient of 0.12, showing no significant correlation. These results showed that the scenarios that can be completed with low state-levels that had large difference values were relatively easy to solve using the random search method. As the sub-chain lengths get longer, the number of hBots required to react with an appropriate portion of the object shape increases. Therefore, this is less likely to occur, reducing the correlation with the outcome.

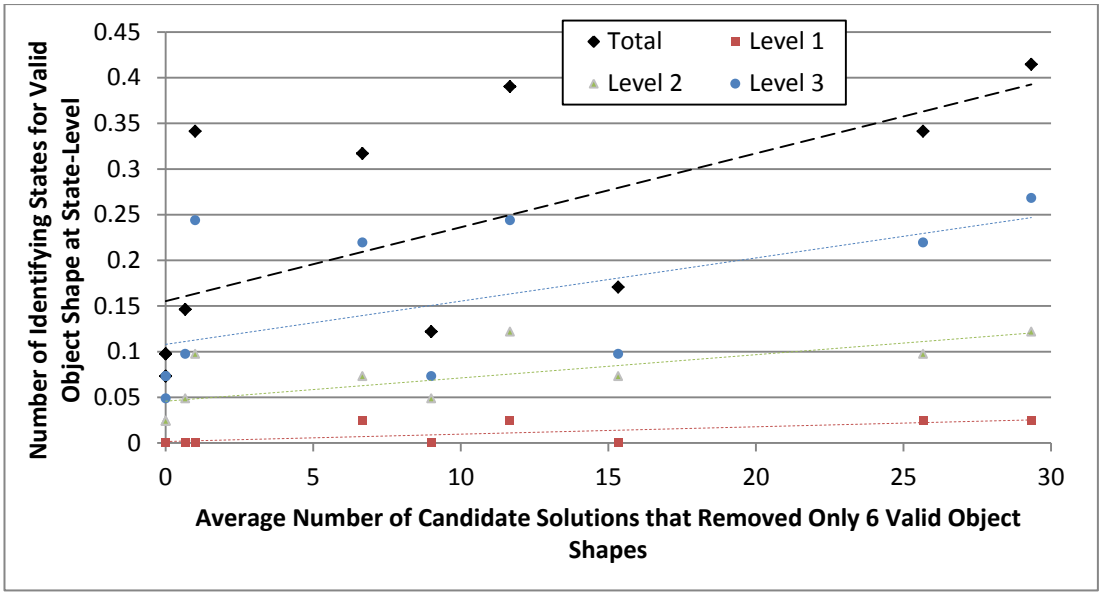


Figure 10.10: The expected difficulty due to the total number of distinguishing states at each state-level against the measured difficulty of the random method to find a solution to the relevant scenario, measured by the average fitness value.

Looking only at the number of identifying states available for potential candidate solutions gave a high positive correlation at low state-levels, 0.69 ($p<.02$), 0.72 ($p<.02$) for state-levels, 1 and 2 respectively and 0.58 ($p<.1$) for state-level 3, figure 10.10. Interestingly, in this measurement there is a marginally stronger correlation for state-level 2 than state-level 1, where in the difference value measurement the results were the same. This outcome is due to the difference in the two measurements. The difference value considers which and how many of the features of the data-chains are different from each other, even if those differences are identical to each other. In the case of the number of identifying states, this considers the number of unique identifying features of the valid object shape. This number of unique features relates to the identifying states, which if there are more of them, it is more likely that they can be found by the random method.

The measurement of difficulty at state-level 3 is equivalent to the difference value at sub-chain length 5, as both require five hBots neighbouring each other. However, where the number of identifying states is concerned there is a more significant correlation where there was none for the difference values. This is due to lack of variation of sub-chain length 5 when compared to the identifying states at state-level 3.

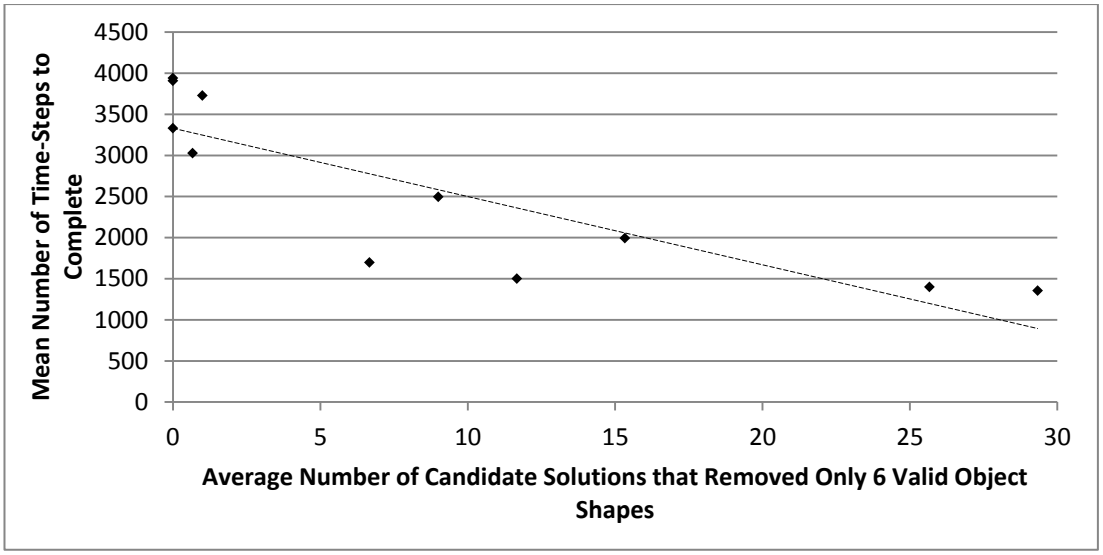


Figure 10.11: The expected difficulty due to average number of time-steps to complete the scenarios in an unbounded test against the measured difficulty of the random method to find a solution to the relevant scenario, measured by the average fitness value

The number of time-steps required to complete the task has a relatively high correlation coefficient with a value of -0.84 ($p<.01$), figure 10.11. The reason are the same as for the GA method, see section 10.1.1. However, this correlation was marginally less significant than the GA relationship with the number of time-steps.

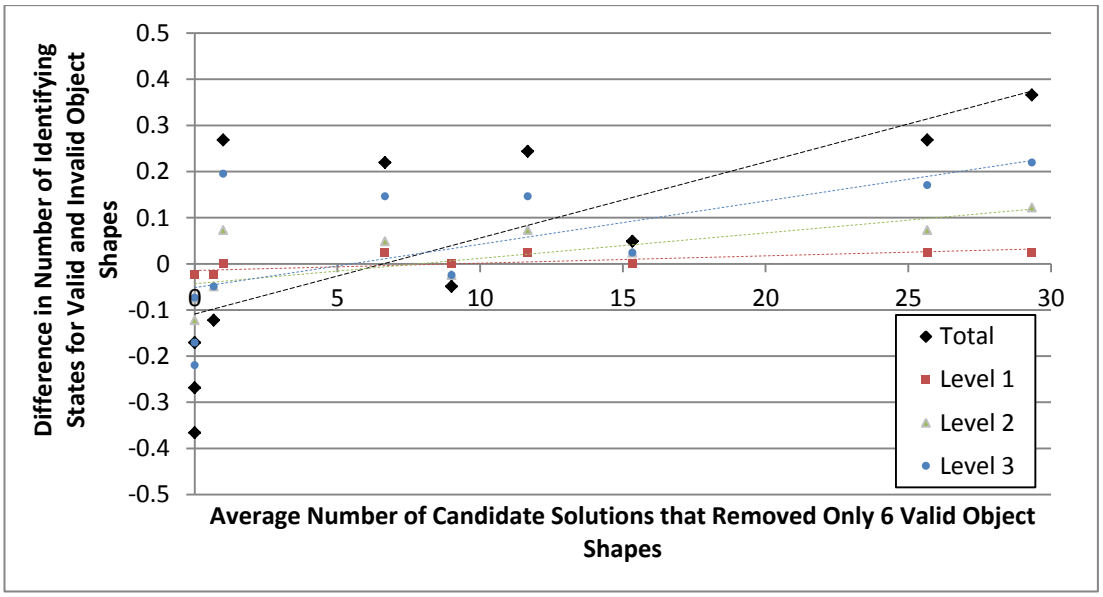


Figure 10.12: The expected difficulty due to the difference in number of distinguishing states at each state-level against the measured difficulty of the random method to find a solution to the relevant scenario.

Figure 10.12 shows the difference in the number of identifying states for the valid and invalid object shapes. The correlation coefficient whilst considering all three state-levels

is 0.70 ($p<.02$). As with the GA method the correlations are more significant for lower state-levels than higher state-levels, 0.77 ($p<.01$), 0.74 ($p<.01$) and 0.65 ($p<.05$) for state-levels 1, 2 and 3 respectively.

10.3 Correlation of Difficulty Measurements to Scenario Difficulty

All the correlation coefficients for all the different measurements of difficulty against the perceived difficulty for both the random and the GA method are collated in table 10.3.

	Random Method	Genetic Algorithm
Sub-chain length 1 difference value	0.688757	0.786602
Sub-chain length 3 difference value	0.68721	0.706011
Sub-chain length 5 difference value	0.118016	0.201136
Average sub-chain length difference value	0.633661	0.704133
Number of identifying states at state-level 1	0.696835	0.79023
Number of identifying states at state-level 2	0.719364	0.749145
Number of identifying states at state-level 3	0.57913	0.627301
Total Number of identifying states at state-levels 1, 2 and 3	0.648998	0.697743
Number of time-steps to complete the task with baseline solution	-0.84215	-0.95553
Difference in number of identifying states at state-level 1	0.767571	0.907736
Difference in number of identifying states at state-level 2	0.740281	0.778752
Difference in number of identifying states at state-level 3	0.651904	0.739804
Total difference in number of identifying states at all state-levels	0.698914	0.777003

Table 10.3: The correlation coefficients for all methods of predicting the difficulty of different scenarios when compared with the measured results from the random method and the genetic algorithm.

In general the correlations for predicting the difficulty of finding the solutions were more significant for the GA method when compared to the random method of determining the solutions. This is perhaps due to the feedback present in a GA. Candidate solutions that have a higher fitness are recombined, producing new solutions sharing their traits. The feedback in the method means that the suitable solutions will produce more suitable solutions. This means that the ease or difficulty of finding a suitable solution is magnified relative to the random method where there is no relationship between the different candidate solutions.

In both methods the most significant measurement of difficulty was the amount of time-taken to complete the task with the given generic solution without time restrictions. Given an ideal solution takes more time-steps to solve for a more difficult task, the

effect for less than ideal solutions should be the same. There is a gradient of fitness between where these less ideal candidate solutions are only part capable of completing the scenario and not capable of completing the scenario in the allotted number of time-steps. The point at which this occurs will relate to the difficulty of the task scenario. Therefore, the amount of time-steps taken for the generic solution provides a clear indication of how difficult it is for all other partial candidate solutions to complete the scenarios. However, of all the measurements this is the most time consuming test to find values for by a significant margin.

After the time-steps the most significant group of measurements was the difference in the number of identifying states, followed by the number of identifying states for the valid object shape and finally the difference values for each of the sub-chain lengths. The reason being is that the difference in the number of identifying states not only considers how easy it is to find a suitable solution but how easy it would be to find the opposite, where the invalid object shapes are removed. It is by considering these two opposing views at the same time that a clearer difficulty measurement of the scenarios is found. The difference values which were found in section 8.3.2, do not give the most ideal solution when the consideration of finding the solution is added to situation. The difficulty values of the scenarios do not consider the many possible suitable solutions of completing the scenario there are or the difficulty of finding those solutions, but simply the task of removing the object shapes that the hBots perform.

Over these three group measurement types which considered the interaction of one, three and five hBots, either through state-level or sub-chain length, there was more significant coefficients when this was one, decreasing through three and five hBots. The reason for this is both because it is less likely that more hBots will be interacting with each other around the object shape with the current amount of hBots and that if the hBots correctly identify the valid object shape at lower states they will remove it meaning there is less need for the higher state-level states.

10.4 Comparison of Fittest Solutions

To determine the overall capability of the cooperative object recognition GA it was compared to the solutions with the highest fitness from both the randomly derived solutions and those from the generic solutions for each scenario. Figure 10.13 shows the maximum fitness values of the two different solution deriving methods and the generic solution, and figure 10.14 shows a more detailed view of the upper fitness values. The average improvement of the GA over the random method and the generic method were approximately 0.001 and 0.003 respectively where a suitable solution was found. Despite what at first may appear like a small difference it was in fact a significant improvement in efficiency. Given that at this level of fitness, excluding the cases where the random method failed to find a solution, only the six correct object shape types are removed. Therefore, the comparison between the scores is based solely on the amount time-steps it took to remove the first and last correct object shape. These two factors have a total effect of up to 7000 time-steps, which translates to approximately 0.010 of the fitness value.

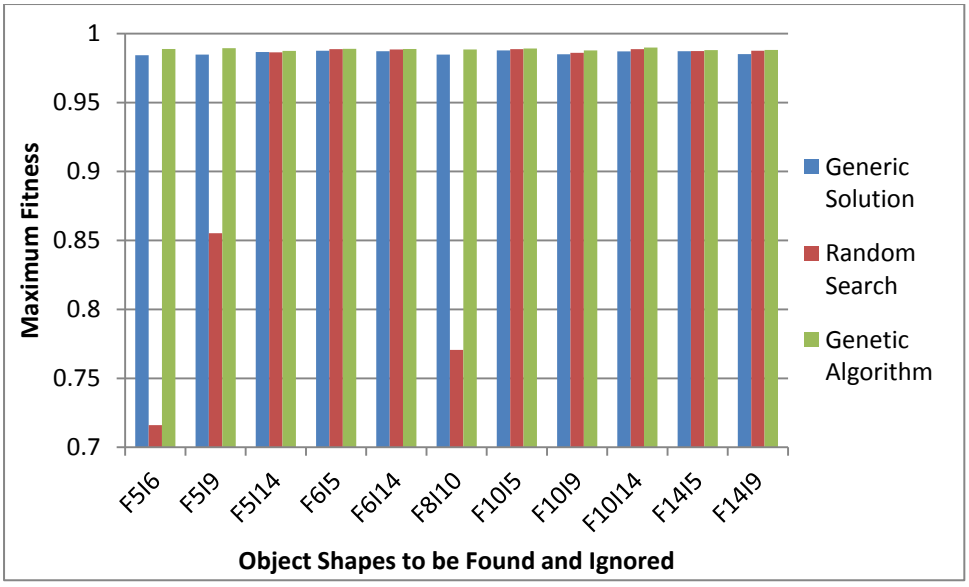


Figure 10.13: The maximum fitness of the random and GA method of finding solutions compared to the generic method for each of the eleven scenarios.

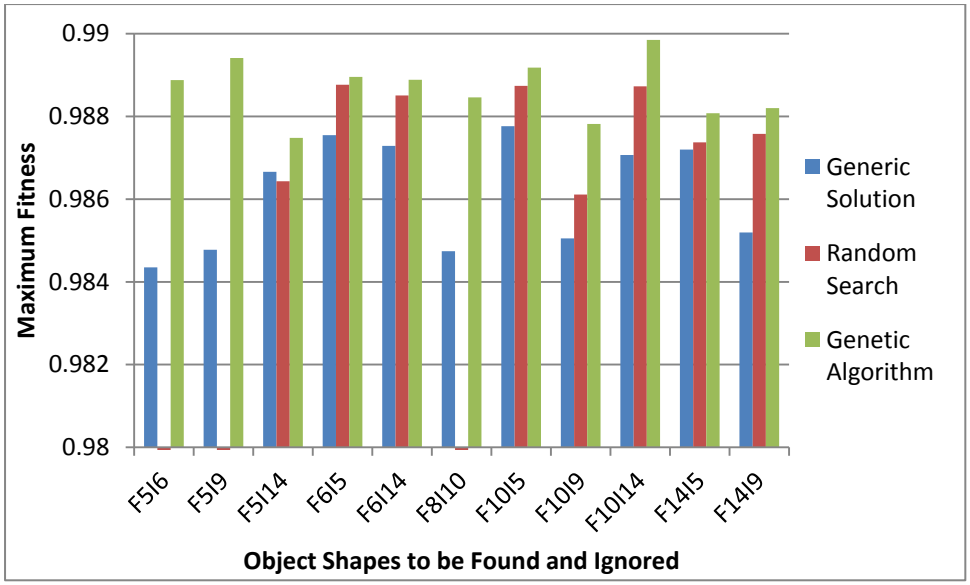


Figure 10.14. A detailed view of the upper fitness values of the random and GA method of finding solutions compared to the generic method for each of the eleven scenarios.

An interesting outcome was that the GA method found better solutions for all the different scenarios than both the generic solutions and the random solutions. In seven of the eleven scenarios the random solution had a higher fitness value than the generic solution fitness value. This improvement showed the capability of the two systems to find solutions that are more tailored to the specific scenarios than the general method derived to form the generic solutions to the scenarios.

State	1	4	22	25	26	27	5	31	32	33	36	37	40	6	52	53	57	7	70	71	8	19	2	10	103	104	105	11	112	113	117	12	133	13	151	3	16	184	185	17	193
Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40
f5	A						A				A							A	A				A	A		A																
i6	A	A			A		A		A		A	A	A	A	A		A						A	A				A	A							A	A	A			
G	3	2	2	2	2	2	3	2	2	2	2	2	2	2	2	2	2	1	1	2	2	2	3	3	2	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
R	2	3	1	1	1	1	3	2	3	3	2	3	3	2	3	1	2	1	3	3	1	2	3	3	3	2	3	3	2	3	1	2	2	1	3	2	2	2	3	1	2
GA	2	3	2	1	1	3	2	1	2	2	3	3	3	2	3	1	3	1	1	1	1	2	3	3	3	1	2	2	3	3	1	1	2	3	2	2	2	2	3	1	2
f5	A						A				A							A	A				A	A		A															
i9	A	A			A		A	A						A	A						A	A	A	A			A								A	A		A			
G	3	2	2	2	2	2	3	2	2	2	1	2	2	2	2	2	2	1	1	2	2	2	3	3	2	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
R	2	2	3	3	3	1	2	1	1	2	2	1	2	3	3	1	2	1	2	2	2	3	3	2	1	3	2	1	1	1	1	1	1	1	1	3	3	2	3	2	1
GA	2	2	3	3	3	1	3	3	1	1	1	1	1	2	1	1	3	1	1	1	2	3	3	3	3	1	2	1	3	3	1	3	3	3	3	2	2	1	2	2	3
f5	A						A				A							A	A				A	A		A															
i14	A	A		A			A		A														A				A			A		A				A	A				
G	3	2	2	2	2	2	3	2	2	2	1	2	2	2	2	2	2	1	1	2	2	2	3	1	2	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
R	3	2	2	2	3	3	3	3	3	3	3	3	3	2	3	3	1	1	1	1	1	2	3	1	2	2	2	2	3	2	1	1	1	2	3	2	2	2	1	3	3
GA	2	2	3	2	2	1	2	1	2	3	1	1	1	1	1	3	2	1	3	3	1	2	3	1	2	1	2	1	2	3	2	3	2	3	2	3	2	3	2	3	1
f6	A	A			A		A		A		A	A	A	A	A		A						A	A				A	A							A	A	A			
i5	A						A				A							A	A				A	A		A															
G	3	1	2	2	1	2	3	2	1	2	3	1	1	1	1	2	1	2	2	2	2	2	3	3	2	2	2	1	1	2	2	2	2	2	2	1	1	1	2	2	2
R	2	1	2	1	3	3	2	2	2	1	3	1	1	3	1	3	1	2	2	3	2	1	2	2	3	3	1	1	1	2	2	3	3	2	3	1	1	1	2	2	2
GA	2	1	3	2	1	2	2	3	3	2	2	2	2	2	2	1	1	3	3	3	3	1	2	2	2	3	1	2	1	3	2	1	3	2	1	3	2	1	1	1	2
f6	A	A			A		A		A		A	A	A	A	A		A						A	A				A	A							A	A	A			
i14	A	A		A			A		A														A				A			A		A				A	A				
G	3	3	2	2	1	2	3	2	3	2	1	1	1	1	1	2	1	2	2	2	2	2	3	1	2	2	2	3	1	2	2	2	2	2	2	1	1	1	2	2	2
R	2	2	3	2	1	3	3	2	2	3	2	1	3	2	3	1	2	1	3	3	1	2	2	2	2	1	2	3	2	3	1	2	1	3	2	1	2	2	2	3	3
GA	2	3	2	3	3	3	3	1	2	1	3	2	2	1	1	1	1	1	2	1	3	1	2	1	3	3	1	2	3	2	3	1	1	2	2	1	1	1	3	2	3
f8	A	A				A								A	A																					A	A	A			
i10	A	A		A	A		A	A	A	A				A		A		A		A			A	A		A		A	A		A	A	A			A				A	A
G	3	3	2	2	2	1	2	2	2	2	2	2	2	3	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	3	1	1	2	2	2
R	2	2	2	3	2	2	3	1	3	3	2	1	1	2	2	2	2	3	3	3	1	2	2	2	2	3	2	2	2	1	1	2	3	1	2	3	3	1	2	3	2
GA	2	2	3	2	2	3	2	2	3	3	1	3	1	2	2	3	3	2	1	2	1	2	2	2	3	1	3	2	3	1	2	3	2	3	1	3	1	1	1	2	3
f10	A	A		A	A		A	A	A	A				A		A		A		A			A	A		A		A			A	A	A			A				A	A
i5	A						A				A							A	A				A	A		A															
G	3	1	2	1	1	2	3	1	1	1	2	2	2	1	2	1	2	3	2	1	2	2	3	3	2	3	2	1	1	2	1	1	1	2	2	1	2	2	2	1	1
R	2	2	1	3	2	1	2	3	2	1	2	1	2	2	1	1	2	2	3	3	2	1	2	2	1	3	2	1	1	3	2	1	2	2	2	1	2	1	3	2	1
GA	2	2	3	2	2	2	2	1	1	1	3	1	1	3	1	1	2	2	1	2	3	3	2	2	1	3	2	1	3	3	1	1	1	2	3	1	3	1	2	3	3
f10	A	A		A	A		A	A	A	A				A		A		A		A			A	A		A		A			A	A	A			A				A	A
i9	A	A			A		A	A						A	A						A	A	A	A			A								A	A		A			
G	3	3	2	1	3	2	3	3	1	1	2	2	2	3	2	1	2	1	2	1	2	2	3	3	2	1	2	1	2	2	1	1	1	2	2	3	2	2	2	1	1
R	2	2	3	3	2	1	2	2	2	3	1	2	3	2	2	2	3	3	1	3	3	2	3	2	2	3	3	1	2	2	1	2	2	2	3	2	3	3	3	2	3
GA	2	3	1	3	2	1	3	3	1	2	3	1	2	2	2	3	2	1	1	2	2	2	3	2	1	3	3	1	3	1	3	1	2	3	1	2	2	1	3	1	1
f10	A	A		A	A		A	A	A	A				A		A		A		A			A	A		A		A			A	A	A			A				A	A
i14	A	A		A			A		A														A				A		A						A	A				
G	3	3	2	3	1	2	3	1	3	1	2	2	2	1	2	1	2	1	2	1	2	2	3	1	2	1	2	3	1	2	1	1	1	2	2	1	2	2	2	1	1
R	2	3	3	2	2	1	2	2	3	3	1	3	3	2	2	1	2	2	2	3	3	1	2	3	2	3	2	2	2	3	3	3	3	3	2	1	1	1	2	1	1
GA	2	3	2	3	2	2	2	1	2	1	3	2	1	1	1	3	3	1	2	1	2	1	2	3	2	2	2	2	3	2	1	1	1	2	1	1	1	3	2	2	1
f14	A	A		A			A		A														A				A		A						A	A				
i5	A						A				A							A	A				A	A		A															
G	3	1	2	1	2	2	3	2	1	2	2	2	2	2	2	2	2	2	2	2	2	2	3	2	2	2	2	1	2	1	2	2	2	1	1	2	2	2	2	2	2
R	3	1	3	2	2	3	2	1	1	2	3	3	1	3	3	1	2	2	2	1	2	1	2	2	3	3	1	1	2	2	2	1	2	1	1	1	1	2	1	1	1
GA	2	2	3	1	3	3	2	2	2	3	2	3	1	3	3	2	3	2	2	3	3	3	3	3	3	2	1	1	1	3	1	3	3	1	2	1	2	2	2	1	2
f14	A	A		A			A		A														A				A		A						A	A				
i9	A	A			A		A	A						A	A						A	A	A	A			A								A	A		A			
G	3	3	2	1	2	2	3	2	1	2	2	2	2	2	2	2	2	2	2	2	2	2	3	2	2	2	2	1		1	2	2	2	1	1	2	2	2	2	2	2
R	2	3	2	2	2	1	3	3	1	2	3	3	2	2	2	1	1	3	1	3	3	3	3	2	2	1	2	1	2	1	2	2	2	1	3	3	3	3	3	1	1
GA	2	2	1	1	2	2	3	3	3	1	1	1	3	3	2	2	3	2	3	3	3	2	3	3	3	2	3	1	1	2	2	2	1	1	2	2	2	2	3	3	1

Table 10.4: Fittest candidate solution genomes for the three methods for the eleven scenarios. The achievable states are noted with an 'A'. Each instance where a state-behaviour 1 could remove the wrong object shape is highlighted.

Through looking closely at the solutions derived by the different methods, table 10.4, an understanding of what made the GA method perform better than both the random method and the baseline generic method could be gained. The most important behaviour to consider is behaviour 1, this is the behaviour that determined that an object shape should be removed. If this behaviour was applied to the wrong hBot state then the wrong object shapes could be removed and the incorrect object shapes could be left alone. This behaviour had the largest effect on the fitness values achieved by the different potential solutions. However, just looking at this behaviour alone did not give a clear indication as to why the GA achieved a higher fitness score than the generically derived solution. From table 10.4 it can be seen that not only did the GA not find all the instances where it could have had a behaviour 1 (i.e. where the object shape to be found has a potential state that the object shape to be ignored does not) it has them in the opposite case which could allow the incorrect object shapes to be removed if given enough time. The frequency that this occurs is discussed in section 10.4.2.

The reason that these, errors in state-behaviours, did not affect the overall fitness of these high performing solutions was due to their high state-levels. In all the cases where the hBots could potentially remove the wrong shape the state-level of the relative behaviour was always level 3, the most unlikely level to be reached. Also there is further complexity, as to reach this state-level there was another issue, what are the likely hoods of the hBots staying in place to reach those results. It was this point where behaviours 2 and 3 come into play, where they represent a low and high probability of moving away from an object shape respectively.

Taking for example the solution with the highest fitness from the GA method for the scenario F8I10. In this instance there was state-rule-behaviour which states remove the object, where it would remove the object shape that should be ignored. This behaviour was found in this instance at state 104. In order for a hBot to reach state 104 it had to be in state 10 and have neighbours in states 5 and 7. Of these three states the hBots need to reach, in this specific solution all three of them have behaviour 2, which is stay in place with a low probability. These three states, 10, 5 and 2 all require three hBots in states 1 and 2 to be reached. Both states 1 and 2 have the stay in place with a low probability behaviours. Therefore, there is a lower likelihood of the needed number of hBots staying in the same place to reach this state-level 3 state which would perform the wrong action, compared to these being state-behaviour 3.

10.4.1 Testing the Fittest Solutions

For each of the eleven scenarios the best performing candidate solutions both the random method and the GA method were allowed to run for an extended amount of time-steps. The environment is identical to that used in section 8.4, however the maximum number of time-steps allowed for the test is increased to 50000. This increase allows the swarm to potentially act on states which may remove the incorrect object shape if reached.

		Fraction of attempts where a specific number of invalid object shapes were removed.					
Number of Invalid Removed		**1**	**2**	**3**	**4**	**5**	**6**
F5I6	**GA**	1.00	0.96	0.88	0.80	0.58	0.40
	Random	1.00	1.00	1.00	1.00	0.98	0.90
F5I9	**GA**	1.00	0.92	0.80	0.68	0.50	0.24
	Random	1.00	1.00	1.00	1.00	1.00	1.00
F8I10	**GA**	0.94	0.74	0.50	0.24	0.12	0.02
	Random	0.98	0.96	0.90	0.78	0.60	0.40
F10I5	**GA**	1.00	1.00	1.00	0.98	0.94	0.54
F10I14	**GA**	1.00	0.96	0.90	0.78	0.40	0.16

Table 10.5: In solutions where an error is present in the genome, the fraction of the 50 tests that removed a specific number of the six invalid object shapes for the relative scenarios.

The fraction of invalid object shapes removed for each of the scenarios where there was a potential error in the genome is presented in table 10.5. From this it is shown that in situations where the hBots are allowed to run for a longer time they have opportunity to act on errors in their state-behaviours and remove invalid object shapes. With the exceptions of F10I5 and F10I14, where the random method does not remove any invalid object shapes, the fittest GA solution removes incorrect object shapes less often than the fittest random solution.

Table 10.6 shows the ratio of time-steps between the last valid object shape removed and the first invalid object shape removal. This gives an indication of how long it takes the hBot swarm to start removing the incorrect object shape after it has accomplished the task of removing the valid object shapes. In each instance where both methods have errors in their genome, the GA outperforms the random solutions, taking longer to remove the invalid object shapes relative the valid object shapes.

	Random	GA
F5I6	2.77	4.62
F5I9	1.62	5.93
F8I10	0.52	7.03
F10I5	-	7.67
F10I14	-	11.65

Table 10.6: The ratio of time-steps taken to remove the first invalid object shape relative to the last valid object shape.

This analyses shows that despite the higher state-levels of the states that have this errors and the state-relationships that lead to them, if given enough time they are acted upon. Showing that these are not as optimal a solution as hoped.

10.4.2 Assessing the Suitable Solutions

Considering all of the suitable solutions found from each method shows a potential solution to this problem. Figure 10.15 and figure 10.16 show the number of suitable solutions out of the 900 candidate solutions and the number of those that had an error in them, for both the GA method and random method respectively.

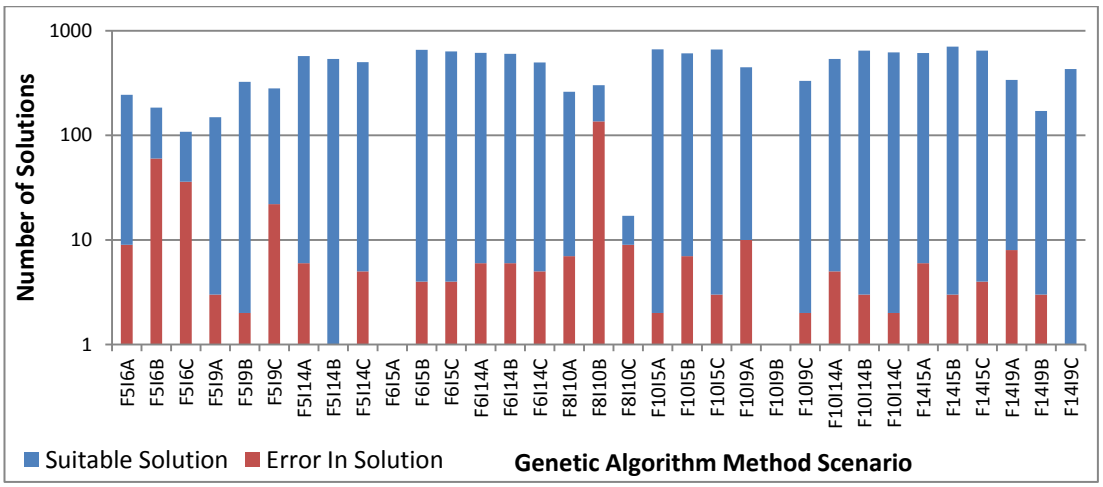


Figure 10.15: The number of successful solutions and the number of those with errors in their genomes, for each scenario test out of the 900 candidate solutions for the GA method.

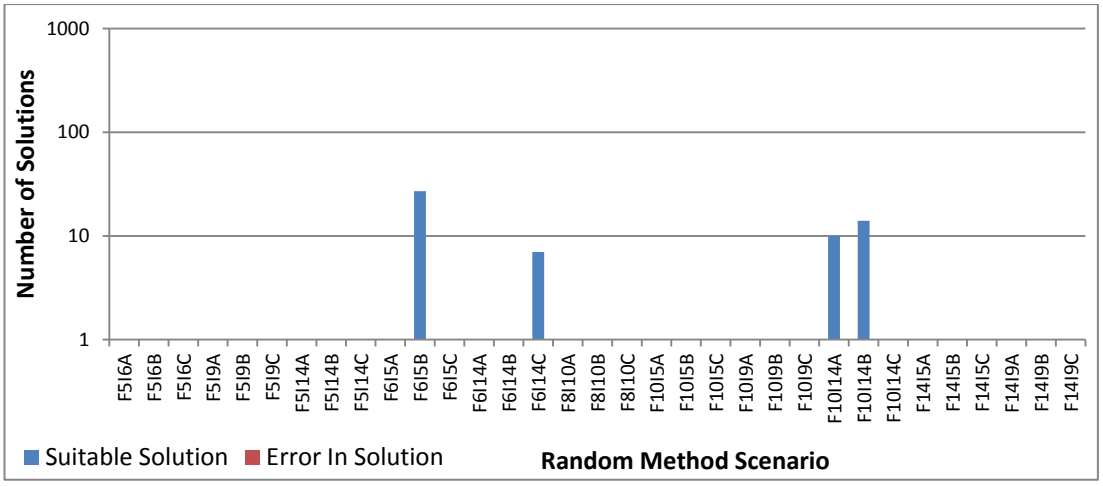


Figure 10.16: The number of successful solutions and the number of those with errors in their genomes, for each scenario test out of the 900 candidate solutions for the Random method.

Out of all the suitable solutions found through the GA method, the majority of them do not have errors in their genome which would cause them to incorrectly remove the invalid object shape. This suggests that variations could be implemented on the GA that would account for this behaviour. For example this could be achieved allowing the hBots more time-steps in later generations. Alternatively, increasing the amount of hBots in the system would increase the interaction of the hBots around the object shapes making these higher level state-behaviours more likely to occur.

10.5 Summary

Across all eleven of the scenarios the GA method found solutions with higher fitness values than both the generic method and the random method. More suitable solutions, which removed only the six valid object shapes in the 7000 time-steps, were found when using the GA rather than the random search method. A number of the fittest solutions for both methods had potential errors in them where the genome would have the hBots remove the invalid object shapes if they reached certain states. These states were commonly at state-level 3 and had related lower level states with behaviours that were more likely to have the hBot move from that specific cell. It was found through further testing that despite the lower probability the behaviours would be acted on if given enough run time. The number of time-steps between the last removed valid object shape and the first invalid object shape was greater for the GA than the random search method. This problem could be reduced in the GA method by having longer runs in later generations or using more hBots for training increasing the interaction with the object shapes. This will need to be considered in future experimentation.

Of the metrics that determine the difficulty of finding a suitable solution, the number of time-steps it takes to solve has the highest significance, followed by the difference in the number of identifying states between the valid and invalid object shape. This is the most suitable measure of difficulty as it can be found without having to run simulations.

Overall the GA method outperformed the random search method in determining suitable behaviours for the hBots at different states, although there is further research required in tailoring the specifics of the GA.

Chapter 11: Discussion

From the investigation it was found that a genetic algorithm was capable of generating suitable solutions for the state-behaviours of hBots such that they could solve a range of cooperative object recognition task scenarios. Furthermore, metrics of the scenarios' difficulty were found that will allow future experiments, using similar methods, to determine the relative difficulty for agents to solve a specific cooperative recognition task scenario. This chapter identifies and discusses the strengths and weaknesses of the research undertaken and the implication that the findings have on cooperative object recognition research.

11.1 Task and Application

A task was envisioned which involves distinguishing one type of object shape from another and then removing that object shape. The distinction between object shapes was made by their size and shape as there were no other marks or unique characteristics. Removing an object shape was achieved by either pushing it from the search space or destroying it completely, depending on the specific test.

This type of cooperative object recognition had previously being completed with multi-agents systems. It was hoped that by initiating the development in swarm robotic techniques for cooperative object recognition the task could be undertaken by individual simpler agents who have no need for complex sensors for viewing the outside world, eliminating the processes of stitching multiple images together in two or three-dimensional spaces or, depending on the application, have no need for motors or actuators to move them, as they could move randomly through Brownian motion. These simplifications in the individual agents would, in addition, allow them to be produced smaller and smaller as technical advances continue.

The limitation of using this method for cooperative object recognition is that it requires the agents to physically interact with the boundary of the object shape which depending on the medium the agents exist in, could increase the complexity of the real world robots when produced. However, it is believed that the potential benefits outweigh these issues especially if a suitable application is found to implement the strategies in. In the experiments for the research for this thesis the agents move through a random walk

which is akin to Brownian motion which suggests applications in fluids to be the most ideal.

The capability to reduce the size of the agents due to their simplicity could open potential medical applications. A system is envisioned where a swarm of cooperative agents could be injected into a human body to find potential threats to the body and either remove them directly or mark them for removal. The swarm would travel around the body interacting with the contours of different entities, possibly viruses, germs, bacteria, cancers or tumours and distinguishing them through their shape. To do this the swarm would initially have to be trained through a GA or other method to identify that type of object shape. This could be done without a shape analysis of the entity as long as a suitable training environment was given to the agents with sufficient feedback. The behaviours found could be replicated from the small scale test group for any scale swarm. The major limitation to these medical applications is the relative size of the items to be identified and the size of the robotic agents.

A less distant application for which a cooperative object recognition swarms could be used for includes cooperative construction tasks. In these scenarios the swarm could analyse the current state of the constructed item or the assembled system and determine the next action to take. This would give the swarm larger information about the structure whilst still only using local communication.

11.2 A Suitable Task for a Swarm Robotic Approach

Swarm robotic tasks often mimic the behaviours of social insect groups directly. In this research on cooperative object recognition it was not the direct mimicking of a specific social insects behaviour that inspired the method only the general idea of social intelligence and swarm behaviours that give guidance. Taking indirect inspiration opens up new areas for potential uses for those core behaviours found at the risk of not having a system to mimic directly to compare to.

Three noted attributes of tasks suitable for swarm robotics are those that cover a region, change scale and require redundancy in the swarm. As the object shapes are distributed in space they cover a region. The identified shapes are removed or destroyed changing the scale of the problem to solve. As the number of object shapes in the search space is reduced some of the agents of the swarm become redundant.

These points in addition to the analyses of the current cooperative object recognition techniques showed that there was an opportunity for a novel swarm robotics approach to the problem.

11.3 The Outcomes

11.3.1 The State-Behaviours of hBot Swarms

Through this research it was found that it was possible to complete a varied range of cooperative object recognition tasks with a swarm of agents that were, homogeneous, anonymous, had limited recall, had no common sense of direction and had a sensor range that was considerably smaller than the area they inhabited. In order to complete this task a state-behaviour system was employed. The agents would change their states based on what they perceived directly next to them. These changing states caused a feedback loop of agents perceiving each other and changing their states accordingly. As the agents states were also affected by a small portion of the object shapes they were in contact with, this would allow them to eventually determine what shape type they were next to.

The number of states used was limited to 264 states at three state-levels, this allowed any hBot to potentially gather the same amount of information as five hBots at their first state-level. This was enough information for the hBots to be able to distinguish all of the object shapes used in the experiments. The number of object shapes used was initially reduced to a set of four celled object shape pairs. However, from the investigation it was shown that these scenarios had a range of difficulty to complete both in terms of solving their state-behaviours and completing the task once the state-behaviours are known.

Despite the capability of these strategies, these are early endeavours in cooperative object recognition with swarm robotic techniques. This observation is made apparent by the system that was used to initially implement the experiments, the Simplified Hexagonal Model. The potential issue with using a grid based system rather than a physical platform or a more realistic simulation technique was that the methods derived cannot be transferred directly. The hBots also have perfect communication with each other as there is currently no noise in the system.

Reverting to a simplistic model allowed the focus of the research to be on the strategies used rather than the designing and implementation of hardware or complex physical simulations. The intentions of the research are to place the ground work leading towards the implementation of the strategies on a physical platform where modular robotic systems are used. The benefits the state-behaviour method provides are ones that will allow much simpler and therefore smaller agents, with limited capabilities, which have been shown in a simulation to be capable of distinguishing between object shapes.

11.3.2 The Training of hBots

In considering the applications of a swarm that could cooperatively identify objects through their shape a more interesting aspect than creating a state-behavioural rule-set from the known geometrics of the object shapes would be allowing the hBots to derive their own state-behaviours through a GA. The basic conduct of the hBots remained unaffected by the GA, they moved and changed between states as previously. However, which one of the three behaviours that they exhibit at those states would be determined by the GA, where the behaviours were; move with low probability, move with high probability and remove object shape. Eleven scenarios were selected to give a range of perceived difficulties from differences in the object shape, discussed further in section 11.3.3.

Running a GA is a computationally heavy task to derive solutions. To determine if the GA was beneficial the best performing solution of the GA for each scenario was compared to the best performing solutions from both a group of 900 random solutions and the previously generated generic solutions. In all scenarios the GA out-performed both other methods. This improvement, although minor in appearance could, depending on the size of the task given to the swarm, make a large difference to the overall efficiency. One interesting anomaly arose. The behaviours rules that the GA had determined would on occasion include the remove behaviour for states that are common to the invalid object shape, i.e. the wrong object shape. These errors in the state-behaviours of the hBots could occur as the number of time-steps were relatively low in the GA, and due to them being behaviours at the highest state-level they would not be acted upon during the testing phase of the GA.

A further investigation was carried out using the fittest members of both the GA and random method in order to determine the effect of the errors in the state-behaviours. It

was found that given enough time, despite the lower probability of them occurring, the behaviours would be acted upon. In the case of the GA solutions where similar random solutions were available the difference between the last valid and first invalid being removed was greater, showing that the GA solutions performed better. For the GA solutions there are also a larger amount of solutions which were deemed suitable, where a suitable solution is one that removes all six valid object shapes leaving the six invalid object shapes during the GA testing. Upon analyses a larger portion of these suitable solutions did not have errors in their state-behaviours that would allow them to ever remove invalid object shapes. Extending the test duration once a certain percentage of solutions found are deemed suitable would reduce this problem in future GA training and would improve the agents state-behaviours and minimise the effect on the overall run time. Another option is to run two parallel sets of tests in order to determine the fitness value, one with a high amount of agents to increase interaction, the other with the standard amount to increase efficiency with the correct distribution of agents relative to arena space.

Overall the results and analyses show that the most optimal GA was not used. It was shown, however, that the GA method outperformed both of the other methods. Therefore, improving the GA would only further improve the efficiency of the GA relative to the other methods. Other methods could be used to train the hBots to distinguish between to object shapes but these need considering in further research.

Additionally, any further study with the GA will have to consider the increasing length of genome with the increase of object shapes and the effect that this will have on the efficiency of the training method.

Overall, a training method would provide a way to have the hBots learn the difference between object shapes without those shapes having to be analysed and compared beforehand. The GA results show that it is possible to derive the state-behaviours required. This will in the future allow far simpler calibration of the agents to any specific object recognition task.

11.3.3 The Metric of Scenario Difficulty

The initial experiments that were carried out only used two types of object shape. For the investigation to be an insightful examination of the hBots capability a study into the possible object shapes that could be produced from hexagonal cells needed to be

carried out. This allowed a range of different scenarios of cooperative object recognition to be produced and a metric of their difficulty to be defined.

Object shapes were created for each combination of different number of hexagonal object cells up to and including nine object cells. Using this method 16673 different object shapes were found. Each object shape was described without considering placement or orientation by using their data-chain. A data-chain was formed by observing all of the empty cells around the object shape, and counting how many object cells they were in contact with, then going round the shape in a clockwise order recording all these values. It is the data-chain of an object shape that allows its difference value from another object shape to be calculated, by considering different length sub-chains.

It was found that in some circumstances two different object shapes could be described by the same data-chain. These involved object shapes that had concaved sections with a single cell thickness. This is due to the way the data-chains are made and could potentially be resolved with a different system. However, the system was implemented as it reflects the way which the hBots see the boundary of the object shapes and interact with each other, and other systems would not. As the limitation is only apparent in cases where there is a tunnel in the object shape of equal width to the hBots this is a limitation at the cusp of the physical dimensions of the hBots. In any real world system there would be a limitation of the fidelity that the agents could respond to. This circumstance of the object shapes having a common data-chain is an example of this limitation seen in the simulation.

As the hBots do not consider the position of their neighbours and only their state, for each sub-chain length the states the hBots can reach is determined to find the difference value. However, this choice in the state relationship rules means that object shapes that are identical when one of them is rotated on the x or y axis, as if mirrored, appear the same to the hBots. These shapes can be noted by their data-chains being the reverse of each other. This limitation in ability was deemed acceptable for the research, as it would also mean that any hBot that was rotated in the same way would still function successfully.

By comparing two object shapes' data-chain, where one is classed as valid and the other invalid, a difference value can be determined which varies with the length of sub-chain

considered. The more the object shapes had in common with each other, the lower their difference value, and the harder they would be for the swarm to distinguish from each other. It was found that the difference value of shape A from shape B is not the same as shape B from shape A. This is because the boundary features that shape A has that makes it distinguishable from shape B are different to those which make shape B distinguishable from shape A.

The difference values calculated showed a correlation with the amount of time-steps it took a hBot group to complete a cooperative object recognition task. Therefore a metric of the difficulty of completing a scenario can be determined by the two object shapes that are being distinguished from one another. The results showed that there were a range of difficulties in completing different task scenarios when a working state-behavioural rule-set had been given to the hBots. The difficulty ranged, where object shapes with up to four cells are considered, from pairs of object shapes that can be differentiated with a single hBot and those that could not even be differentiated with five hBots cooperating.

The training scenarios required a different metric of difficulty, one that would not depend solely on how hard it would be to complete the task of removing the correct object shapes, but on how difficult it would be to find suitable solutions' state-behaviours. Of the four measurements the difference in the number of identifying states for both the valid and invalid object shape provided the best overall correlation coefficient with the measured difficulties from the averaged GA results. This was because it gave an indication of how easy it was to find a solution whilst avoiding solutions that removed the invalid object shapes as well. A high coefficient was also found for the number of time-steps taken to complete a preliminary task, however this is a less efficient way of determining the difficulty as it requires running the initial tests with a pre-determined solution, multiple times.

The major limitations to the investigation into the comparison of object shapes for the training methods was the reduction of which object shapes could be used. Reducing the object shapes so that they only contained ones, twos and threes in their data-chain removed a large aspect of potential shapes. This could be amended in future research by reassessing the hBots state relationships, although it would increase the number of states by a considerable factor. This increase would have caused a time delay that was too long for this research increasing the cost of completing the GA.

Given this is an initial investigation, the range of pairs of object shapes chosen was deemed acceptable, as they were shown to have a range of difficulties indicated by the different metrics available. The metrics identified here will allow future research to identify specific object shape pairs which have a broader range of difficulty when considering both training and task scenarios.

11.4 Future Work

Further work is required for the Simplified Hexagonal Model. More complex shapes need to be considered whilst noting the effects on the amount of states and therefore the number of state-levels required for the hBots. There should also be consideration to the relative difficulty of distinguishing object shapes from one another where their similarities and differences change with the complexity of the individual object shapes. Although work was carried out on reducing the number of redundant states in the system, further analyses would be required as the number of state-levels increased to reduce the amount of redundant states, which all take up memory for each of the agents. This consideration is increasingly important as the size of the agents in the physical world decrease. There needs to be further consideration regarding how the hBots sense each other and the object shape. Currently the agents can sense how many object cells they are next to and this informs the first state-levels. However, it may be possible to reduce this sensor ability to, 'am I next to an object cell or not' thus reducing the required external sensor capabilities of the future physical robotic agents. The focus would be more on the relationships between the position and states of the agents that are neighbouring each other, this would be far more controllable than multiple sensors devised for external purposes. To do this strategy in practice would require knowledge of the neighbouring agents positions relative to each other, this in fact should give a very similar output to the current state rules, although would require testing and clarifying.

Currently the simulation deals with objects in a two-dimensional lattice, it would be interesting to find if the same strategies could be implemented into a three-dimensional grid-world allowing for the differentiation of shapes that have width, depth and height. This move into three-dimensions would be a massive step towards real world applications which are more varied. The potential problems to overcome is the ever increasing size of the state relationship rules as the potential object shapes become increasingly complicated and this is exacerbated by the move into three dimensions.

The object shapes used in all the experiments were also static whilst been identified by the swarm of agents. It was mentioned that by utilising quicker state relation reactions and by not relying on building up an image of an object shape would allow for the agents to deal with object shape that are moving. This hypothesis requires testing.

Finally further development could be carried out for the GA. All of the variables for the GA remained static. Changing the parent selection mechanism, survivor selector system or mutation rate could have interesting effects of the GA. All of the tests that were carried out using the hBots considered only two object shapes both with a cell allowance of four. It is likely that real world applications will have more than two object shape types where it would be possible to have multiple wanted shapes and multiple unwanted shapes. The relationship between the differences of these multiple object shapes would include further complications to the scenarios and test the limitations of the GA further. This would then reflect back into the need to improve the effectiveness of the GA itself.

An additional area of research which shares a number of similarities with the current task scenarios are invader detection tasks. The agents could be trained to monitor an area of known object shapes and act on any unknown object shapes. The precise details of the action the agents take would depend on the specific task.

11.4.1 Moving to a Physical Platform

There are a number of areas that need to be considered to move the strategies developed in the SHM to a physical platform. These include how the agents move around their environment, how they interact with object shapes and how they interact with each other. One of the initial goals of the research was to create individually simple agents in the hope that their size could be reduced opening the number of applications available for cooperative object recognition systems. The testing of variations of these simple agents on similar physical scenarios will provide useful insight into what is possible and how simple the agents can actually be made. In a situation where a specific task or scenario is determined it will be possible to choose the most important elements from the system and change the others as necessary to complete the task. The behaviours and reactions the hBots have used will need reconsidering when completing this conversion from simulation to physical platform.

11.4.1.1 Movement of the Agents

The agents could use Brownian motion to move within a liquid environment causing the interactions with each other and the object shapes to occur at random. This, however, does not consider when the agents are required to stay still next to an object shape. In the later experimentation the agents had two different probabilities of staying near an object shape. For this to work in a physical system the agents would need a way to halt their movement. Being near an object shape may reduce the chance of movement but in an uncontrolled manner. If it were to be done with a physical system the agents would either require grippers to hold onto the object shape, which may limit the type of objects used, or find a way to reduce their random movement. Giving the agents either a propulsion system or changing their shape in some manner would increase their complexity moving away from the ideal of a simple agent without mechanical actuation.

A situation or task may exist where a more traditional mobile agent is used, in this situation it would clearly be possible to have the agents stop their movement as required. Although this would most likely be at the sacrifice to the minimum size of the agents.

11.4.1.2 Sensing Object Shapes

In the SHM the agents are aware if they neighbour an object shape on one, two or three of their sides. It is believed that this capability could be reduced to having the agent aware of if it is against an object shape or not. This reduction in object shape knowledge would require that the agents are aware of the position of their neighbouring agents as well as their state. To determine the position of an object or an agent requires multiple sensors, one for each side, that can then determine where the physical contact is made. This sensor could use switches controlled by whiskers or pressure pads, or a conductive material. Whichever system is used the agents need to know when it is touching an object shape as opposed to an agent, and the state of any agent it contacts. The simplest test is that if the side that is contact with an object or agent is not receiving state information then by reduction it must be part of an object shape. This will lead to testing and research with fault tolerance, and the consideration of how an agent that is not transmitting a state will appear to another agent.

11.4.1.3 Sensing other agents and describing states

The hBots were capable of reaching and communicating 264 states. Any neighbouring agent is required to be capable of understanding each of those different states at six

different positions around their perimeter. The distance this information is required to travel is short as the agents would be in contact with each other. For the physical agents to achieve this they would need hardware to both transmit and receive this information. A suitable system could involve nano scale radio transmitters and receivers. The current method of constant communication may need to be reconsidered as it would potentially cause a large drain on the power source.

11.4.1.4 Power, storage and gathering

One final and major consideration is the source of power for the agents' actions and calculations. The more hardware and capabilities that are added to the individual agents the more energy they will require to run. As this requirement increases the need for larger batteries or a method for directly converting energy from the environment for use by the agents. All of these issues will likely influence the minimum size limitations of the individual agents.

11.5 Closing Statement

The research presented here demonstrates a novel approach to a cooperative object recognition tasks for distinguishing between objects through their shape using a swarm of individually simplistic agents, inspired by the techniques utilised in swarm robotics. This approach provides a grounding for further research on the subject area by providing a strategy of states-relationships and state-behaviours that allow the agents to cooperate in a simulated arena. These strategies are suitable for further development considering small scale robotic applications. The metric of difficulty of both scenario task and finding the suitable behaviours will allow continued research of increasingly complex and varied problems to be analysed with consideration of their relative difficulties.

References

AFSHAR, M.H. (2012) Rebirthing genetic algorithm for storm sewer network design. *Scientia Iranica*, 19(1), pp. 11-19.

ALTSHULER, E.E. and LINDEN, D.S. (1997) Wire-antenna designs using genetic algorithms. *IEEE Antennas and Propagation*, 39(2), pp.22-43.

ALTSHULER, Y., WAGNER, I.A. and BRUCKSTEIN, A.M. (2009) To add with caution – decreasing a swarm robotics' efficiency by imprudently enhancing the robots capabilities. In: *Autonomous Robots and Agents, 2009. ICARA 2009. 4th International Conference on. Wellington, New Zealand, 10-12 February 2009*. pp. 351-356.

AMPATZIS, C., TUCI, E., TRIANNI, V., and DORIGO, M.. (2006) Evolution of signalling in a group of robots controlled by dynamic neural networks. In: ŞAHIN, E., WILLIAM, S. and WINFIELD, A.F.T. (eds.) *Swarm Robotics, LNCS 4433*. Berlin, Heidelberg: Springer, pp. 173-188.

ARBUCKLE, D. and REQUICHA, A.A.G. (2004) Active self-assembly. In: *Robotics and Automation, 2004. Proceedings. ICRA '04. 2004 IEEE International Conference on, 26 April-1May 2004*. Vol.1, pp. 896-901.

BAHÇECI and ŞAHIN (2005) Evolving aggregation behaviours for swarm robotic systems: A systematic case study. In: *Proceedings of IEEE Swarm Intelligence Symposium, (SIS 2005), 8-10 June 2005*. pp.333-340.

BALCH, T. and ARKIN, RC (1995) Motor Schema-based formation control for multiagents robot teams. In: *Proceedings of the First International Conference on Multi-Agent System*. AAAI Press, pp. 10-16.

BALCH, T. and HYBINETTE, M. (2000) Social potentials for scalable multi-robot formations. In: *Robotics and Automation, 2000. Proceedings. ICRA '00. IEEE International Conference on*. Vol.1, pp. 73-80.

BASSETT, J.K. and DeJONG, K.A. (2000) Evolving behaviours for cooperating agents. In: ZBIGNIEW, R and SETSUO, O (eds.) *Foundations of Intelligent Systems, LNCS 1932*. Berlin, Heidelberg: Springer, pp. 157-165.

BATALIN, M.A. and SUKHATME, G.S. (2002) Spreading out: a local approach to multi-robot coverage. In: *Proceedings of the 6th International Symposium on Distributed Autonomous Robotics Systems, Fukuoka, Japan 25-27 June 2002*. pp. 373-382.

BAXTER, J.L., BURKE, E.K., GARIBALDI, J.M. and NORMAN, M. (2006) The effect of potential field sharing in multi-agent systems. In: *The 3rd International Conference on Autonomous Robots and Agents (ICARA 2006), December 2006*. pp. 33-38.

BAXTER, J.L., BURKE, E.K., GARIBALDI, J.M. and NORMAN, M. (2007) Multi-robot search and rescue: a potential field based approach. In: MUKHOPADHYAY, S and GUPTA, G (eds.) *Autonomous Robots and Agents, Studies in Computational Intelligence*. 1st. Berlin: Springer, 76, pp. 9-16.

BECKERS, R. HOLLAND, O.E. and DENEUBOURG, J.L. (1994) From local actions to global tasks: stigmergy and collective robotics. In: *Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems Artificial Life IV, 1994*. MIT Press, pp. 181-189.

BEEKMAN, M., SWORD, G.A., and SIMPSON, S.J. (2008) Biological Foundations of Swarm Intelligence. In: BLUM, C. and MERKLE, D. (eds) *Swarm Intelligence, Natural Computing Series*. Berlin, Heidelberg: Springer-Verlag, pp. 3-42.

BENI, G. (1998) The concept of cellular robotic systems. In: *Proceedings 3rd IEEE International Symposium on Intelligent Control. Arlington, VA, 24-26 August 1988*. pp. 57-62.

BENI, G. (2005) From swarm intelligence to swarm robotics. In: SAHIN, E. and SPEARS, W.M. (eds.) *Swarm Robotics, LNCS 3342, 2005*. Berlin: Springer, pp. 1-9.

BLUM, C. and Li, X. (2008) Swarm intelligence in optimization. In: BLUM, C. and MERKLE, D. (eds) *Swarm Intelligence, Natural Computing Series*. Berlin, Heidelberg: Springer-Verlag, pp. 43-85.

BOWYER, A. (2000) Automated construction using co-operating biomimetic robots. Technical Report, University of Bath Department of Mechanical Engineering. Bath, UK [online]. Available at: http://biomimetic.pbworks.com/f/Automated+Construction+using+Co-operating+BiomimeticBowyer.pdf [Accessed: 27 September 2012]

BÜKER, U. (2000) Cooperative agents for object recognition. In: Proceedings of 15th International conference on Pattern Recognition, 2000. Vol. 4, pp.157-160.

CAMPO, A., NOUYAN, S., BIRATTARI, M., GROSS, R., and DORIGO, M. (2006) Negotiation of goal direction for cooperative transport. In: DORGIO, M. et al. (eds.) *Ant Colony Optimization and Swarm Intelligence, LNCS*. Berlin, Heidelberg: Springer, pp.191-202.

CAMPO, A. and DORIGO, M. (2007) Efficient multi-foraging in swarm robotics. In: *Proceedings of the 9th European conference on Advances in artificial life, 2007*. Berlin, Heidelberg: Springer-Verlag, pp. 696-705.

CANTÚ-PAZ, E. (1998) A survey of parallel genetic algorithms. In: *Proceedings of 7th International Conference on Genetic Algorithms*, 1997. pp. 113–120.

CASSINIS, R., BIANCO, G., CAVAGNINI, A. and RANSENIGO, P. (1999) Strategies for navigation of robot swarms to be used in landmines detection. In: *Proceedings of 3rd European Workshop on Advanced Mobile Robots Eurobot '99. Züric, Switzerland*, 1999.

CHOUHAN, S.S. and NIYOGI, R. (2012) An analysis of the effect of communication for multi-agent planning in a grid world domain. *International Journal of Intelligent Systems and Applications*, 4 (5), pp. 8-15.

CLARK, C.M., ROCK, S.M. and LATOME, J-C. (2003) Dynamic networks for motion planning in multi-robot space systems. In: *Proceedings of the 7th International Symposium on Artificial Intelligence, Robotics and Automation in Space: i-SAIRAS, NARA, Japan, 19 March 2003*.

CRABBE, F.L. and DYER, M.G. (1999) Second-order networks for wall-building agents. In: *International Joint conference on Neural Networks, 1999*. Vol. 3, pp. 2178-2183.

DÉFAGO, X. and KONAGAYA, A. (2002) Circle formation for oblivious anonymous mobile robots with no common sense of orientation. In: *Proceedings of the second ACM international workshop of Principles of mobile computing*. pp. 97-104.

DÉFAGO, X. and SOUISSI, S. (2008) Non uniform circle formation algorithm for oblivious mobile robots with convergence toward uniformity. *Theoretical Computer Science*, 396 (1-3), pp. 97-112.

DENEUBOURG, J.L., GOSS, N., FRANKS, N., SENDOVA-FRANKS, A., DETRAIN, C. and CHRÉTIEN, L. (1991) The dynamics of collective sorting robot-like ants and ant-like robots. In: *Proceedings of "SAB" Conference, From Animals to Animats, Paris, 1990*. MIT Press, pp.356-363.

DE ROSA, M., GOLDSTEIN, S.C., LEE, P., CAMPBELL, J. and PILLAI, P. (2006) Scalable shape sculpting via hole motion: motion planning in lattice-constrained modular robots. In: *Proceedings of IEEE International Conference on Robotics and Automation, 15-19 May 2006*. pp. 1462-1468.

DIEZ, L., DENEUBOURG, J-L., HOEBEKE, L. and DETRAIN, C. (2011) Orientation in corpse-carrying ants: memory of chemical cues? *Animal Behaviour*, 81(6), pp. 1171-1176.

DORIGO, M., TUCI, E., GROSS, R., TRIANNI, V., LABELLA, T.H., NOUYAN, S., AMPATZIS, C., DENEUBOURG, J-L., BALDASSARRE, G., NOLFI, S., MONDADA, F., FLOREANO, D. and GAMARDELLA, L.M. (2005) The swarm-bots project. In: SHAIN, E. and SPEARS, W.M. (eds.) *Swarm Robotics, LNCS 3342*. Berlin: Springer, pp. 31-34.

EIBEN A.E. and SMITH, J.E. (2007) *Introduction to Evolutionary Computing*. 2nd ed. Berlin: Springer-Verlag.

ENGELS, B. and KAMPHANS, T. (2006) Randolph's game is np-hard! *Electronic Notes in Discrete Mathematics*, 25, pp. 49-53.

E-PUCK (2010) Publication information [online]. http://www.e-puck.org/index.php?option=com_content&view=article&id=34&Itemid=36 [Accessed 12/09/12]

FERNADES, C.M., RAMOS, V. and ROSA, A.C. (2005) Self-Regulated Artificial Ant Colonies on Digital Image Habitats. In: *International Journal of Lateral Computing, IJLC,* 2(1), pp. 1-8.

FLOCCHINI, P., PRENCIPE, G., SANTORO, N. and WIDMAYER, P. (2005) Gathering of asynchronous robots with limited visibility. *Theoretical Computer Science*, 337(1-3), pp. 147-168.

FOX, D., BURGARD, W., KRUPPA, H. and THRUN, S. (2000) A probabilistic approach to collaborative multi-robot localization. *Autonomous Robots*, 8(3), pp. 325-344.

FRANKS, N.R. (1986) Teams in social insects: group retrieval of prey by army ants (Eciton burchelli, Hymenoptera: Formicidae). *Behavioural Ecology and Sociobiology*, 18(6), pp.425-429.

FRANKS, N.R. and DENEUBOURG, J. (1997) Self-organizing nest construction in ants: individual worker behaviour and the nest's dynamics. *Animal Behaviours*. 54, pp. 779-796.

FRANKS, N.R., MALLON, E.B., BRAY, H.E., HAMILTON, M.J. and MISCHLER, T.C. (2003) Strategies for choosing between alternatives with different attributes exemplified by house-hunting ants. *Animal Behaviour*, 65(1), pp. 215-223.

FREDSLUND, J. and MATARIĆ, M.J. (2002) A general algorithm for robot formations using local sensing and minimal communication. *Robotics and Automation*, 18(5), pp. 837-846.

FUJIBAYASHI, K., MURATA, S., SUGAWARA, K. and YAMAMURA, M.. (2002) Self-organizing formation algorithm for active elements. In: *Proceedings of 21st IEEE Symposium on Reliable Distributed Systems.* pp. 416-421.

GERKEY, B., VAUGHAN, R.T. and HOWARD, A. (2003) The player/stage: tools for multi-robot distributed sensor systems. In: *Proceedings of the International Conference on Advanced Robotics, Coimbra, Portugal, 30 June – 3 July 2003*. pp. 317-323.

GILPIN, K., KOTAL, K., RUS, D. and VASILESCU, I. (2008) Miche: modular shape formation by self-disassembly. *The International Journal of Robotics Research*, 27(3-4), pp. 345-372.

GILPIN, K., KNAIAN, A. and RUS, D. (2010) Robot Pebbles, One centimetre modules for programmable matter through self-disassembly. In: *Proceedings of IEEE International Conference on Robotics and Automation,* May 2010, pp. 2485-2492.

GILPIN, K., KOYANAGI, K. and RUS, D. (2011) Making self-disassembling objects with multiple components in robot pebble system. In: *Proceedings of IEEE International Conference on Robotics and Automation,* 2011, pp. 3614-3621.

GILPIN, K. and RUS, D. (2012a) A distributed algorithm for 2d shape with smart pebble robots.

GILPIN, K. and RUS, D. (2012b) What's in the bag: a distributed approach to £D shape duplication.

GOLDSTEIN, S.C., CAMPBELL, J. and MOWRY, T.C. (2005) Programmable Matter. *Computer Science Department. Paper 766*. [online]. Available at: http://repository.cmu.edu/compsci/766

GOLDSTEIN, S.C. and MOWRY, T.C. (2004) Claytronics: A scalable basis for future robots. *Computer Science Department. Paper 770*. [online]. Available at: http://repository.cmu.edu/compsci/770

GOSS, S., ARON, S., DENEUBOURG, J-L. and PASTEELS, J.M. (1989) Self-organised shortcuts in the argentine ant. *Naturwissenschaften*, 76, pp. 579-581.

GRADY, S.A., HUSSAINI, M.Y. and ABDULLAH, M.M. (2005) Placement of wind turbines using genetic algorithms. *Renewable Energy*, 30(2), pp. 259-270.

GROß, R. and DORIGO, M. (2004a) Cooperative transport of object of different shapes and sizes. In: DORIGO, M. et al., (eds.) *Ant Colony Optimization and Swarm Intelligence, LNCS*. Berlin, Heidelberg: Springer, 3172, pp. 33-51.

GROß, R. and DORIGO, M.(2004b)Evolving a Cooperative Transport Behavior for Two Simple Robots. In: *Proceedings of the 6th International Conference on Artificial Evolution, EA 2003, Revised Selected Papers, LNCS 2936*. Berlin: Springer-Verlag, pp. 305-316.

GROß, R., BONANI, M., MONDADA, F. and DORIGO, M. (2005) Autonomous self-assembly in mobile robotics. Technical Report IRIIA/2005-002, IRIDIA. Institut de Recherches Interdisciplinaires et de D´eveloppements en Intelligence Artificielle. Universit´e Libre de Bruxelles [online]. Available at: citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.60.7857&rep=rep1&type=pdf [Accessed: 27 September 2012]

GROß, R., BONANI, M., MONDADA, F. and DORIGO, M. (2006) Autonomous self-assembly in swarm-bots. *IEEE Transactions on Robotics*, 22(6), pp. 1115-1130.

GROSSESCHALLAU, M., WITKOWSKI, U. and RÜCKERT, U. (2005) Low cost blue tooth communication for the autonomous mobile minirobot Khepera. In: *Proceedings of IEEE International Conference on Robotics and Automation, Spain, 2005*, pp. 4194-4199

GUYOT, L., HEINIGER, N. MICHEL, O. and ROHRER F. (2011) Teaching robotics with an open curriculum based on the e-puck robot, simulations and competitions. In: *Proceedings of 2nd International Conference on Robotics in Education (RiE 2011). Vienna, Austria, September, 2011.* pp. 53-58.

HAMILTON, W.D. (1971) Geometry for the Selfish herd. *Journal of Theoretical Biology*, 31(2), pp. 295-311.

HARLAN, R.M., LEVINE, D.B. and McCLARIGAN, S. (2001) The Khepera Robot and the kRobot class: a platform for introducing robotics in the undergraduate curriculum. In: *Proceedings of the 32nd SIGCSE Technical Symposium on Computer Science Education*, pp. 105-109.

HARRIS, A. and CONRAD, J.M. (2011) A survey of popular robotics simulators, frameworks and toolkits. In: *Proceedings of IEEE Southeastcon.* pp. 243-249.

HAMANN, H. and WÖRN, H. (2007) A analytical and spatial model of foraging in a swarm of robots. In: SAHIN, E. and SPEARS, W.M. (eds.) *Swarm Robotics*. Berlin, Heidelberg: Springer-Verlag, pp. 43-55.

HERIANTO, SAKAKIBARA, T. and KURABAYASHI, D. (2007) Artificial pheromone system using RFID for navigation of autonomous robots. *Journal of Bionic Engineering,* 3(4), pp. 245-253.

HROLENOK, B., LUKE, S., SULLIVEN, K. and VO, C. (2010) Collaborative foraging using beacons. In: *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems,* 3(3), pp. 1197-1204.

HSIANG, T.R., ARKIN, E.M., BENDER, M., FEKETE, S.P. and MITCHELL, J.S.B. (2004) Algorithms for rapidly dispersing robot swarms in unknown environments. In: BOISSONNAT, J-D. et al., (eds.) *Algorithmic Foundations of Robotics V, Springer Tracts in Advanced Robotics*. Berlin, Heidelberg: Springer, Vol. 7, pp. 77-94.

JACKSON, D.E. and CHÂLINE, N. (2007) Modulation of pheromone trail strength with food quality in Pharaoh's ant, Monomorium pharaonis. *Animal Behaviour*, 74(3), pp. 463-470.

JØRGENSEN, M.W., ØSTERGAARD, E. and LUND, H.H. (2004) Modular ATRON: Modules for a self-reconfigurable robot. In: *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 2068–2073.

KATASAGGELOS, A.K., KONDI, L.P., MEIER, F.W., OSTERMANN, J. and SCHUSTER, G.M. (1998) MPEG-4 and rate-distortion-based shape-coding techniques. *Proceedings of the IEEE*, 86(6), pp. 1126-1154.

KAZADI, S., ABDUL-KHALIQ, A. and GOODMAN, R. (2002) On the convergence of puck clustering systems. *Robotics and Autonomous Systems,* 38(2), pp. 93-117.

KEANE, A.J. and BROWN, S.M. (1996) The design of a satellite beam with enhanced vibration performance using genetic algorithm techniques. In: *Proceedings of ACEDC '96, PEDC, University of Plymouth, UK*,. pp. 107-113.

KENNEDY, J. and EBERHART, R. (1995) Particle swarm optimization. In: *Proceedings of IEEE International Conference on Neural Networks, November/December 1995*. Vol. 4, pp. 1942-1948.

KING, D. and BREEDON, P. (2011a) Towards Cooperative Robotic Swarm Recognition: Object Classification and Validation. In: *Proceedings of 3rd International Conference on Advanced Design Manufacture (ADM2010), Nottingham, UK, 8-10 September 2010*. Switzerland: Trans Tech Publication, pp. 320-324.

KING, D. and BREEDON, P. (2011b) Robustness and stagnation of a swarm in a cooperative object recognition task. In: *Advances in Swarm Intelligence, LNCS 6728*. Berlin, Heidelberg: Springer, pp. 19-27.

KLASING, R., MARKOU, E. and PELC, A. (2008) Gathering asynchronous oblivious mobile robots in a ring. *Theoretical Computer Science*, 390(1), pp. 27-39.

KRIEGER, M.J., BILLETER, J.B. and KELLER, L. (2000) Ant-like task allocation in cooperative robots. *Nature*, 406(6799), pp. 992-995.

KUBE, C.R. and BONABEAU, E. (2000) Cooperative transport by ants and robots. *Robotics and Autonomous Systems*, 30(1-2), pp. 85-101.

KUBE, C.R. and ZHANG, H. (1994) Collective robotics: From social insects to robots. *Adaptive Behaviour*, 2(2), pp. 189-218.

KUBE, C.R. and ZHANG, H. (1996) The use of perceptual cues in multi-robot box-pushing. In: *Proceedings of 1996 IEEE International Conference on Robotics and Automation*. pp. 2085-2090.

KUO, R.J., CHEN, C.H. and HWANG, Y.C. (2001) An intelligent stock trading decision support system through integration of genetic algorithm based fuzzy neural network and artificial neural network. *Fuzzy Sets and Systems*, 118(1), pp. 21-45.

LABELLA, T.H., DORIGO, M. and DENEUBOURG, J-L. (2006) Division of labor in a group of robots inspired by ants' foraging behaviour. *ACM Transactions on Autonomous and Adaptive Systems*, 1(1), pp. 4-25.ACM TAAS

LAENGLE, T. and LUETH, T.C. (1994) Decentralized control of distributed intelligent robots and subsystems. *Annual Review in Automatic Programming*, 19, pp. 281-286.

LAGOUDAKIS, M.G., MARKAKIS, E., KEMPE, D., KESKINOCAK, P., KLEYWEGT, A., KOENIG, S., TOVEY, C., MEYERSON, A. and JAIN, S. (2005) Auction-based multi-robot routing. In: *Robotics: Science and Systems (RSS)*. Available from: http://www.cs.ucla.edu/~awm/papers/robotauction.pdf [Accessed 26 September 2012]

LATECKI, L.J. and LAKÄMER, R. (2002) Application of planar shape comparison to object retrieval in image databases. *Pattern Recognition*, 35(1), pp. 15-29.

LeBLANC, K. and SAFFIOTTI, A. (2008) Cooperative anchoring in heterogeneous multi-robot systems. In: *IEEE International Conference on Robotics and Automation, 19-23 May 2008*. pp. 3308-3314.

LEMAY, M., MICHAUD, F., LÉTOURNEAU, D. and VALIM, J-M.. (2004) Autonomous initialization of robot formations. In: *Proceedings, ICRA '04, IEEE International Conference on Robotics and Automation, 26-April – 1 May 2004*. pp. 3018-3023.

LERMAN, K., JONES, C., GALSTYAN, A. and MATARIĆ, M.J.. (2006) Analysis of dynamic task allocation in multi-robot systems. *The International Journal of Robotics Research*, 25(3), pp. 225-241.

LINDEN, D.S. (2002) Antenna design using genetic algorithm, In: *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'02)*. San Francisco, USA: Moran Kaufmann Publishers, pp. 113-1140.

LIONI, A., SAUWENS, C., THERAULAZ, G. and DENEUBOURG, J-L. (2001) Chain formation in OEcophylla longinoda. *Journal Insect Behavior*, 14(5), pp. 679–696.

LIU, Y. and PASSINO, K.M. (2004) Stable social foraging swarms in a noisy environment. *IEEE Transactions on Automatic Control*, 49(1), pp. 30-44.

LIU, W., WINFIELD, A.F.T., SA, J., CHEN, J. and DOU, L. (2007) Towards energy optimization: emergent task allocation in a swarm of foraging robots. *Adaptive Behaviour*, 15(3), pp. 289-305.

LIU, W., WINFIELD, A.F.T. and SA, J. (2007) Modelling swarm robotic systems: a case study in collective foraging. In: *Towards Autonomous Robotics Systems (TAROS 07), Aberystwyth, September 2007*. pp. 25-32.

LIU, W., and WINFIELD, A.F.T. (2011) Open-hardware e-puck Linux extension board for experimental swarm robotics research. *Microprocessors and Microsystems*, 35(1), pp. 60-67.

LUCARINI, G., VAROLI, M., CERUTTI, R. and SANDINI, G. (1993) Cellular robotics: simulation and HW implementation. In: *Proceedings, 1993 IEEE International Conference on Robotics and Automation, 2-6 May 1993*. pp. 846-852.

LUDWIG, L. and GINI, M. (2006) Robotic swarm dispersion using wireless intensity signals. In: GINI, M. and VOYLES, R. (eds.) *Distributed Autonomous Robotic Systems 7*. Japan: Springer, pp.135-144.

LUKE, S. and ZIPARO, V. (2010) Learn to behave! Rapid training of behaviour automata. In: GRZES, M. and TAYLOR, M. (eds.) *Proceedings of Adaptive and Learning Agents Workshop at AAMAS 2010*. pp. 61-68.

LUKE, S., CIOFFI-REVILLA, C., PANAIT, L., SULLICAN, K. and BALAN, G. (2005) MASON: a multi-agent simulation environment. *Simulation*, 81(7), pp. 517-527.

MADHAVEN, R. FREGENE, K. and PARKER, L.E. (2002) Distributed heterogeneous outdoor multi-robot localization. In: *Proceedings. ICRA '02. IEEE International Conference on Robotics and Automation, 2002*. pp. 374-381.

MAMEI, M. and ZAMBONELLI, F. (2007) Pervasive pheromone-based interaction with RFID tags. *ACM Transactions on Autonomous and Adaptive Systems*. 2(2) (in press).

MARTINS, D. and SIMONI, R. (2009) Enumeration of planar metamorphic robot configurations. In: *Proceedings of ASME/IFToMM REMAR 2009 Conference, London, June 2009*, pp. 580-588.

MASON, Z. (2002) Programming with stigmergy: using swarms for construction. In: STANDISH, R.K., BEDAU, M.A. and ADDASS, H.A. (eds) *Artificial Life VIII*. MIT Press, pp. 371-374

MATARIĆ, M.J., NILSSON, M. and SIMSARIN, K.T. (1995) Cooperative Multi-robot box pushing. In: *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems 95, Human Robot Interaction and Cooperative Robots, 5-9 August 1995*. pp. 556 -561.

MATARIĆ, M.J.,SUKHATME, G.S. and ØSTERGAARD, E.H. (2003) Multi-robot task allocation in uncertain environments. *Autonomous Robots*, 14(2-3), pp. 255-263.

McLURKIN, J. and DEMAINE, E.D. (2009) A distributed boundary detection algorithm for multi-robot systems. In: Proceedings of IEEE/RSJ international conference on Intelligent robots and systems, 2009, pp. 4791-4798.

McLURKIN, J. and SMITH, J. (2007) Distributed algorithms for dispersion in indoor environments using a swarm of autonomous mobile robots. In: ALAMI, R. CHATILA, R. and ASAMA, H. (eds.) *Distributed Autonomous Robotic Systems 6*. Japan: Springer, pp. 399-408.

MICHEL, O. (2004) Webots: professional mobile robot simulation. *International Journal of Advanced Robotic Systems*, 1(1), pp. 39-42.

MICONI, T. (2003) When evolving populations is better than coevolving individuals: The blind mice problem. In: *Proceedings 6th International Conference on Genetic Algorithms (ICGA-95), Pittsburgh, PA, 15-19 July*, pp. 271-278.

MONDADA, F., BONANI, M., RAEMY, X., PUGH, J., CIANCI, C., KLAPTOCZ, A., MAGNENAT, S., ZUFFEREY, J-C., FLOREANO, D. and MARTINOLI, A. (2009) The e-puck, a Robot Designed for Education in Engineering. In: *Proceedings of the 9th Conference on Autonomous Robot Systems and Competitions*. 1(1) pp. 59-65.

MONADA, F., FRANZI, E. and GUIGNAND, A. (1999) The Development of Khepera. In: *Proceedings of the 1st International Khepera Workshop, Paderborn, Germany, December 10-11, 1999*, pp. 7-13.

NOUYAN, S. et al. (2006) Group transport along a robot chain in a self-organised robot. In: ARAI, T. et al., (eds.) *Intelligent Autonomous Systems*. IOS Press, pp. 433 – 442.

NUNES, P. et al. (2000) A contour-based approach to binary shape coding using a multiple grid chain code. *Signal Processing: Image Communication*, 15(7-8), pp. 585-599.

ØSTERGAARD, E.H. and LUND, H.H. (2003) Evolving control for modular robot units. In: *Proceedings of IEEE International Symposium on Computation Intelligence in Robotics and Automation*, July 2003, 2 pp. 886-892.

ØSTERGAARD, E.H. and LUND, H.H. (2004) Distributed cluster walk for the ATRON self-reconfigurable robot. In: *Proceedings of The 8th Conference on Intelligent Autonomous Systems (IAS-8), IOS Press, Amsterdam*, pp. 291–298.

ØSTERGAARD et al (2006) Design of the ATRON lattice based self-reconfigurable robot. *Autonomous Robot*, 21 pp. 165-183.

OSWALD, N. and LEVI, P. (1997) Cooperative Vision in Multi-Agent Architecture. In: DEL BIMBO, A. (ed.) *Image Analysis and Processing, LNCS 1310*. Berlin: Springer, pp. 709- 716.

OSWALD, N. and LEVI, P. (2001) Cooperative object recognition. *Pattern Recognition Letters*, 22(12), pp. 1273-1282.

PARK, H., MARTIN, G.R. and YU, A.C. (2008) Compact representation of contours using directional grid chain code. *Signal Processing: Image Communication*, 23(2), pp. 87-100.

PACKARD, N.H. and WOLFRAM, S. (1985) Two-dimensional cellular automata. *Journal of Statistical Physics*, 38(5-6), pp. 901-946.

PARKER, L.E., KANNA, B., TANG, F. and BAILEY, M. (2004) Tightly-coupled navigation assistance in heterogeneous multi-robot teams. In: *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems, 28 September – 2 October, 2004*. Vol. 1 pp. 1016-1022.

PARKER, L.E. (2002) Distributed algorithms for multi-robot observation of multiple moving targets. *Autonomous Robots*, 12(3), pp. 231-255.

PAVONE, M. and FRAZZOLI, E. (2007) Decentralized policies for geometric pattern formation and path coverage. In: *American Control Conference, 9-13 July 2007*. pp.3949-3954.

PAYTON, D., DAILY, M., ESTOWSKI, R., HOWARD, M. and LEE, C. (2001) Pheromone Robotics. *Autonomous Robots,* 11(3), pp. 319- 324.

PAYTON, D., ESTKOWSKI, R. and HOWARD, M. (2003) Compound behaviours in pheromone robotics. *Robotics and Autonomous Systems*, 44 (3-4), pp. 229-240.

POTTER, M.A., MEEDEN, L.A. and SCHULTZ, A.C. (2001) Heterogeneity in the coevolved behaviors of mobile robots: The emergence of specialists. In: *Proceedings of the 17th International Conference on Arificial Intelligence (IJCAI-2001).* pp. 1337-1343.

PUGH, J. and MARTINOLI, A. (2007) Inspiring and modeling multi-robot search with particle swarm optimization. In: *Swarm Intelligence Symposium, 1-5 April 2007*. IEEE, pp. 332-339.

PROCESSING (n.d.) *Home Page*, [online]. Available from: http://processing.org/ [Accessed 12 September 2012].

RAMOS, V. and ALMEIDA, F. (2004) Artificial ant colonies in digital image habitats a mass behaviour effect study on pattern recognition. In: *Proceedings of 2nd International Works on Ant Algorithms, ANTS 2000, Brussels, Belgium, September 2000*, pp. 113-124.

REKLEITIS, I.M., DUDEK, G. and MILIOS, E.E. (1997) Multi-robot exploration of an unknown environment, efficiently reducing the odometry error. In: *International Joint Conference in Artificial Intelligence, Nagoya, Japan, August 1997*. Vol. 2, pp. 1340-1346.

REN, W. and SORENSEN, N. (2008) Distributed coordination architecture for multi-robot formation control. *Robotics and Autonomous Systems*, 56(4), pp. 324-333.

REYNOLDS, C.W. (1987) Flocks, herds and schools: a distributed behavioral model. *ACM SIGGRAPH Computer Graphics*, 21(4), pp. 25-34.

REYNOLDS, C.W. (1993) An evolved, vision-based behavioral model of coordinated group motion. In: MEYER, J.A. et al.. (eds.) *From Animal to Animats, Proceedings 2nd International Conference on Simulation of Adaptive Behavior*. MIT Press, pp. 384-392.

RICCI, A., OMICINI, A., VIROLI, M., GARDELLI, L. and A, E.O. (2006) Cognitive stigmergy: a framework based on agents and artifacts. In: WEYNS, D., PARUNKA, H.D.V. and MICHEL, F. (eds) *Environments for Multi-Agent Systems III, Third International Workshop, E4MAS 2006. LNCS (LNAI) 4389*. Springer, pp. 124-140.

ROBINSON. P., HALL, P., WOLF, J., PHILLIPS, R., CULVERHOUSE, C.P.P., BRAY, R. and SIMPSON, A.J. (2004) *The technology and challenges of microsot robot football.*[online]. Available from: www.tech.plym.ac.uk/robofoot/publications/paulrobinson/Warwick%20paper%202004.pdf [Accessed 12 September 12].

ŞAHIN, E., LABELLA, T.H., DENEUBOURG, J-L., RASSE, P., FLOREANO, D., GAMBARDELLA, L., MONDADA, F., NOLFI, S. and DORIGO, M. (2002) SWARM-BOT: pattern formation in a swarm of self-assembling mobile robots. In: *IEEE International Conference on Systems, Man and Cybernetics, 6-9 Oct 2002*. Vol.4.

ŞAHIN, E. (2005) Swarm Robotics: From source of inspiration to domains of application. In: ŞAHIN, E. and SPEARS, W.M. (eds.) *Swarm Robotics, LNCS 3342*. Berlin, Heidelberg: Springer-Verlag, pp. 10-20.

SAJJANHAR, A. and LU, A. (1997) A grid based shape indexing and retrieval method. *Australian Computer Journal, Special Issue on Multimedia Storage and Archiving Systems*, 29, pp. 131-140.

SANCHES, G. and LATOMBE, J.C. (2002) Using a PRM planner to compare centralized and decoupled planning for multi-robot systems. In: *Proceedings of IEEE International Conference on Robotics and Automation, 2002*. Vol. 2, pp. 2112-2119.

SANTANA, P. and CORRIEA, L. (2011) Swarm cognition on off road autonomous robots. *Swarm Intelligence*, 5, pp. 45-72.

SATO, S., OTORI, K., TAKIZAWA, A., SAKAI, H., ANDO, Y. and KAWAMURA, H. (2002) Applying genetic algorithms to the optimum design of a concert hall. *Journal of Sound and Vibration*, 258(3), pp. 517-526.

SCNEIDER-FONTÁN, M. and MATARIĆ, M.J. (1998) Territorial multi-robot task division. *Robotics and Automation*, 14(5), pp. 815-822.

SCHOLTEN, D.K., and WILSON, S.G. (1983) Chain coding with a hexagonal lattice. IEEE Transactions on Pattern Analysis and Machine Intelligence, PAMI-5 (5), pp.526-533.

SEHAD, S. and TOUZET, C. (1994) Self-organizing map for reinforcement learning. In: Proceedings of From Perception to Action Conference, 1994, pp. 420-423.

SERUGENDO, G.D.M., FOUKIA, N., HASSAS, S., KARAGEORGOS, A., MOSTEFAOUI, S.K., RANA, O.F., ULIERU, M., VALCKENAERS, P. and VAN ART, C. (2004) Self-organisation: Paradigms and applications. In: SERUGENDO, G.D.M. et al., (eds.) *Engineering Self-Organising Systems, LNCS 2977*. Berlin, Heidelberg: Springer, pp. 1-19.

SENDOVA-FRANKS, A.B., SCHOLES, S.R., FRANKS, N.R. and MELHUISH, C. (2004) Brood sorting by ants: two phases and differential diffusion. *Animal Behaviour*, 68(5), pp.1095-1106.

SHELL, D.A. and MATARIĆ, M.J. (2006) On foraging strategies for large-scale multi-robot systems. In: *IEEE/RSJ International Conference on Intelligent Robots and Systems, 9-15 October 2006*. pp. 2717-2723.

SHEN, W.M., WILL, P. and GALSTYAN, A. (2004) Hormone-inspired self-organization and distributed control of robotic swarms. *Autonomous Robots*, 17(1), pp. 93-105.

SHILONI, A., AGMON, N. and KAMINKA, G.A. (2009) Of robot ants and elephants. In: *Proceedings of the 8th International Conference on Autonomous Agents and Multiagent Systems*. Vol. 1, pp. 81-88.

SHILONI, A., AGMON, N. and KAMINKA, G.A. (2011) Of Robot Ants and Elephants: A computational comparison. *Theoretical Computer Science*, 412(41), pp. 5771-5788.

SIMMONS, R., APFELBAUM, D., BURGARD, W., FOX, D., MOORS, M., THRUN, S., and YOUNES, H. (2000) Coordination for multi-robot exploration and mapping. In: *Proceedings of the AAAI National Conference on Artificial Intelligence, Austin, TX, July 2000*. AAAI, pp. 851-858.

SOUISSI, S., DÉFAGO, X and YAMASHITA, M. (2005) Eventually consistent compasses for robust gathering of asynchronous mobile robots with limited visibility. Research report, School of Information Science, Japan Advanced Institute of Science and Technology, IS-RR-2005-010, [online] pp. 1-20. Available at: https://dspace.jaist.ac.jp/dspace/bitstream/10119/4790/1/IS-RR-2005-010.pdf [Accessed: 27 September 2012]

SOYSAL, O. and ŞAHIN, E. (2005) Probabilistic aggregation strategies in swarm robotic systems. In: *Proceedings of IEEE Swarm Intelligence Symposium, 2005, 8-10 June*. pp. 325-332.

SOYSAL, O. and ŞAHIN, E. (2007) A macroscopic model for self-organized aggregation in swarm robotic systems. In: ŞAHIN, E. and SPEARS, W.M. (eds.) *Swarm Robotics, LNCS 4433*. Berlin, Heidelberg: Springer-Verlag, pp. 27-42.

SPEARS, W.M., SPEARS, D.F., HEIL, R., KERR, W. and HETTIARACHCHI, S. (2005) An overview of physicomimetics. In: ŞAHIN, E. and SPEARS, W.M. (eds.) *Swarm Robotics, LNCS 3342*. Berlin, Heidelberg: Springer-Verlag, pp. 84-97.

SPLETZER, J., DAS, A.K., FIERRO, R., TAYLOR, C.J., KUMAR, V. and OSTROWSHI, J.P. (2001) Cooperative localization and control for multi-robot manipulation. In: *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems, 2001*. Vol. 2, pp. 631-636.

STOLZMANN, W. and BUTZ, M. (2000) Latent learning and action planning in robot. In: LANZI, P.L., STOLZMANN, W. and WILSON, S.W. (eds.) *Learning Classifier Systems, From Foundations to Applications*, *LNCS 1813*, Berlin, Heidelberg: Springer-Verlag, pp. 301-317.

STUDER, G. and LIPSON, H. (2006) Spontaneous emergence of self-replicating structures in molecube automata. In: *Proceedings of the 10th International Conference on the Simulation and Synthesis of Living Systems (Artificial Life X), MIT Press, Cambridge, 2006*, pp. 227–233.

SUGAWARA, K. and SANO, M. (1997) Cooperative acceleration of task performance: foraging behavior of interacting multi-robots system. *Physica D: Nonlinear Phenomena,* 100(3-4), pp. 343-354.

SULLIVAN, K. and LUKE, S. (2011) Multiagent Supervised Training with Agent Hierarchies and Manual Behavior Decomposition. In: *Proceedings of the IJCAI 2011 Workshop on Agents Learning Interactively from Human Teachers (ALIHT).*

SUSNEA, I., VASILIU, G. and FILIPESCU, A. (2008) RFID digital pheromones for generating stigmergic behaviour to autonomous mobile robots. In: *4th WSEAS/IASME International Conference on DYNAMICAL SYSTEMS and CONTROL (CONTROL'08), Corfu, Greece, 26-28 October 2008*. pp. 20-24.

SUTTON, R.S. and BARTO, A.G. (1998) Reinforcement learning: an introduction. MIT Press, Cambridge, MA.

SUZUKI, I. and YAMASHITA, M. (1999) Distributed anonymous mobile robots: Formation of geometric patterns. *SIAM Journal on Computing*, 28(4), pp. 1347-1363.

TERADA, Y. and MURATA, S. (2006) Modular structure assembly using blackboard path planning system. In: *International Symposium on Automation and Robotics in Construction (ISARC 2006), Tokyo Institute of Technology, Japan, 2006*. pp. 852-857

THERAULAZ, G. and BONABEAU, E. (1999) A brief history of stigmergy. *Artificial Life*, 5(2), pp. 97-116.

THERAULAZ, G. and BONABEAU, E. (1995a) Coordination in distributed building, *Science*, 269, pp. 686–688.

THERAULAZ, G. and BONABEAU, E. (1995b) Modelling the collective building of complex architectures in social insects with lattice swarms. *Journal of Theoretical Biology*, 177(4), pp. 381-400.

THERAULAZ, G., GAUTRAIS, J., CAMAZINE, S. and DENEUBOURG, J-L. (2003) The formation of spatial patterns in social insects: from simple behaviours to complex structures. *Philosophical Transactions of the Royal Society London A*, 361, pp. 1263-1282.

THRUN, S., BURGARD, W. and FOX, D. (2000) A real-time algorithm for mobile robot mapping with applications to multi-robot and 3D mapping. In: *Proceedings of IEEE International Conference on Robotics and Automation*. Vol. 1, pp. 321-328.

TRIANNI, V., GROß, R., LABELLA, T.H., ŞAHIN, E. and DORIGO, M. (2003) Evolving aggregation behaviours in a swarm of robots. In: WOLFGANG, B. et al. (eds.) *Advances in Artificial Life, LNCS 2801*. Berlin, Heidelberg: Springer, pp. 865-874.

TRIANNI, V., LABELLA, T. and DORIGO, M. (2004) Evolution of direct communication for a swarm-bot performing hole avoidance. In: DORIGO, M. et al. (eds.) *Ant Colony Optimization and Swarm Intelligence, LNCS 3172*, Berlin, Heidelberg: Springer, pp.174-197.

TRIANNI, V., NOLFI, S. and DORIGO, M. (2006) Cooperative hold avoidance in a swarm-bot. *Robotics and Autonomous Systems*, 54(2), pp. 97-103.

TRIANNI, V. and TUCI, E. (2011) Swarm cognition and artificial life. In: *Proceedings of 10th European Conference, ECAL 2009, Budapest, Hungary, September 13-16, 2009, Revised Selected Papers, Part II*, pp. 270-277.

TUCI, E., GROß, R., TRIANNI, V., MONDADA, F., BONANI, M. and DORIGO, M. (2006) Cooperation through self-assembly in multi-robot systems. *Transactions on Autonomous and Adaptive Systems*, 1(2), pp. 115-150.

TUCI, E., MASSERA, G. and NOLFI, S. (2010) Active categorical perception of object shapes in a simulated anthropomorphic robotic arm. *IEEE Transactions on Evolutionary Computation*, 14(6), pp. 885-899.

TUCI, E., TRIANNI, V. and DORIGO, M. (2004) Evolving the feeling of time through sensory-motor coordination: a robot based model. *Parallel Problem Solving from Nature-PPSN VIII. Springer Berlin Heidelberg*, pp. 1001-1010.

WAWERLA, J., SUKHATME, G.S. and MATARIC, M.J. (2002) Collective construction with multiple robots. In: *Proceedings of 2002 IEEE/RSJ International*

Conference on Intelligent Robots and Systems, Lausanne, Switzerland, 2002. Vol. 3, pp. 2696-2701.

WERFEL, J. (2007) Robot search in 3D swarm construction. In: *Proceedings 1st International Conference on Self-Adaptive and Self-Organizing Systems, 9-11 July 2007*, pp. 363–366.

WERFEL, J., BAR-YAM, Y. and NAGPAL, R. (2005a) Construction by robot swarms using extended stigmergy. AI Memo AIM-2005-011, MIT CSAIL.

WERFEL, J., BAR-YAM, Y. and NAGPAL, R. (2005b) Building patterned structures with robot swarms. In *Proceedings 19th International Joint Conference on Artificial Intelligence (IJCAI'05), Edinburgh, Scotland*, 2005.

WERFEL, J. and NAGPAL, R. (2008) Three-dimensional construction with mobile robots and modular blocks. *International. Journal of Robotics Research*, 27(3-4), pp. 463–479.

WERFEL, J., YANEER BAR-YAM, RUS, D., and NAGPAL, R.. (2006) Distributed construction by mobile robots with enhanced building blocks. In: *Proceedings of IEEE International Conference on Robotics and Automation, 15-19 May 2006*, pp. 2787-2794.

WINFIELD, A.F.T., SA, J., FERNANDEZ-GAGO, M-C., DIXON, C. and FISHER, M. (2005) On formal specification of emergent behaviours in swarm robotic systems. *International Journal of Advanced Robotics Systems*, 2(4), pp. 363-370.

WINFIELD, A.F.T. (2009) Towards an engineering science of robot foraging. In: ASAMA, H. et al. (eds.) *Distributed Autonomous Robotic Systems 8*. Springer-Verlag Berlin Heidelberg, pp.185-192.

WINFIELD, A.F.T, and GRIFFITHS, F. (2010). Towards the emergence of artificial culture in collective robotic systems. In: *Symbiotic multi-robot organisms. Cognitive systems monographs, Vol.7*. Berlin, Heidelberg: Springer-Verlag, pp. 425-433.

WOLFRAM, S. (1982) Cellular automata as simple self-organizing systems. *Caltech preprint, CALT-68-938*. Available from: http://www.stephenwolfram.com/publications/articles/ca/82-cellular/ [Accessed 26 September 2012]

WOLFRAM (2002) A new kind of science. Wolfram Media. Available from: http://www.wolframscience.com/nksonline/toc.html [Accessed 26 June 2013]

YAMASHITA, M. and SUZUKI (2010) Characterizing geometric patterns formable by oblivious anonymous mobile robots. *Theoretical Computer Science,* 411(26), pp. 2433-2453.

YANG, J. and LUO, Z. (2007) Coalition formation mechanism in multi-agent systems based on genetic algorithms. *Applied Soft Computing*, 7(2), pp. 561-568.

YIM, M., SHEN, W-M., SALEMI, B., RUS, D., MOLL, M., LIPSON, H., KLAVINS, E. and CHIRIKJIAN,G.S.. (2007) Modular self-reconfigurable robot systems [grand challenges of robots]. *Robotics and Automation*, 14(1), pp. 43-52.

YE, Y and TSOTSOS (1997) On the Collaborative Object Search Team: a Formulation. In: WEIß, G. (ed) *Distributed Artificial Intelligence Meets Machine Learning Learning in Multi-Agent Environments, LNCS 1221*. Berlin: Springer, pp. 94-116.

YONG, C.H. and MIIKKULAINEN, R. (2001) Cooperative coevolution of multi-agent systems. Technical Report AI01-287, Department of Computer Sciences, The University of Texas at Austin [online]. Available at: http://www.cs.utexas.edu/~ai-lab/pubs/yong.tr287.pdf [Accessed: 27 September 2012]

ZHANG, D. and LU, G. (2004) Review of shape representation and description techniques. *Pattern Recognition*, 37(1), pp. 1-19.

ZHANG, D., XIE, G., YU, J. and WANG, L. (2007) Adaptive task assignment for multiple mobile robots via swarm intelligence approach. *Robotics and Autonomous Systems*. 55(7), pp. 572-588.

ZYKOV, V., CHAN, A. and LIPSON, H. (2007) Molecubes: An open source modular robotic kit. *Proceedings IROS*, 7.

ZYKOV, V., MYTILINAIOS, E., DESNOYER, M. and LIPSON, H. (2007). Evolved and designed self-reproducing modular robotics. *Robotics, IEEE Transactions on*, 23(2), pp. 308-319.

ZYKOV, V., PHELPS, W., LASSABE, N. and LIPSON, H. (2008). Molecubes extended: Diversifying capabilities of open-source modular robotics. In: *International Conference on Intelligent RObots and Systems, Self-Reconfigurable Robots Workshop (IROS 08).*

Appendices

Appendix A: The Simplified Hexagonal Model for Initial Investigation

This program, written in Processing, is used in the initial investigation. The variables for each of the experiments are manipulated by changing the text. Results from this set of experiments influenced the later design of the SHM which was re-written for the later experiments. The results from this program are discussed in Chapter 6 of the current research project.

```
////////////////////////////////////////////////////////
// Simplified Hexagonal Model for Initial Investigation //
////////////////////////////////////////////////////////

PrintWriter output;

// variables ////////////////////////////////////////
float probabilityOfMovingAway = 0.0;  // between 0 and 1
float hexWidth = 10;           //width of each hexagon
int cols = 45;// 55;           // number of columns (odd number)
int numberOfBots = 10;        // number of hBots with centre hole = 5,
                              // (30,66,108,156,210,270)
int[] randOrder;

int pushers = 4;              // number of hBots required to move object

int noOfTriangles = 3;
int triSize = 6;              //number of hexagons making up triangle 6,10
int noOfHexagons = 3;
int hexSize = 7;              // number of hexagons making up hexagon 7,19
int largestSize = 7;          // the largest number of hexagons in either a triangle or hexagon
boolean triObjValid = false;
boolean hexObjValid = !triObjValid;
String validType;

int centreHoleConst = 5;            // change size of centre hole (top line size)


float hexHeight = 4 * (0.5 * hexWidth) * (tan((radians(30))));
int rows = cols;
int centreSpot = (cols-1)/2;

int noOfShapes = noOfTriangles + noOfHexagons;

// 6 Positions for objects (left to right, top to bottom)
int x0 =   ((cols-1)/2);
int y0 =   int(((cols)/4)-(centreHoleConst/2));
int x1 =   int(((3*cols)/4)+(centreHoleConst/2));
int y1 =   int(((cols)/4)-(centreHoleConst/2));
int x2 =   int(((cols)/4)-(centreHoleConst/2));
int y2 =   ((cols-1)/2);
int x3 =   int(((3*cols)/4)+(centreHoleConst/2));
int y3 =   ((cols-1)/2);
int x4 =   int(((cols)/4)-(centreHoleConst/2));
int y4 =   int(((3*cols)/4)+(centreHoleConst/2));
int x5 =   ((cols-1)/2);
int y5 =   int(((3*cols)/4)+(centreHoleConst/2)); //cols-17;

color white = color(255,255,255);    // empty cell
color black = color(60,60,60);       // object
color darkGrey = color(1,1,1);       // wall
color grey = color(120,120,120);     // hBot state 0
color green = color(0,200,0);        // hBot state 1
color blue = color(0,0,200);         // hBot state 2
color red = color(200,0,0);          // hBot state 3, identified triangle
color purple = color(200,0,200);     // hBot state 4, identified hexigon


// calculate the window size based on the number and size of grid
int xWindow = int((1.5*cols*hexWidth)-(12*hexWidth));
int yWindow = int((cols*hexHeight*0.75)+(hexHeight*0.25));

int steps = 0; // number of steps taken to complete the task

Cell[][] grid;   // the hexagonal grid is made up of individual cells
Wall[] wall;   // the walls of the arena
Bot[] hBot;      // the bots move around from cell to adjacent cell

ObjectCell[][] oCell; // a single hexagon making part of an object
TriangularObject[] triObj;
HexagonalObject[] hexObj;

int maxTests = 50; // 50
```

```
int currentTest = 0;
int results[] = new int[3];
int noTimeCouldNotMove = 0;
int noOfValidRemoved = 0;
int maxNumberOfBots = 200; // 200

int maxTimeSteps = 15000;

boolean useGraphicDisplay;
boolean first = true;

void setup() {
  //frameRate(1);

  if(hexObjValid){
    validType = "HexValid-";
  }
  if(triObjValid){
    validType = "TriValid-";
  }

  size(xWindow,yWindow);
  rectMode(CENTER);
  background(0);

  // create arena, grid of cells
  grid = new Cell[cols][rows];
  for (int i = 0; i < cols; i++) {
    for (int j = 0; j < rows; j++) {
      // xPosition,yPosition,colour
      grid[i][j] = new Cell(i*hexWidth,j*hexHeight*0.75,white);
    }
  }

  // create walls (remove top left triangle of rhombus to create hexagon)
  for (int j = 0; j < cols/2; j++)
    for (int i = 0; i < (cols/2)+2 - j; i++){
      grid[i][j] = null;
    }

  // create walls (remove bottom right triangle of rhombus to create hexagon)
  for (int j = 0; j < (cols-1)/2; j++)
    for (int i = cols - j; i < cols; i++){
      grid[i - 2][j + ((cols-1)/2)] = null;
    }

  // create walls (provide border round edges)
  for (int i = 0; i < cols; i++) {
    for (int j = 0; j < cols; j++) {
      if (i < 2 || i > cols-3 || j < 2 || j > cols-3)
        grid[i][j] = null;
    }
  }

  // create centre hole (for valid objects to be collected
  int centreHole = centreHoleConst;
  int jSwitch = 0;
  for (int j = 0; j < (centreHole*2)-1; j++) {
    if (j == centreHole)
      jSwitch +=2;
    if (j > centreHole)
      jSwitch += 2;

      for (int i = 0; i < centreHole+j-jSwitch; i++){
        int xPos = i+((cols-1)/2)-j;
        int yPos = j+((cols-1)/2)+ 1 - centreHole;

        if (j > centreHole-2){
          xPos += j - centreHole+1;

        }
        grid[xPos][yPos] = null;
      }
  }

  // create robots (surrounding centre hole)
  hBot = new Bot[numberOfBots];
  int xStart = 0;
  int yStart = 0;
  int deleteAmount = 0;

    for (int j = 0; j < numberOfBots; j++) {
      int i = j - deleteAmount;

      if (i == (centreHole*6)){
        deleteAmount += (centreHole*6);
        i = i - (centreHole*6);
        centreHole++;
      }

      xStart = (cols-1)/2 + (i);            // across top
      yStart = ((cols-1)/2)-centreHole;

      if (i > (centreHole)){ // top right side
        xStart = xStart - (i - centreHole);
        yStart = yStart + (i - centreHole);
      }

      if (i > (centreHole*2)) { // bottom right side
        xStart = xStart - (i - (centreHole*2));
      }
```

```
      if (i > (centreHole*3)) { // bottom side
        xStart = ((cols-1)/2) - (i - (centreHole*3));
        yStart = ((cols-1)/2)+centreHole;
      }

      if (i > (centreHole*4)) { // bottom left side
        xStart = xStart + (i - (centreHole*4));
        yStart = yStart - (i - (centreHole*4));
      }

      if (i > (centreHole*5)) { // top left side
        xStart = xStart + (i - (centreHole*5));
      }

      hBot[j] = new Bot(xStart,yStart);
    }


  // create objects
  oCell = new ObjectCell[noOfShapes][largestSize];
  triObj = new TriangularObject[noOfShapes];
    int position = 0;
    triObj[0] = new TriangularObject(0);
    //triObj[1] = new TriangularObject(1);
    //triObj[2] = new TriangularObject(2);
    triObj[3] = new TriangularObject(3);
    triObj[4] = new TriangularObject(4);
    //triObj[5] = new TriangularObject(5);

  hexObj = new HexagonalObject[noOfShapes];
    //hexObj[0] = new HexagonalObject(0);
    hexObj[1] = new HexagonalObject(1);
    hexObj[2] = new HexagonalObject(2);
    //hexObj[3] = new HexagonalObject(3);
    //hexObj[4] = new HexagonalObject(4);
    hexObj[5] = new HexagonalObject(5);

  for(int i = 0; i < 3; i++){
    results[i] = 0;
  }

  output = createWriter(validType + numberOfBots + "hBots_" + probabilityOfMovingAway +
"probabilityOfMoving.txt");
  output.println(validType + numberOfBots + "hBots_" + probabilityOfMovingAway + "probabilityOfMoving");
  output.println("Removed 1st;Removed 2nd;Removed 3rd;Number of times could not move;Number of Valid Removed");
}

void draw() {

  if(!first){
    ////// start this is for initial tests only
    // Objects tally number of hBots trying to move them
    for (int i = 0; i < noOfShapes; i++) {
      for (int j = 0; j < largestSize; j++) {
        if (oCell[i][j] != null)  {
          oCell[i][j].senseSurroundings();
          oCell[i][j].sumUp();
        }
      }
    }

    // Objects Moved off Arena are Deleted
    for (int i = 0; i < noOfShapes; i++) {
      int tallyForDelete = 0;
      // delete hexagonal objects when a certain amount of parts are over the edge
      if (hexObj[i] != null){
        for (int j = 0; j < hexSize; j++){
          if (grid[oCell[i][j].x][oCell[i][j].y] != null){
            tallyForDelete++;
          }
        }
        if (tallyForDelete < 5){
          hexObj[i].delete();
          hexObj[i] = null;
          displayData(true); // true for hexagon
        }
      }
      // delete triangular objects when a certain amount of parts are over the edge
      if (triObj[i] != null){
        for (int j = 0; j < triSize; j++){
          if (grid[oCell[i][j].x][oCell[i][j].y] != null){
            tallyForDelete++;
          }
        }
        if (tallyForDelete < 4){
          triObj[i].delete();
          triObj[i] = null;
          displayData(false); // false for triangle
        }
      }
    }

    // Objects are moved
    for (int i = 0; i < noOfShapes; i++) {
      if (hexObj[i] != null){
        hexObj[i].tally(); // number of correct colour contacts
        if (hexObjValid == true)
          hexObj[i].move();
      }

      if (triObj[i] != null) {
        triObj[i].tally();
```

```
        if (triObjValid == true)
          triObj[i].move();
      }
    }
    ///// end this is for initial tests only
    ////////////////////////////////////////

    // refresh and check
    checkCurrentCellStates();

    //// HBOTS SENSE then MOVE
    randomiseOrder();

    // SENSE hBot SURROUNDINGS
    for (int i = 0; i < numberOfBots; i++) {
      hBot[i].senseSurroundings();
      hBot[i].changePosition(i);   // hBots in state 1 or 2 revert to state 0 when    //moved
    }
    //// end HBOTS SENSE then MOVE

    // refresh and check
    checkCurrentCellStates();

    /////// CHANGING HBOT STATE
    // HBOTS SENSE SURROUNDINGS
    for (int i = 0; i < numberOfBots; i++) {
      hBot[i].senseSurroundings();
    }

    // hBot change state
    for (int i = 0; i < numberOfBots; i++) {
      hBot[i].changeState();
    }
    //// end CHANGE HBOT STATE
  }

  // refresh and check
  checkCurrentCellStates();

  //// DISPLAY GRID OF CELLS
  if(useGraphicDisplay){
    for (int i = 0; i < cols; i++) {
      for (int j = 0; j < rows; j++) {
        if (grid[i][j] != null)
          grid[i][j].display(j);
      }
    }
  }
  //// end DISPLAY GRID OF CELLS

  //// UPDATE TEST
  // increase number of steps taken
  first = false;
  steps++;

  if(noOfValidRemoved == 3 || steps == maxTimeSteps){
    currentTest++;
    println(results[0] + ";" + results[1] + ";" + results[2] + ";" + noTimeCouldNotMove + ";" + noOfValidRemoved);
    output.println(results[0] + ";" + results[1] + ";" + results[2] + ";" + noTimeCouldNotMove + ";" +
noOfValidRemoved);

    if(currentTest == maxTests){

      output.flush();
      output.close();

      if(numberOfBots == maxNumberOfBots){
        exit();
      }
      else{
        currentTest = 0;
        numberOfBots += 10;
        output = createWriter(validType + numberOfBots + "hBots_" + probabilityOfMovingAway +
"probabilityOfMoving.txt");
        output.println(validType + numberOfBots + "hBots_" + probabilityOfMovingAway + "probabilityOfMoving");
        output.println("Removed 1st;Removed 2nd;Removed 3rd;Number of times could not move;Number of Valid
Removed");
      }
    }
    reset();
  }
  //// end UPDATE TEST
}

void checkCurrentCellStates(){
  // REFRESH CELLS
  for (int i = 0; i < cols; i++) {
    for (int j = 0; j < rows; j++) {
      if (grid[i][j] != null)
        grid[i][j].colour = white;
    }
  }

  // CHECK FOR HBOT POSITION AND STATE TO UPDATE DISPLAY CELLS
  for (int i = 0; i < numberOfBots; i++) {
    // change square colour where hBot is based on the state of the hBot
    if (grid[hBot[i].x][hBot[i].y] != null)
      grid[hBot[i].x][hBot[i].y].colourChange(hBot[i].currentState);
  }

  // CHECK FOR OBJECT POSITIONS
  for (int i = 0; i < noOfShapes; i++) {
```

```
    for (int j = 0; j < largestSize; j++) {
      if (oCell[i][j] != null && grid[oCell[i][j].x][oCell[i][j].y] != null)
        grid[oCell[i][j].x][oCell[i][j].y].colourChange(10);
    }
  }
}

void randomiseOrder(){
  // initialise randOrder for movement
  randOrder = new int[numberOfBots];

  for(int i = 0; i < numberOfBots; i++){
    randOrder[i] = i;
  }

  for(int i = 0; i < numberOfBots; i++)
  {
    int posA = (int) random(numberOfBots);
    int posB = (int) random(numberOfBots);
    int tempA = randOrder[posA];
    int tempB = randOrder[posB];

    randOrder[posA] = tempB;
    randOrder[posB] = tempA;
  }
}

void mouseClicked(){
    useGraphicDisplay = !useGraphicDisplay;
}

////////////////////////////////////////////////////////////////////////////////
// A Cell Object Class

class Cell {
  // A cell object knows about its location in the grid
  float x,y; //x,y location
  color colour;

  // Cell Constructor
  Cell(float tempX, float tempY, color tempColour){
    x = tempX;
    y = tempY;
    colour = tempColour;
  }

  //colour change
  void colourChange(int tempState) {
    if (tempState == 0)
        colour = grey;
    if (tempState == 1)
        colour = green;
    if (tempState == 2)
        colour = blue;
    if (tempState == 3)
        colour = red;
    if (tempState == 4)
        colour = purple;

    // solid object
    if (tempState == 10)
      colour = black;

    // wall object
    if (tempState == 11)
      colour = darkGrey;
  }

  void display(int j) {
    translate(0.5*hexWidth,0.5*hexHeight); //start with first cell fully in window

    noStroke();
    strokeWeight(2);
    fill(colour);


    ellipse(x+(j*0.5*hexWidth),y,hexWidth,hexWidth); // speeds up display
    translate(-0.5*hexWidth,-0.5*hexHeight); //reset centre co-ordinate

  }
}

////////////////////////////////////////////////////////////////////////////////
// Display Data Function

void displayData(boolean hexagonal) {
  if (hexagonal == true && hexObjValid == true)  {
    //Valid hex removed
    results[noOfValidRemoved] = steps;
    noOfValidRemoved++;
  }
  if (hexagonal == false && triObjValid == true) {
    // Valid tri removed
    results[noOfValidRemoved] = steps;
    noOfValidRemoved++;
  }
}

//////////////////////////////////////////////////////////////////////////////// Function to CREATE A GROU
P OF HEXAGON CELLS INTO TRIANGLE SHAPE

void triCreate(int xPos, int yPos, int i) {
```

```
  oCell[i][0] = new ObjectCell(xPos,yPos);
  oCell[i][1] = new ObjectCell(xPos-1,yPos+1);
  oCell[i][2] = new ObjectCell(xPos,yPos+1);

  if (triSize > 3) {
    oCell[i][3] = new ObjectCell(xPos-2,yPos+2);
    oCell[i][4] = new ObjectCell(xPos-1,yPos+2);
    oCell[i][5] = new ObjectCell(xPos,yPos+2);
  }

  if (triSize > 6) {
    oCell[i][6] = new ObjectCell(xPos-3,yPos+3);
    oCell[i][7] = new ObjectCell(xPos-2,yPos+3);
    oCell[i][8] = new ObjectCell(xPos-1,yPos+3);
    oCell[i][9] = new ObjectCell(xPos,yPos+3);
  }

}

////////////////////////////////////////////////////////////////////////////////// Function to CREATE A GROU
P OF HEXAGON CELLS INTO HEXAGON SHAPE
void hexCreate(int xPos, int yPos, int i) {

  oCell[i][0] = new ObjectCell(xPos,yPos);

  oCell[i][1] = new ObjectCell(xPos+1,yPos-1);
  oCell[i][2] = new ObjectCell(xPos+1,yPos);
  oCell[i][3] = new ObjectCell(xPos,yPos+1);
  oCell[i][4] = new ObjectCell(xPos-1,yPos+1);
  oCell[i][5] = new ObjectCell(xPos-1,yPos);
  oCell[i][6] = new ObjectCell(xPos,yPos-1);

  if (hexSize > 7) {
  oCell[i][7] = new ObjectCell(xPos+1,yPos-2);
  oCell[i][8] = new ObjectCell(xPos+2,yPos-2);
  oCell[i][9] = new ObjectCell(xPos+2,yPos-1);
  oCell[i][10] = new ObjectCell(xPos+2,yPos);
  oCell[i][11] = new ObjectCell(xPos+1,yPos+1);
  oCell[i][12] = new ObjectCell(xPos,yPos+2);
  oCell[i][13] = new ObjectCell(xPos-1,yPos+2);
  oCell[i][14] = new ObjectCell(xPos-2,yPos+2);
  oCell[i][15] = new ObjectCell(xPos-2,yPos+1);
  oCell[i][16] = new ObjectCell(xPos-2,yPos);
  oCell[i][17] = new ObjectCell(xPos-1,yPos-1);
  oCell[i][18] = new ObjectCell(xPos,yPos-2);

  }
}



//////////////////////
// reset

void reset(){
// create robots (surrounding centre hole)
  hBot = new Bot[numberOfBots];
  int xStart = 0;
  int yStart = 0;
  int deleteAmount = 0;
  int centreHole = centreHoleConst;

  for (int j = 0; j < numberOfBots; j++) {
    int i = j - deleteAmount;

    if (i == (centreHole*6)){
      deleteAmount += (centreHole*6);
      i = i - (centreHole*6);
      centreHole++;
    }

    xStart = (cols-1)/2 + (i);            // across top
    yStart = ((cols-1)/2)-centreHole;

    if (i > (centreHole)){ // top right side
      xStart = xStart - (i - centreHole);
      yStart = yStart + (i - centreHole);
    }

    if (i > (centreHole*2)) { // bottom right side
      xStart = xStart - (i - (centreHole*2));
    }

    if (i > (centreHole*3)) { // bottom side
      xStart = ((cols-1)/2) - (i - (centreHole*3));
      yStart = ((cols-1)/2)+centreHole;
    }

    if (i > (centreHole*4)) { // bottom left side
      xStart = xStart + (i - (centreHole*4));
      yStart = yStart - (i - (centreHole*4));
    }

    if (i > (centreHole*5)) { // top left side
      xStart = xStart + (i - (centreHole*5));
    }

    hBot[j] = new Bot(xStart,yStart);
  }
```

```
  // create objects
  oCell = new ObjectCell[noOfShapes][largestSize];
  triObj = new TriangularObject[noOfShapes];
    int position = 0;
    triObj[0] = new TriangularObject(0);
    //triObj[1] = new TriangularObject(1);
    //triObj[2] = new TriangularObject(2);
    triObj[3] = new TriangularObject(3);
    triObj[4] = new TriangularObject(4);
    //triObj[5] = new TriangularObject(5);

  hexObj = new HexagonalObject[noOfShapes];
    //hexObj[0] = new HexagonalObject(0);
    hexObj[1] = new HexagonalObject(1);
    hexObj[2] = new HexagonalObject(2);
    //hexObj[3] = new HexagonalObject(3);
    //hexObj[4] = new HexagonalObject(4);
    hexObj[5] = new HexagonalObject(5);

  for(int i = 0; i < 3; i++){
    results[i] = 0;
  }

  //
  steps = 0;
  noTimeCouldNotMove = 0;
  noOfValidRemoved = 0;

}



////////////////////////////////////////////////////////////////////////////////
// A Bot object class

class Bot {
  // A Bot object knows about its location in the grid, it's current state and the
  // state of surrounding grid.
  int x,y;   // x,y grid location
  int currentState; // state, determins action
  color[] sensed = {white,white,white,white,white,white}; // NE E SE SW W NW

  // Bot constructor
  Bot(int tempX, int tempY) {
    x = tempX;
    y = tempY;
    currentState = 0;

    // inner ring sensor
    for(int i = 0; i < sensed.length; i++){
      sensed[i] = white;
    }
  }

  // Change state based on sensed data
  void changeState() {

    int noOfObjContacts = 0;
    int noOfGreenContacts = 0;
    int noOfBlueContacts = 0;

    // checks for objects (black)
    for(int i = 0; i < sensed.length; i++){
      if(sensed[i] == black){
        noOfObjContacts++;
      }
    }

    // set current state to 0 (no contact) when there is no contact
    if (noOfObjContacts == 0)
      currentState = 0;

    // check for states 3 and 4
    if (currentState != 0){
      for(int i = 0; i < sensed.length; i++){
        // checks for state 3 (red)
        if (sensed[i] == red && noOfObjContacts > 0){
          if (currentState != 4) // blue (2) or green (1) changes to red (3)
            currentState = 3;
          if (currentState == 4) // purple (4) changes to grey (0) temporarily
            currentState = 0;
        }
        // checks for state 4 (purple)
        if (sensed[i] == purple && noOfObjContacts > 0){
          if (currentState != 3) // blue (2) or green (1) changes to purple (4)
            currentState = 4;
          if (currentState == 3) // red (3) changes to grey (0)
            currentState = 0;
        }

      }
    }


    if (currentState != 3 && currentState != 4) {
      // if contact is with object, change state
      if (noOfObjContacts == 1)
        currentState = 1;
      if (noOfObjContacts == 2)
        currentState = 2;
```

```
    // checks for state 1 contacts (green)
    for(int i = 0; i < sensed.length; i++){
      if (sensed[i] == green)
        noOfGreenContacts++;
      if (sensed[i] == blue)
        noOfBlueContacts++;
    }

    if (noOfObjContacts == 1 && noOfBlueContacts == 2 && noOfGreenContacts == 0)
      currentState = 4;

    if (noOfObjContacts == 1 && noOfBlueContacts == 1 && noOfGreenContacts == 1)
      currentState = 3;

  }


  // change to zero state when near edge
  for(int i = 0; i < sensed.length; i++){
    if (sensed[i] == darkGrey)
      currentState = 0;
  }
}

// Sense surroundings
void senseSurroundings() {
  // reset sensed values to white
  for(int i = 0; i < sensed.length; i++){
    sensed[i] = white;
  }

  if (grid[x + 1][y - 1] != null)
    sensed[0] = grid[x + 1][y - 1].colour; // NE looks at colour value of cell to determine colour
  if (grid[x + 1][y] != null)
    sensed[1] = grid[x + 1][y].colour; // E
  if (grid[x][y + 1] != null)
    sensed[2] = grid[x][y + 1].colour; // SE
  if (grid[x - 1][y + 1] != null)
    sensed[3] = grid[x - 1][y + 1].colour; //SW
  if (grid[x - 1][y] != null)
    sensed[4] = grid[x - 1][y].colour;   //W
  if (grid[x][y - 1] != null)
    sensed[5] = grid[x][y - 1].colour; //NW

  // sets sensed value to darkGrey (wall/hole) if value is null
  if (grid[x + 1][y - 1] == null)
    sensed[0] = darkGrey;
  if (grid[x + 1][y] == null)
    sensed[1] = darkGrey;
  if (grid[x][y + 1] == null)
    sensed[2] = darkGrey;
  if (grid[x - 1][y + 1] == null)
    sensed[3] = darkGrey;
  if (grid[x - 1][y] == null)
    sensed[4] = darkGrey;
  if (grid[x][y - 1] == null)
    sensed[5] = darkGrey;
}


// change position of bot based on surroundings and probability
void changePosition(int currentBot) {
  int xOld = x;
  int yOld = y;
                            // NE,  E,  SE,  SW,  W,   NW
  boolean possDirections[] = {true,true,true,true,true,true};
  int direction = -1;

  // check possible directions (don't move into object shape or arena boundary
  for(int i = 0; i < sensed.length; i++){
    if (sensed[i] == black || sensed[i] == darkGrey) {
      possDirections[i] = false;
    }
  }

  int[] checkAll = {0,0,0,0,0,0};
  boolean leaveLoop = false;

  do{
    direction = (int)(random(6)); // pick random direction
    checkAll[direction] = 1; // note direction has been selected
    int totalCheck = 0;
    for (int i = 0; i < checkAll.length; i++){
      totalCheck += checkAll[i]; // tally the number of directions selected
    }

    if (totalCheck == 6){ // if all directions selected
      leaveLoop = true;
      direction = -1; // if all directions are not possible stay still
    }

  }while (leaveLoop == false && possDirections[direction] == false); // if direction not possible loop

  if(direction != -1 && possDirections[direction] == false){
    println("bad move by hBot");
    exit();
  }

  if((currentState == 1 || currentState == 2) && (random(1) > probabilityOfMovingAway)){
    x = xOld;
    y = yOld;
  }
```

```
    // if next to valid object and in state 3 or 4 as appropriate stay still
    else if((currentState == 3 && triObjValid) || (currentState == 4 && hexObjValid)){
      // stay still
      x = xOld;
      y = yOld;
    }
    else{
      // move in the selected direction
      if (direction == 0){ // NE
        x += 1;
        y -= 1;
      }
      else if (direction == 1){ // E
        x += 1;
      }
      else if (direction == 2){ // SE
        y += 1;
      }
      else if (direction == 3){ // SW
        x -= 1;
        y += 1;
      }
      else if (direction == 4){ // W
        x -= 1;
      }
      else if (direction == 5){ // NW
        y -= 1;
      }
      else{
        x = x;
        y = y;
        // stay still
        //println("Stay still");
      }
    }

    // if movement isn't possible because of other hBot stay still
    // check for bots on top of each other
    for (int i = 0; i < numberOfBots; i++){
      if(i != currentBot){ // is this another hBot
        if(hBot[i].x == x && hBot[i].y == y){ // is there an agent in this cell
          //println("hBot in that cell!   " + direction); // move back to original possition
          x = xOld;
          y = yOld;
        }
      }
    }

    // when a state 1 or 2 hBot moves change it to state zero
    if ((xOld != x || yOld != y) && (currentState == 1 || currentState == 2))
        currentState = 0;
  }
}


////////////////////////////////////////////////////////////////////////////////////////// An Object Cell Class
class ObjectCell {

  int x,y;
  int sumPurple, sumRed;
  color[] sensed = {white,white,white,white,white,white};

  ObjectCell(int tempX, int tempY) {
    x = tempX;
    y = tempY;

    sumPurple = 0;
    sumRed = 0;

    // inner ring sensor
    for(int i = 0; i < sensed.length; i++){
      sensed[i] = white;
    }
  }

  // Sense surroundings
  void senseSurroundings() {
    // reset sensed values to white
    for(int i = 0; i < sensed.length; i++){
      sensed[i] = white;
    }

    // make sure sensed value is within range.
    if (x-1 > 0 && y - 1 > 0 && x+1 < cols && y + 1 < cols) {
      if (grid[x + 1][y - 1] != null)
        sensed[0] = grid[x + 1][y - 1].colour; // looks at colour value of cell to determine colour
      if (grid[x + 1][y] != null)
        sensed[1] = grid[x + 1][y].colour;
      if (grid[x][y + 1] != null)
        sensed[2] = grid[x][y + 1].colour;
      if (grid[x - 1][y + 1] != null)
        sensed[3] = grid[x - 1][y + 1].colour;
      if (grid[x - 1][y] != null)
        sensed[4] = grid[x - 1][y].colour;
      if (grid[x][y - 1] != null)
        sensed[5] = grid[x][y - 1].colour;

      // sets sensed value to darkGrey (wall/hole) if value is null
      if (grid[x + 1][y - 1] == null)
        sensed[0] = darkGrey;
      if (grid[x + 1][y] == null)
```

```
        sensed[1] = darkGrey;
      if (grid[x][y + 1] == null)
        sensed[2] = darkGrey;
      if (grid[x - 1][y + 1] == null)
        sensed[3] = darkGrey;
      if (grid[x - 1][y] == null)
        sensed[4] = darkGrey;
      if (grid[x][y - 1] == null)
        sensed[5] = darkGrey;
    }
  }

  // sum up total red and purple hexagons.
  void sumUp() {
    sumRed = 0;
    sumPurple = 0;

    for(int i = 0; i < sensed.length; i++){
      if (sensed[i] == purple)
        sumPurple++;
      if (sensed[i] == red)
        sumRed++;
    }
  }

}

// A triangular object
class TriangularObject {

  int x,y;
  int object;
  int totalRed, totalPurple;

  TriangularObject(int tempPos) {

    if (tempPos == 0) {
      x = x0 + 1;
      y = y0 - 1;
    }
    else if (tempPos == 1) {
      x = x1;
      y = y1 - 1;
    }
    else if (tempPos == 2) {
      x = x2 + 1;
      y = y2 - 1;
    }
    else if (tempPos == 3) {
      x = x3;
      y = y3 - 1;
    }
    else if (tempPos == 4) {
      x = x4;
      y = y4 - 1;
    }
    else if (tempPos == 5) {
      x = x5 + 1;
      y = y5 - 1;
    }

    object = tempPos;
    totalRed = 0;
    totalPurple = 0;

    triCreate(x,y,object);
  }

  void delete() {
    for (int j = 0; j < triSize; j++) {
      oCell[object][j] = null;
    }
  }

  void tally() {
    totalRed = 0;
    totalPurple = 0;

    for (int j = 0; j < triSize; j++) {
       totalRed = totalRed + oCell[object][j].sumRed;
       totalPurple = totalPurple + oCell[object][j].sumPurple;
    }
  }

  void move() {
    boolean testForMove = true;

    for (int i = 0; i < numberOfBots; i++) {
      //check of hBot contact with object
      if (   (hBot[i].x == x     && hBot[i].y == y - 1)
          || (hBot[i].x == x + 1 && hBot[i].y == y - 1)
          || (hBot[i].x == x + 1 && hBot[i].y == y)
          || (hBot[i].x == x + 1 && hBot[i].y == y + 1)
          || (hBot[i].x == x + 1 && hBot[i].y == y + 2)
          || (hBot[i].x == x     && hBot[i].y == y + 3)
          || (hBot[i].x == x - 1 && hBot[i].y == y + 3)
          || (hBot[i].x == x - 2 && hBot[i].y == y + 3)
          || (hBot[i].x == x - 3 && hBot[i].y == y + 3)
          || (hBot[i].x == x - 3 && hBot[i].y == y + 2)
          || (hBot[i].x == x - 2 && hBot[i].y == y + 1)
          || (hBot[i].x == x - 1 && hBot[i].y == y)){
```

```
            //is there a nearby robot in state 1
            for(int j = 0; j < 6; j++){
              if(hBot[i].sensed[j] == grey)
                testForMove = false;
            }
        }
      }

      if(testForMove == false){
        //println("CANNOT MOVE TRIANGULAR OBJECT");
        noTimeCouldNotMove++;
      }

      for (int j = 0; j < largestSize; j++) {
        if(oCell[object][j] != null){
          for(int k = 0; k < 6; k++){
            if(oCell[object][j].sensed[k] == grey)
              testForMove = false;
          }
        }
      }

      if (testForMove == true){
        int xMove = 0;
        int yMove = 0;

        // check for suitable number of contacts
        if (totalRed > pushers + 1) {

          if (object == 0){
            yMove = 1;
          }
          if (object == 1) {
            yMove = 1;
            xMove = -1;
          }
          if (object == 2) {
            xMove = 1;
          }
          if (object == 3) {
            xMove = -1;
          }
          if (object == 4) {
            yMove = -1;
            xMove = 1;
          }
          if (object == 5) {
            yMove = -1;
          }

          // switch direction if object invalid
          if (triObjValid == false) {
             yMove = -yMove;
             xMove = -xMove;
          }


          for (int j = 0; j < triSize; j++) {
            oCell[object][j].x += xMove;
            oCell[object][j].y += yMove;
          }

          // if a hBot surrounds the triangle it should move with it
          for (int i = 0; i < numberOfBots; i++) {
            if (   (hBot[i].x == x     && hBot[i].y == y - 1)
                || (hBot[i].x == x + 1 && hBot[i].y == y - 1)
                || (hBot[i].x == x + 1 && hBot[i].y == y)
                || (hBot[i].x == x + 1 && hBot[i].y == y + 1)
                || (hBot[i].x == x + 1 && hBot[i].y == y + 2)
                || (hBot[i].x == x     && hBot[i].y == y + 3)
                || (hBot[i].x == x - 1 && hBot[i].y == y + 3)
                || (hBot[i].x == x - 2 && hBot[i].y == y + 3)
                || (hBot[i].x == x - 3 && hBot[i].y == y + 3)
                || (hBot[i].x == x - 3 && hBot[i].y == y + 2)
                || (hBot[i].x == x - 2 && hBot[i].y == y + 1)
                || (hBot[i].x == x - 1 && hBot[i].y == y)){

                   hBot[i].x += xMove;
                   hBot[i].y += yMove;
                }
            }
            x += xMove; // update current object position
            y += yMove;
          }
        }
      }

    }

    // A hexagonal object
    class HexagonalObject {

      int x,y;
      int object;
      int totalRed, totalPurple;

      HexagonalObject(int tempPos) {

        if (tempPos == 0) {
          x = x0;
          y = y0;
        }
```

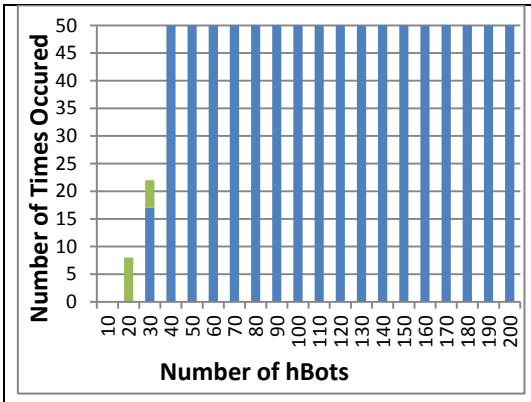
```
  else if (tempPos == 1) {
    x = x1;
    y = y1;
  }
  else if (tempPos == 2) {
    x = x2;
    y = y2;
  }
  else if (tempPos == 3) {
    x = x3;
    y = y3;
  }
  else if (tempPos == 4) {
    x = x4;
    y = y4;
  }
  else if (tempPos == 5) {
    x = x5;
    y = y5;
  }

  object = tempPos;
  totalRed = 0;
  totalPurple = 0;

  hexCreate(x,y,object);
}

void delete() {
  for (int j = 0; j < hexSize; j++) {
    oCell[object][j] = null;
  }
}

void tally() {
  totalRed = 0;
  totalPurple = 0;

  for (int j = 0; j < hexSize; j++) {
     totalRed = totalRed + oCell[object][j].sumRed;
     totalPurple = totalPurple + oCell[object][j].sumPurple;
  }
}

void move() {
  boolean testForMove = true;

  for (int i = 0; i < numberOfBots; i++) {
    //check of hBot contact with object
    if (    (hBot[i].x == x     && hBot[i].y == y - 1)
        || (hBot[i].x == x + 1 && hBot[i].y == y - 1)
        || (hBot[i].x == x + 1 && hBot[i].y == y)
        || (hBot[i].x == x + 1 && hBot[i].y == y + 1)
        || (hBot[i].x == x + 1 && hBot[i].y == y + 2)
        || (hBot[i].x == x     && hBot[i].y == y + 3)
        || (hBot[i].x == x - 1 && hBot[i].y == y + 3)
        || (hBot[i].x == x - 2 && hBot[i].y == y + 3)
        || (hBot[i].x == x - 3 && hBot[i].y == y + 3)
        || (hBot[i].x == x - 3 && hBot[i].y == y + 2)
        || (hBot[i].x == x - 2 && hBot[i].y == y + 1)
        || (hBot[i].x == x - 1 && hBot[i].y == y)){

          //is there a nearby robot in state 1
          for(int j = 0; j < 6; j++){
            if(hBot[i].sensed[j] == grey)
              testForMove = false;
          }
    }
  }

  if(testForMove == false){
    //println("CANNOT MOVE HEXAGONAL OBJECT");
    noTimeCouldNotMove++;
  }

  for (int j = 0; j < largestSize; j++) {
    if(oCell[object][j] != null){
      for(int k = 0; k < 6; k++){
        if(oCell[object][j].sensed[k] == grey)
          testForMove = false;
      }
    }
  }

  if (testForMove == true){

    int xMove = 0;
    int yMove = 0;

    // check for suitable number of contacts
    if (totalPurple > pushers + 1) {

      if (object == 0){
        yMove = 1;
      }
      if (object == 1) {
        yMove = 1;
        xMove = -1;
      }
      if (object == 2) {
        xMove = 1;
      }
```

```
        if (object == 3) {
          xMove = -1;
        }
        if (object == 4) {
          yMove = -1;
          xMove = 1;
        }
        if (object == 5) {
          yMove = -1;
        }

        // switch direction if object invalid
        if (hexObjValid == false) {
           yMove = -yMove;
           xMove = -xMove;
        }

        for (int j = 0; j < hexSize; j++) {
          oCell[object][j].x += xMove;
          oCell[object][j].y += yMove;
        }

        // if a hBot surrounds the hexagon it should move with it
        for (int i = 0; i < numberOfBots; i++) {
          if (   (hBot[i].x == x     && hBot[i].y == y - 2)
              || (hBot[i].x == x + 1 && hBot[i].y == y - 2)
              || (hBot[i].x == x + 2 && hBot[i].y == y - 2)
              || (hBot[i].x == x + 2 && hBot[i].y == y - 1)
              || (hBot[i].x == x + 2 && hBot[i].y == y)
              || (hBot[i].x == x + 1 && hBot[i].y == y + 1)
              || (hBot[i].x == x     && hBot[i].y == y + 2)
              || (hBot[i].x == x - 1 && hBot[i].y == y + 2)
              || (hBot[i].x == x - 2 && hBot[i].y == y + 2)
              || (hBot[i].x == x - 2 && hBot[i].y == y + 1)
              || (hBot[i].x == x - 2 && hBot[i].y == y)
              || (hBot[i].x == x - 1 && hBot[i].y == y - 1)){

                hBot[i].x += xMove;
                hBot[i].y += yMove;
              }
        }
        x += xMove; // update current object position
        y += yMove;
      }
    }
  }

}

////////////////////////////////////////////////////////////////////////////////
// A Wall Object ///////////////////////////////////////////////////////////////

class Wall {

  int x,y;

  Wall(int tempX, int tempY) {
    x = tempX;
    y = tempY;
  }

}
```

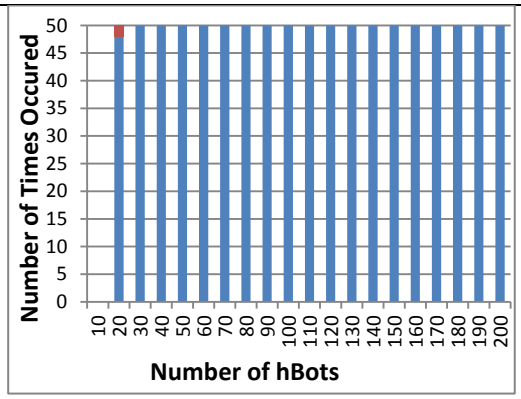
Appendix B: Initial Experiment Results

The results from the initial experiment regarding cooperative object recognition with both a hexagonal and triangular object shape are contained here. The tests considered the number of hBots in the swarm, ranging from 10 – 200, and the probability that a hBot in state 1 or state 2 would move away from the object shape, 0.0 – 1.0. The number of valid object shapes removed was recorded along with the number of time-steps to complete the task. Each test was repeated 50 times and the mean average, minimum, maximum and standard deviation are included on the plots.

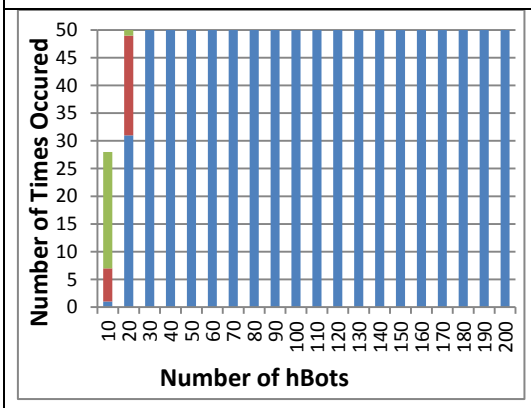
B.1: Number of Hexagonal Object Shapes Removed when Valid



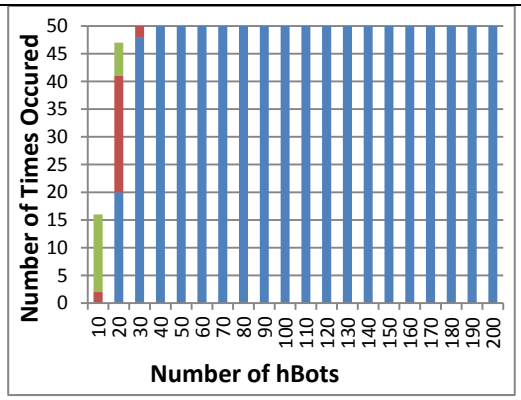
Hexagon Valid, 0.0 Probability of Movement



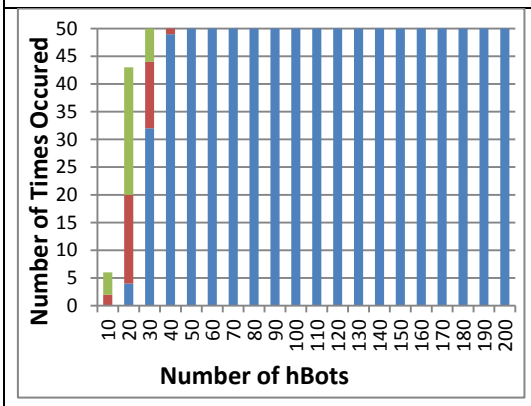
Hexagon Valid, 0.1 Probability of Movement



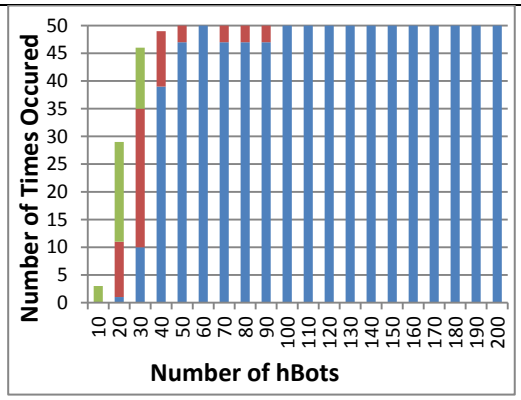
Hexagon Valid, 0.2 Probability of Movement



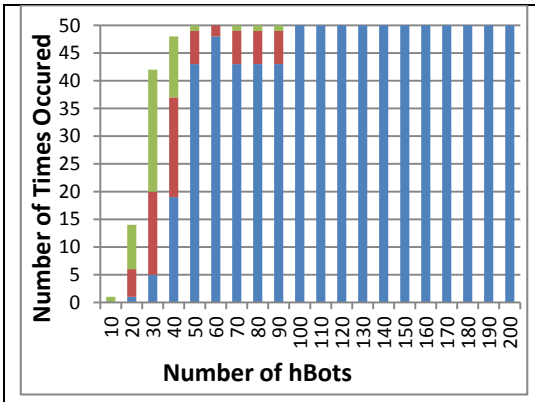
Hexagon Valid, 0.3 Probability of Movement



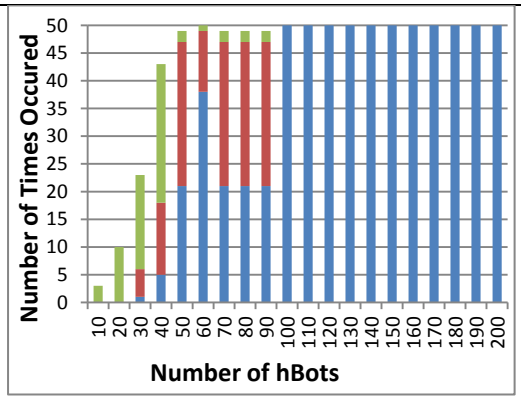
Hexagon Valid, 0.4 Probability of Movement



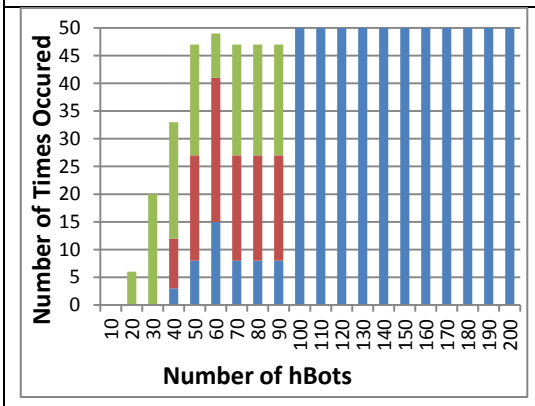
Hexagon Valid, 0.5 Probability of Movement



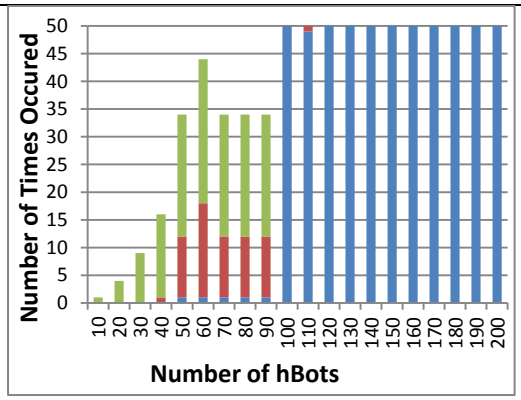
Hexagon Valid, 0.6 Probability of Movement



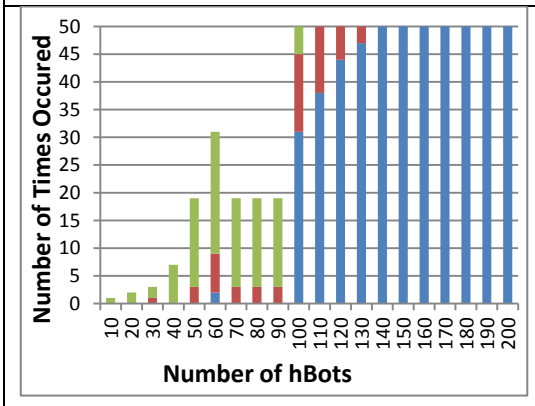
Hexagon Valid, 0.7 Probability of Movement



Hexagon Valid, 0.8 Probability of Movement

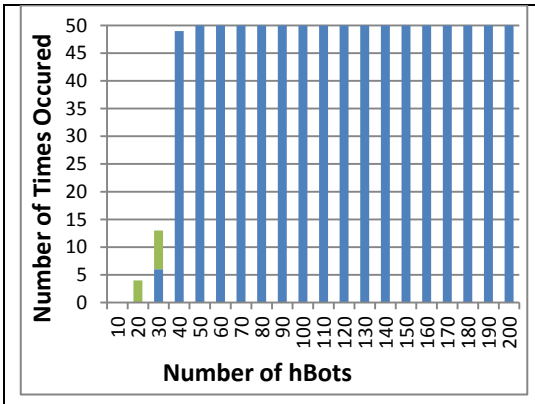


Hexagon Valid, 0.9 Probability of Movement

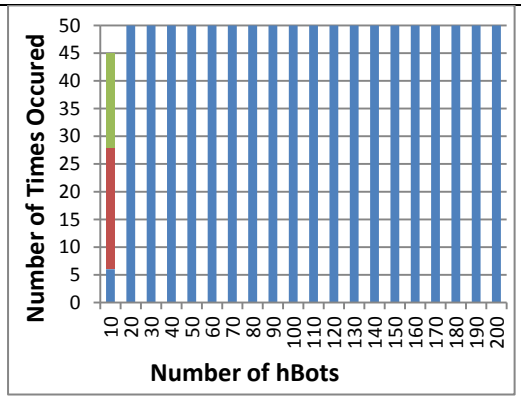


Hexagon Valid, 1.0 Probability of Movement

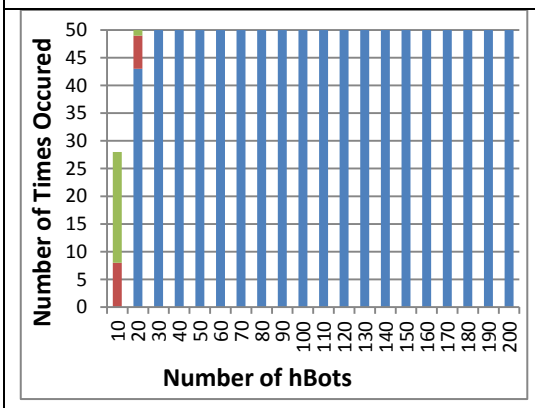
B.2: Number of Triangular Object Shapes Removed when Valid



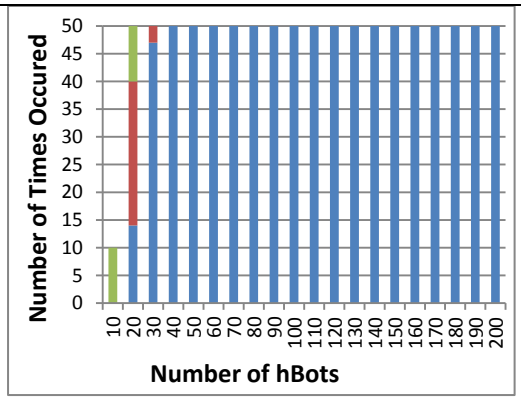
Triangle Valid, 0.0 Probability of Movement



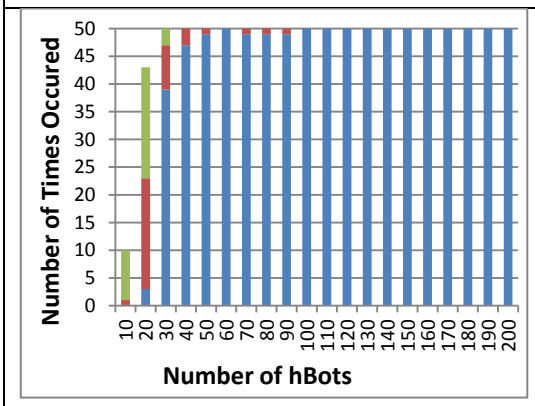
Triangle Valid, 0.1 Probability of Movement



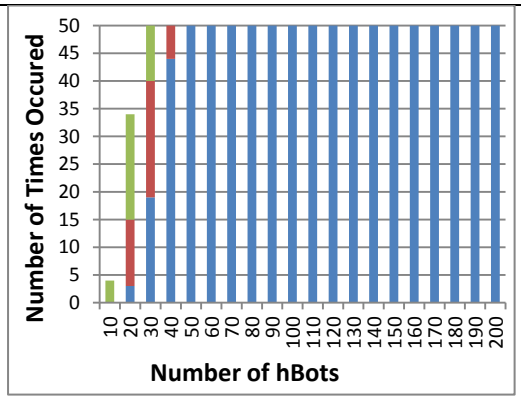
Triangle Valid, 0.2 Probability of Movement



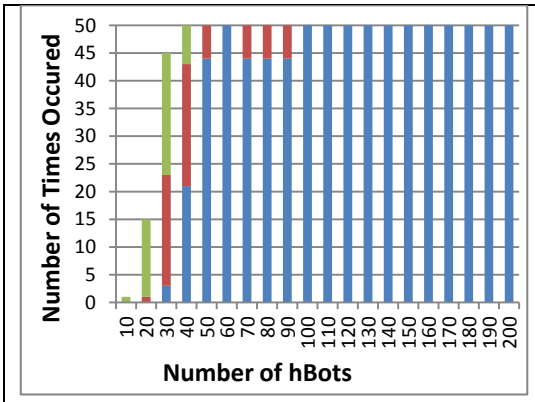
Triangle Valid, 0.3 Probability of Movement



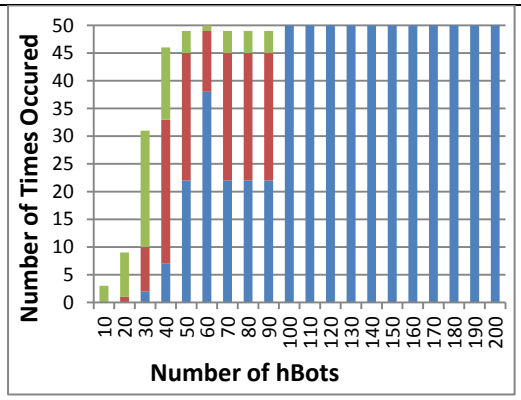
Triangle Valid, 0.4 Probability of Movement



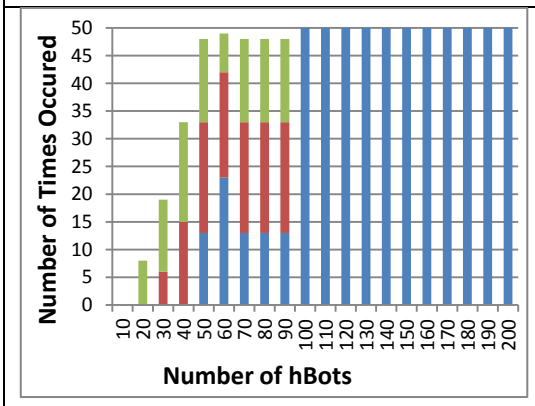
Triangle Valid, 0.5 Probability of Movement



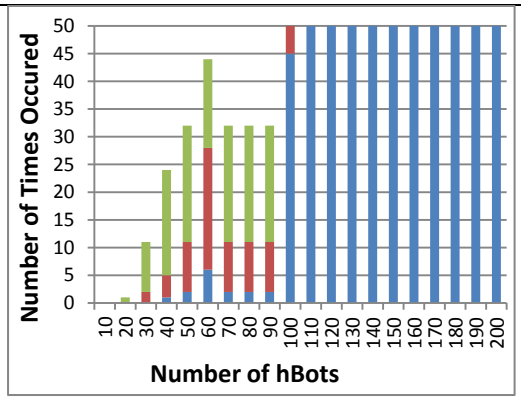
Triangle Valid, 0.6 Probability of Movement



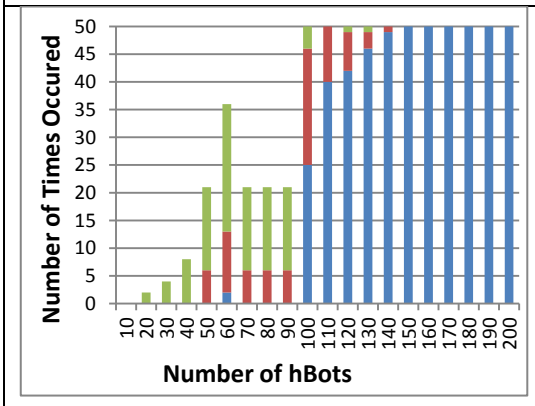
Triangle Valid, 0.7 Probability of Movement



Triangle Valid, 0.8 Probability of Movement

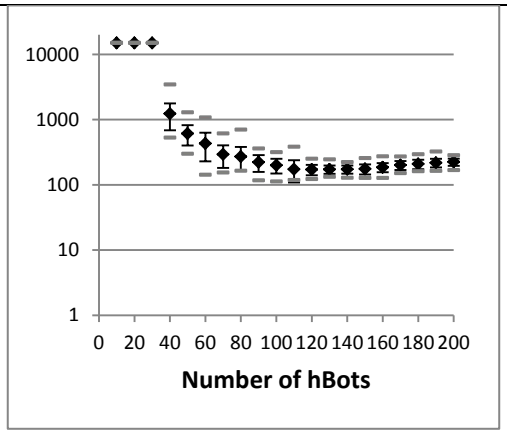


Triangle Valid, 0.9 Probability of Movement

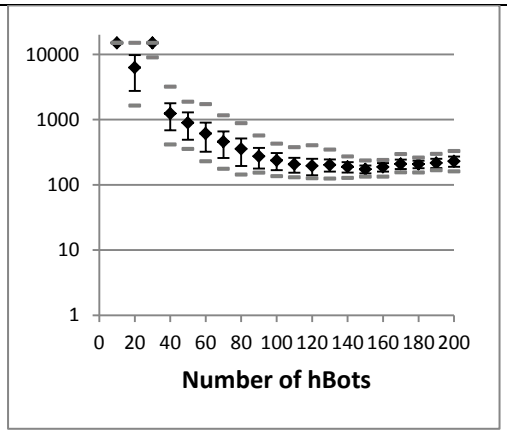


Triangle Valid, 1.0 Probability of Movement

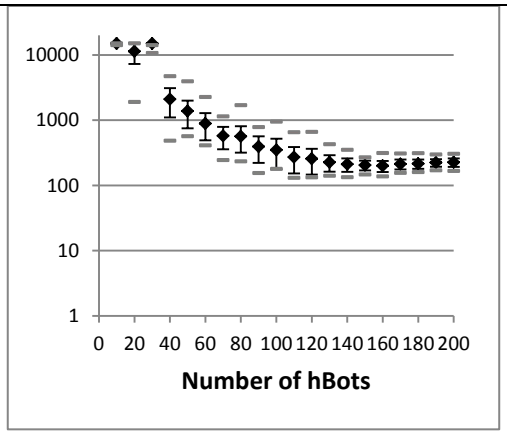
B.3: Time-steps Required to Remove Three Valid Hexagonal Object Shapes



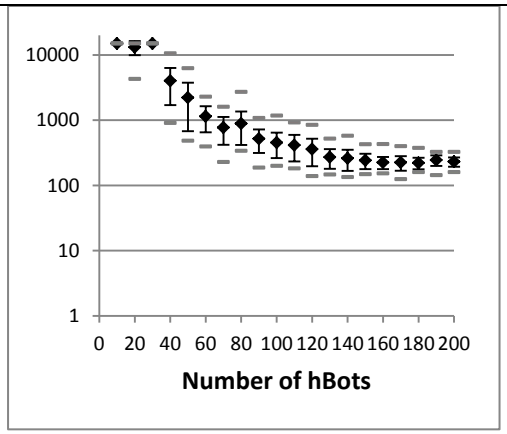
Hexagon Valid, 0.0 Probability of Movement



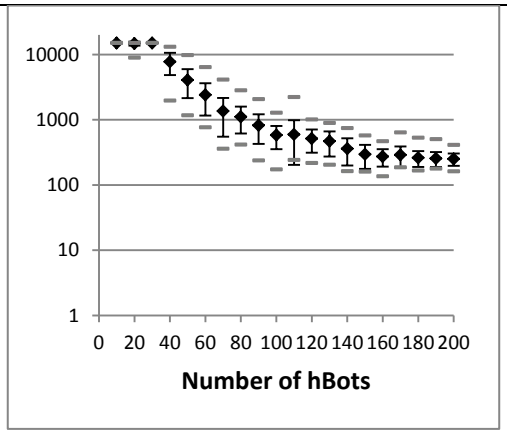
Hexagon Valid, 0.1 Probability of Movement



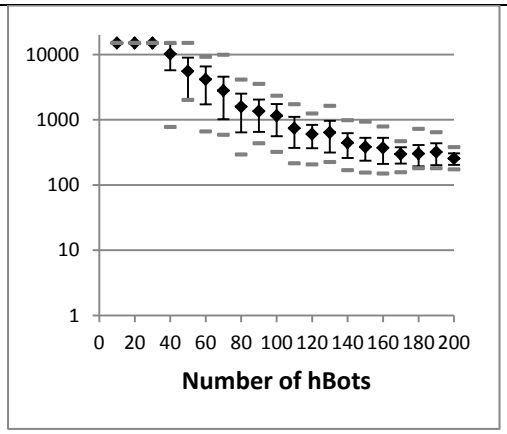
Hexagon Valid, 0.2 Probability of Movement



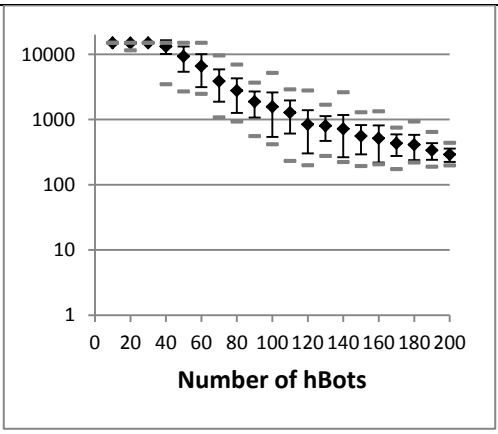
Hexagon Valid, 0.3 Probability of Movement



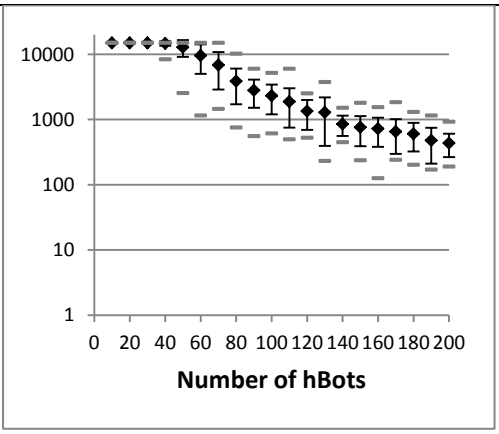
Hexagon Valid, 0.4 Probability of Movement



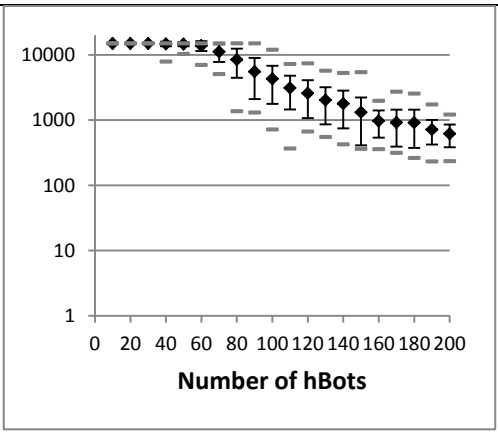
Hexagon Valid, 0.5 Probability of Movement



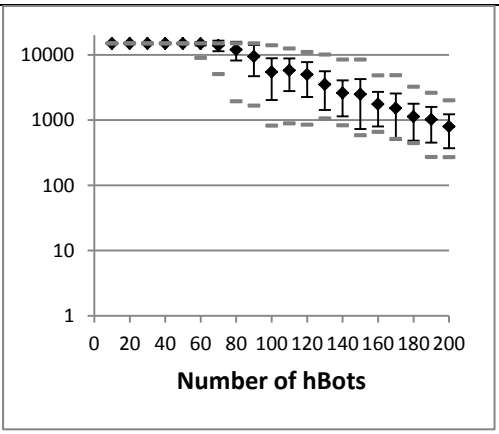
Hexagon Valid, 0.6 Probability of Movement



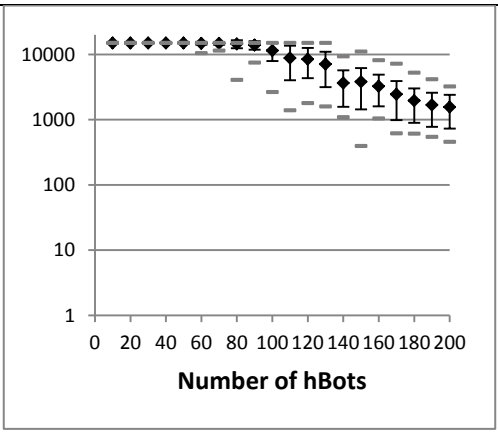
Hexagon Valid, 0.7 Probability of Movement



Hexagon Valid, 0.8 Probability of Movement

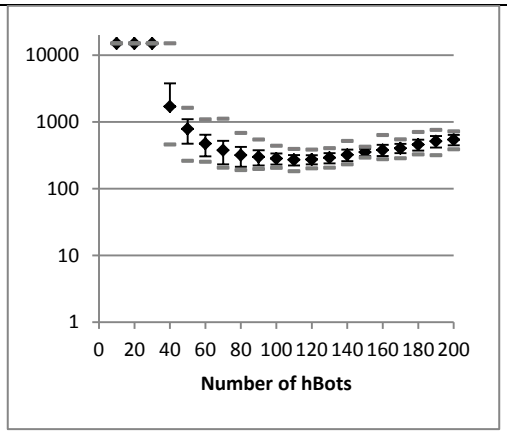


Hexagon Valid, 0.9 Probability of Movement

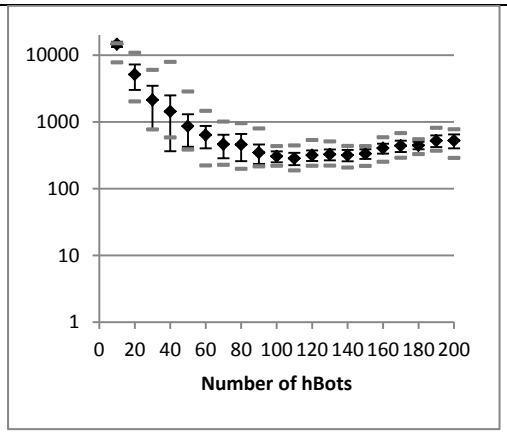


Hexagon Valid, 1.0 Probability of Movement

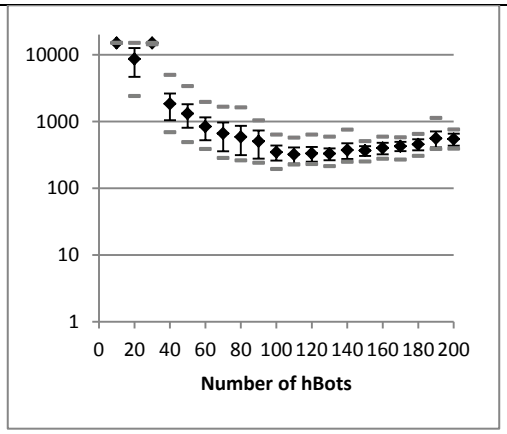
B.4: Time-steps Required to Remove Three Valid Triangular Object Shapes



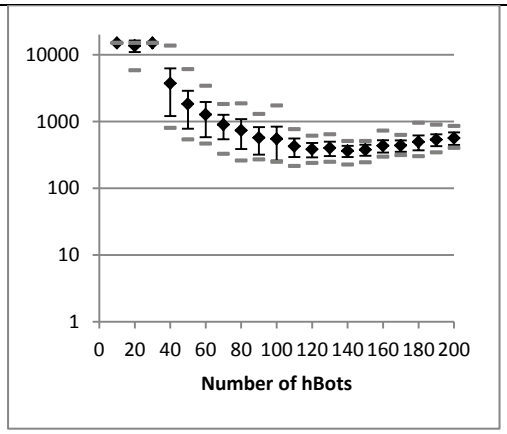
Triangular Valid, 0.0 Probability of Movement



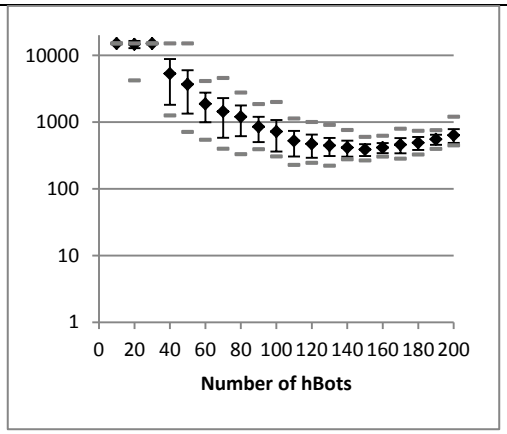
Triangular Valid, 0.1 Probability of Movement



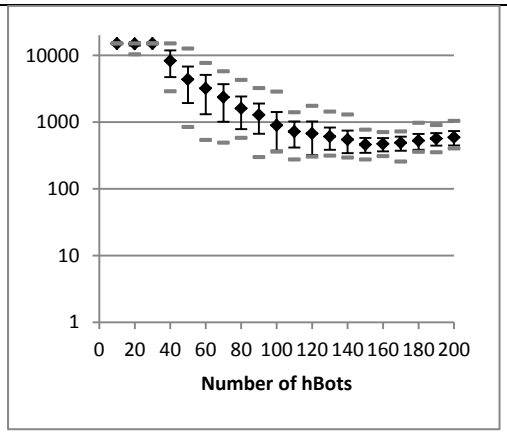
Triangular Valid, 0.2 Probability of Movement



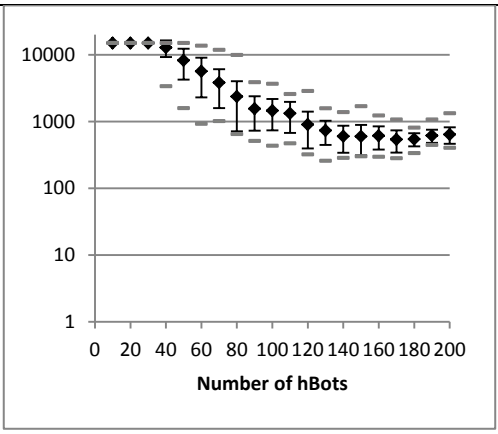
Triangular Valid, 0.3 Probability of Movement



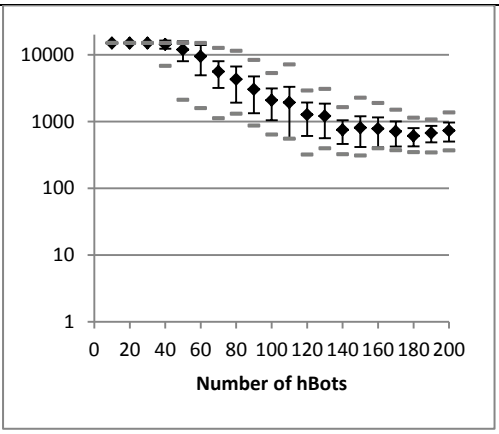
Triangular Valid, 0.4 Probability of Movement



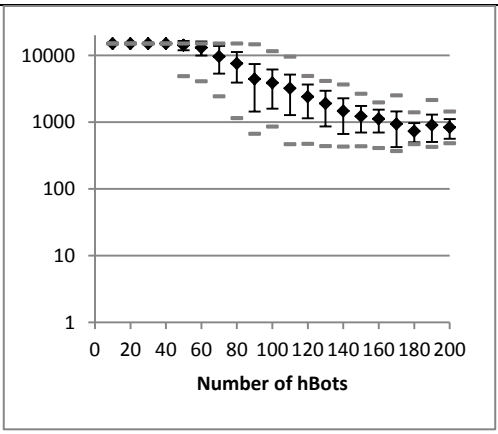
Triangular Valid, 0.5 Probability of Movement



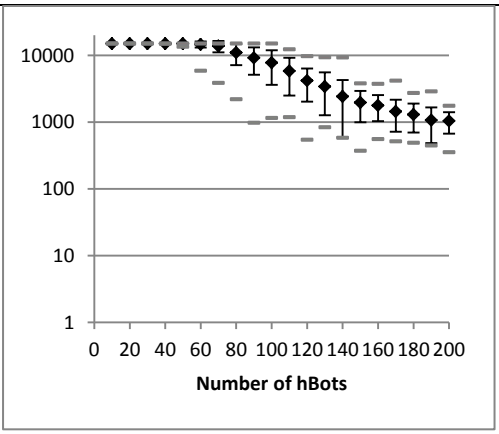
Triangular Valid, 0.6 Probability of Movement



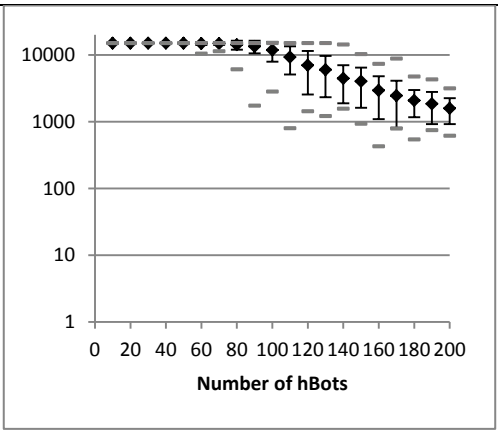
Triangular Valid, 0.7 Probability of Movement



Triangular Valid, 0.8 Probability of Movement

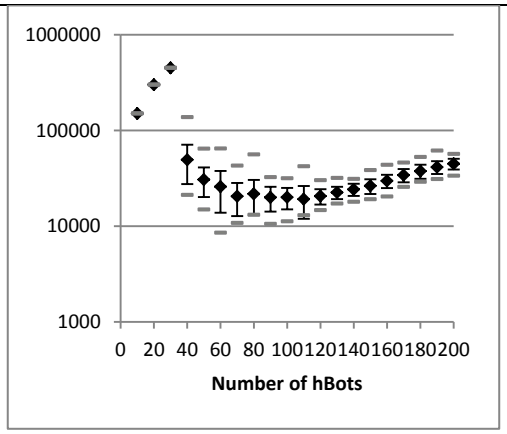


Triangular Valid, 0.9 Probability of Movement

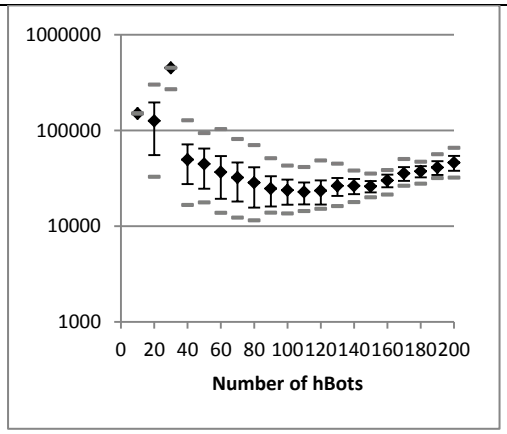


Triangular Valid, 1.0 Probability of Movement

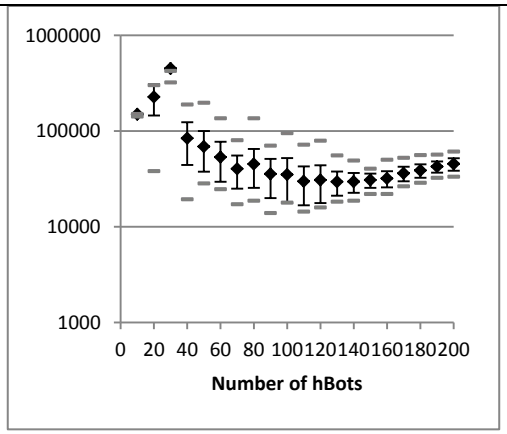
B.5: Energy Consumed to Remove Three Valid Hexagonal Object Shapes



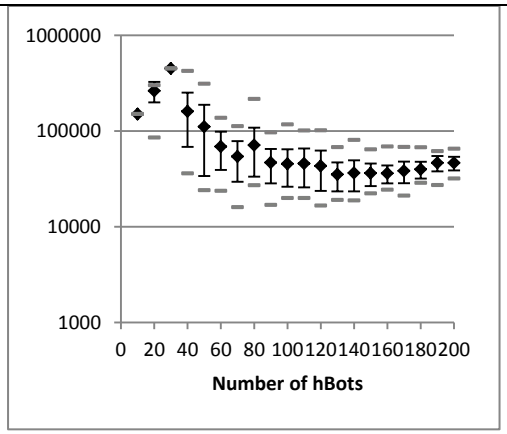
Hexagonal Valid, 0.0 Probability of Movement



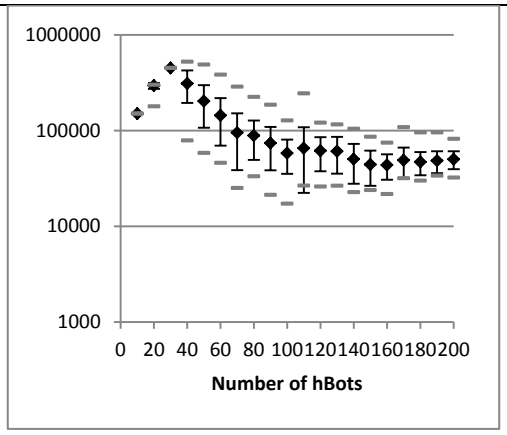
Hexagonal Valid, 0.1 Probability of Movement



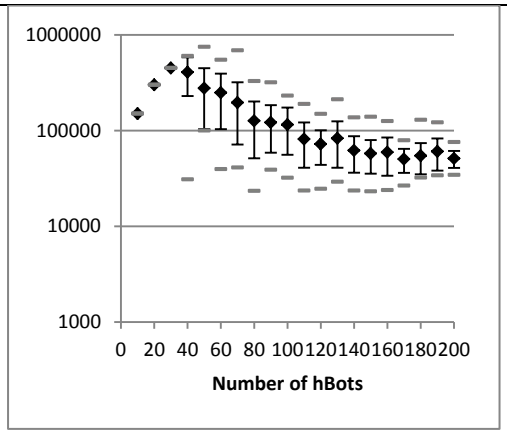
Hexagonal Valid, 0.2 Probability of Movement



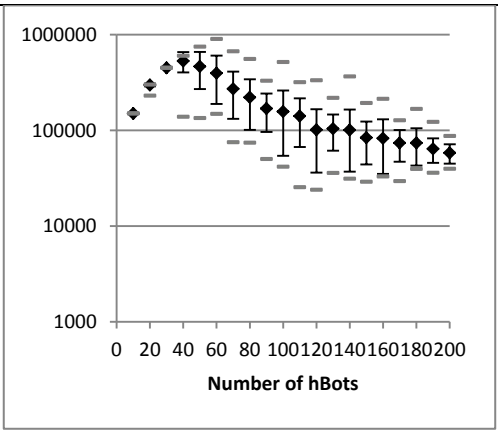
Hexagonal Valid, 0.3 Probability of Movement



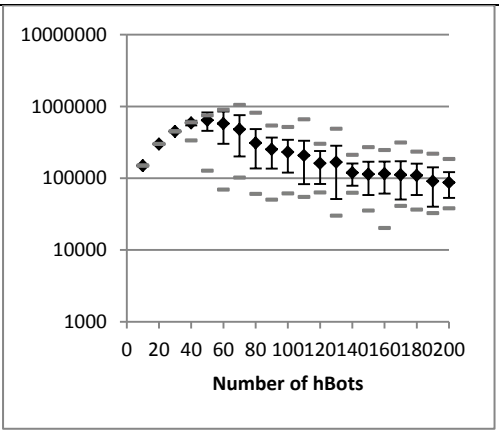
Hexagonal Valid, 0.4 Probability of Movement



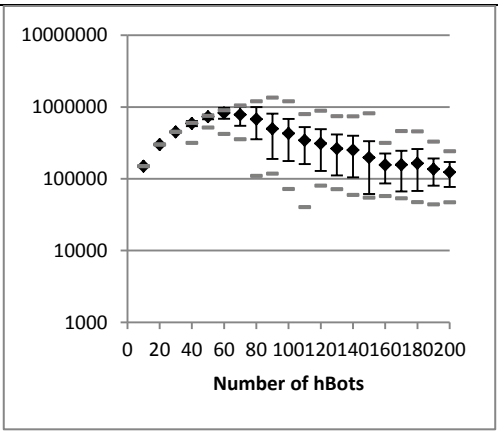
Hexagonal Valid, 0.5 Probability of Movement



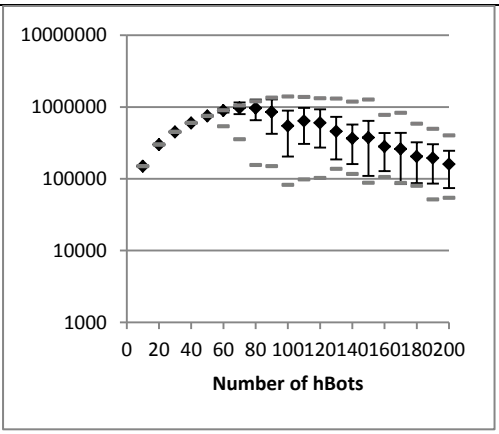
Hexagonal Valid, 0.6 Probability of Movement



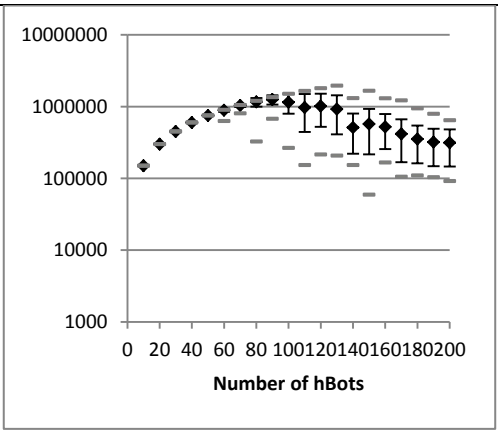
Hexagonal Valid, 0.7 Probability of Movement



Hexagonal Valid, 0.8 Probability of Movement

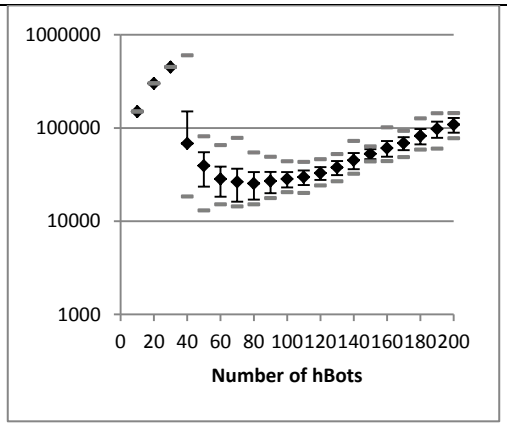


Hexagonal Valid, 0.9 Probability of Movement

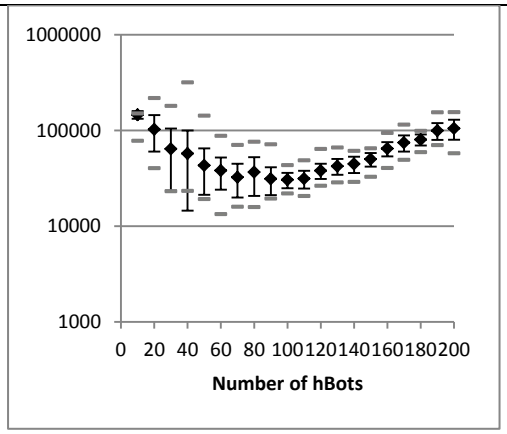


Hexagonal Valid, 1.0 Probability of Movement

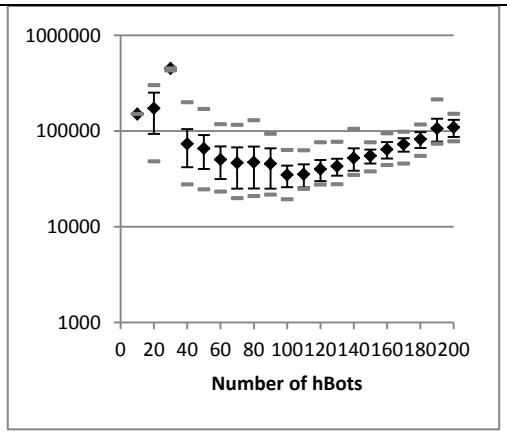
B.6: Energy Consumed to Remove Three Valid Triangular Object Shapes



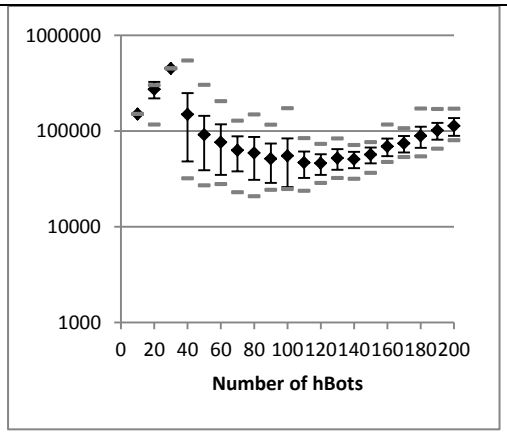
Triangular Valid, 0.0 Probability of Movement



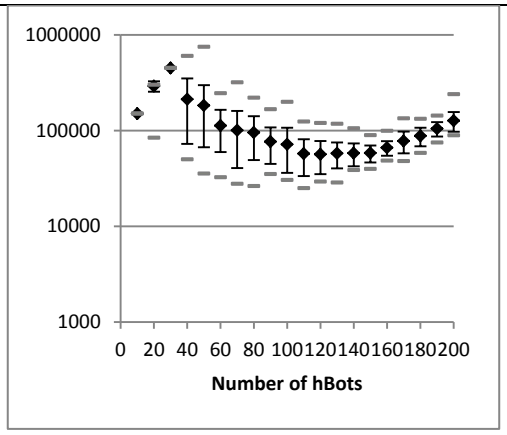
Triangular Valid, 0.1 Probability of Movement



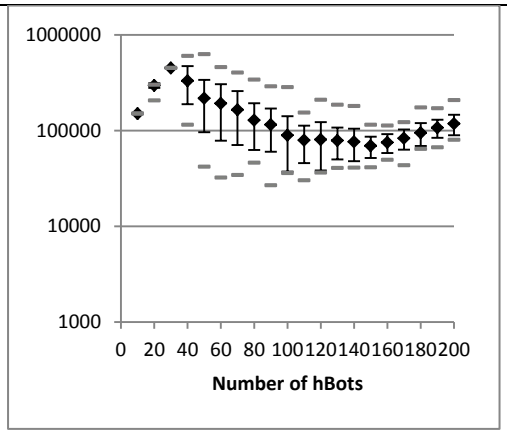
Triangular Valid, 0.2 Probability of Movement



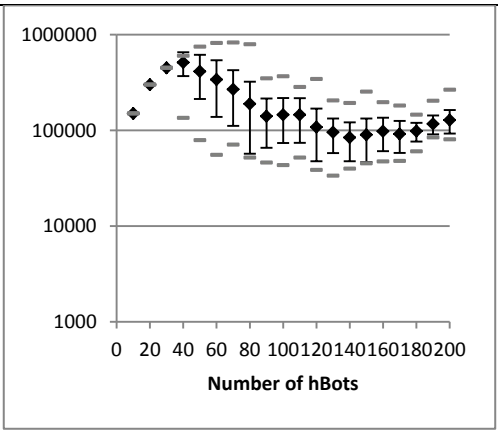
Triangular Valid, 0.3 Probability of Movement



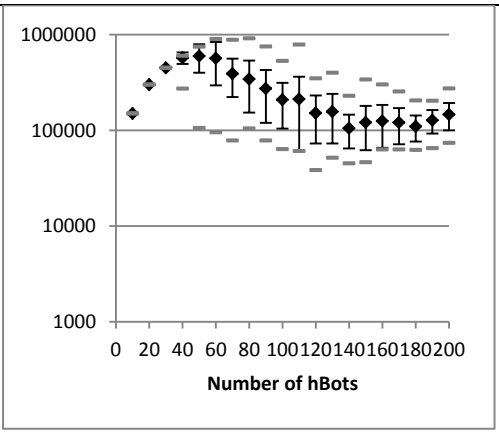
Triangular Valid, 0.4 Probability of Movement



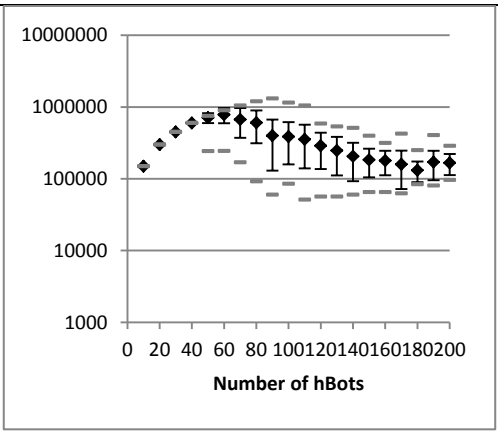
Triangular Valid, 0.5 Probability of Movement



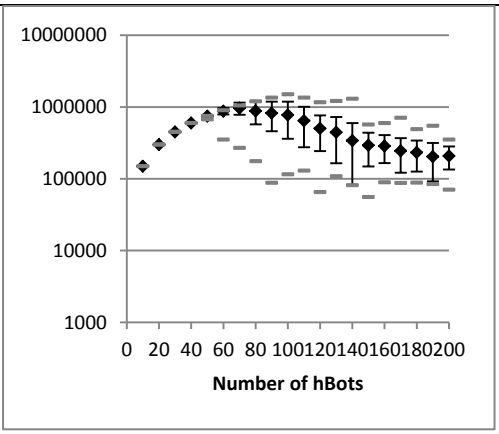
Triangular Valid, 0.6 Probability of Movement



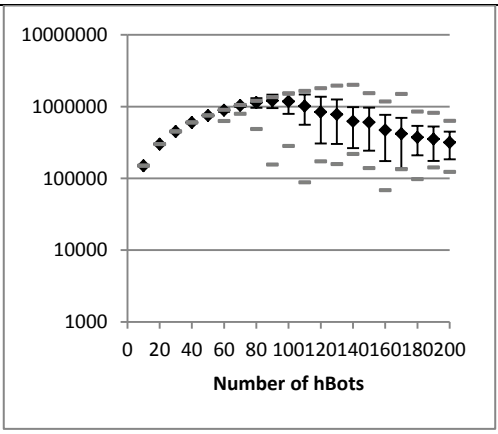
Triangular Valid, 0.7 Probability of Movement



Triangular Valid, 0.8 Probability of Movement



Triangular Valid, 0.9 Probability of Movement



Triangular Valid, 1.0 Probability of Movement

Appendix C: Object Shape Creator and Data-Chain Inspector

This program, written in Processing, is used to create a series of object shapes starting with a single cell and an increasing number of cells, tried in multiple combinations. Data-Chains for each of the object shapes are created to determine if they are simple or complex object shapes.

```
////////////////////////////////////////////////
// Creates Object Shapes and Data-Chains      //
////////////////////////////////////////////////

PrintWriter output;

// runComparison
int maxCells = 10;

boolean showDisplay = false;

// Cell variables
Cell[][] cell;
int cols = 31;//objectsPerRow + ((objectsPerRow+1)*gapsize);//21;
int rows = 31;//objectsPerRow + ((objectsPerRow+1)*gapsize);//21;
float hexWidth = 15;
int cX = ((cols-1)/2);
int cY = ((rows-1)/2);

// types of shapes
int[] numberOfEachShape = new int[30];
boolean checkForHollow = true;

// general
int locationsX[]; // stores the x-coordinates of the spiral location value relative to the centre cell
int locationsY[]; // stores the y-coordinates of the spiral location value relative to the centre cell
int noOfRings = 12;
int lowestCellInRing[]; // stores to the lowest spiral location value for each ring
int falseCounter = 0; // counts the times that a false result is returned (to many falses force an update of
previous object cell)
int displayColour = 0;

ObjectShape[] objectShape;
int currentCell = 0;
int globalID = 0;
int currentRing = 0;

TempShape tempShape;

boolean newBaseObjShape = false;
boolean firstNewObject = true;
int baseID = 0;

boolean updateHighestCellPosition = false;

boolean halt = false;

int fileNumber = 1;
int limitID = 1000;

void setup(){
  output = createWriter("DataChainsCheckedPart" + fileNumber + ".txt");
  output.println("Object Shape ID; Number of Cells; Length of Data-Chain; Base Shape ID; Data-Chain;Number of
Branches;Number of valid Rep-Chains;Rep-Chains;Simple or Complex Shape;Required Visits to First Empty Cell in
forming data-chain");

  size(arenaWidth(),arenaHeight()); // function is in Cell
  background(0);

  // create cells (x,y)
  cell = new Cell[cols][rows];
  for (int i = 0; i < cols; i++) {
    for (int j = 0; j < rows; j++) {
      cell[i][j] = new Cell(i,j);
    }
  }

  // calculate relative spiral locations
  locationsX = new int[1000];
  locationsY = new int[1000];
  findXSpiralLocations();
  findYSpiralLocations();

  locationsX[0] = ((cols-1)/2); // x coordinate of 1st cell
  locationsY[0] = ((rows-1)/2); // y coordinate of 1st cell

  // calculate lowest cell number in each ring
  lowestCellInRing = new int[noOfRings];
  lowestCellInRing[0] = 0;
  for (int i = 1; i < lowestCellInRing.length; i++){
    lowestCellInRing[i] = (3*i*i) - (3*i) + 1;
  }
```

```
  // create objectShape
  objectShape = new ObjectShape[100000];
  objectShape[0] = new ObjectShape(0,0);

  tempShape = new TempShape(globalID,currentCell);

  // fill numberOfEachShape array
  for (int i = 0; i < numberOfEachShape.length; i++){
    numberOfEachShape[i] = 0;
  }

}

void draw(){

  if(globalID == limitID){
    limitID += 10000;
    fileNumber++;
    output.flush(); // Writes the remaining data to the file
    output.close(); // Finishes the file
    output = createWriter("DataChainsCheckedPart" + fileNumber + ".txt");
  }

  if(halt){
    //delay(250);
    halt = false;
  }

  refreshCells();
  tempShape.markCells(0,currentCell);


  // is object valid
  if (tempShape.validObject(currentCell)){
    ////////////////////////////////////////////////////////////////////////////// VALID OBJECT
    falseCounter = 0;

    refreshCells();
    tempShape.markCells(0,currentCell);
    objectCode();// create objectCode

    // does shape already exist
    if (tempShape.compare()){
      // update highest cell position
      updateHighestCellPosition = true;

      // display
      displayTheCells(3); // orange (exising shape)
    }
    else{
      // Display
      displayTheCells(1); // green (valid shape)
      halt = true;

      ////////////////////////////////////////////////////////////////////// SAVE SHAPE
      // save shape to object shapes
      objectShape[globalID] = new ObjectShape(globalID, currentCell);
      objectShape[globalID].copyTempShape();

      /////////////// check the data-chain
      objectShape[globalID].checkDataChain();

      // record number of shapes for each cell allowance
      numberOfEachShape[currentCell+1]++;

      // increase globalID
      globalID++;

      // update base shape and reset cell position
      tempShape = new TempShape(globalID,currentCell);
      tempShape.copyObjectShape(baseID);
      tempShape.ID = globalID;
      tempShape.numberOfCells = currentCell+1;

      tempShape.locations[currentCell] = 0;

      tempShape.locations[currentCell] = objectShape[globalID-1].locations[currentCell]+1;

    }

  }
  ////////////////////////////////////////////////////////////////////////// INVALID OBJECT
  else{
    // update highest cell position
    updateHighestCellPosition = true;

    displayTheCells(2);  // red (invalid shape)
  }

  // update highest cell position
  if (updateHighestCellPosition){
    updateHighestCellPosition = false;

    if (falseCounter == (lowestCellInRing[currentRing+1] - lowestCellInRing[currentRing])){
      // load new base object shape
      newBaseObjShape = true;
      falseCounter = 0;
    }

    tempShape.locations[currentCell]++;

    // update currentRing
```

```
    for (int i = 0; i < noOfRings; i++){
      if(tempShape.locations[currentCell] >= lowestCellInRing[i]){
        currentRing = i;
      }
    }

    // check current ring number, if move to next ring reset false counter
    if (tempShape.locations[currentCell] == lowestCellInRing[currentRing]){ //lowestCellInRing[currentRing + 1]
      falseCounter = 0;
    }

    falseCounter++;
  }

  ////////////////////////////////////////////////////////////////////////////// CREATE NEW BS
  if(newBaseObjShape){
    if (!firstNewObject){
      baseID++;
    }
    else{
      firstNewObject = false;
    }

    newBaseObjShape = false;
    tempShape.copyObjectShape(baseID);
    tempShape.numberOfCells++;
    currentCell = tempShape.numberOfCells-1;
    tempShape.locations[currentCell] = 0;
    tempShape.ID = globalID;

  }

  // compare the shapes
  if (currentCell == maxCells){
    println("FLUSH with currentCell: " + currentCell);
    output.flush(); // Writes the remaining data to the file
    output.close(); // Finishes the file
    exit();
  }
}

void refreshCells(){
  // REFRESH CELLS
  for (int i = 0; i < cols; i++) {
    for (int j = 0; j < rows; j++) {
        cell[i][j].cellState = -1;
    }
  }
}

void displayTheCells(int displayColour){
  if(showDisplay){
    refreshCells();
    tempShape.markCells(displayColour,currentCell);

    // DISPLAY GRID OF CELLS
    for (int i = 0; i < cols; i++) {
      for (int j = 0; j < rows; j++) {
        if (cell[i][j] != null)
          cell[i][j].display(j);
      }
    }


    refreshCells();
    tempShape.markCells(0,currentCell);
  }
}
////////////////////////////////////////////////////////////////////////////////
// A Cell Object ///////////////////////////////////////////////////////////////
class Cell {
  // A cell object knows about its location in the grid
  float x,y; //x,y location of centre point
  //color colour = cWhite;
  int cellState = -1;
  float hexHeight = 4 * (0.5 * hexWidth) * (tan((radians(30))));

  // Cell Constructor
  Cell(float tempX, float tempY){
    x = (tempX*hexWidth)+2;
    y = (tempY*hexHeight*0.75)+1;
  }

  void display(int j) {
    rectMode(CENTER);
    translate(0.5*hexWidth,0.5*hexHeight); //start with first cell fully in window

    noStroke();
    strokeWeight(2);
    fill(colourFromNumber(cellState)); // change colour based on cell state

    translate(j*0.5*hexWidth,0); // shift right for each new line
    //ellipse(x,y,hexWidth,hexWidth);

    translate(x,y);
    for (int i = 0; i < 3; i++) {
      rect(0,0,hexWidth,0.5*hexHeight); // draw rectangle
      stroke(0);
      line(-0.5*hexWidth,-0.25*hexHeight,-0.5*hexWidth,0.25*hexHeight);
      line(0.5*hexWidth,-0.25*hexHeight,0.5*hexWidth,0.25*hexHeight);
      noStroke();
      rotate(PI/3);
```

```
    }

    rotate(PI); // reset rotation
    translate(-x,-y); //reset centre co-ordinate

    translate(-j*0.5*hexWidth,0); //reset centre co-ordinate

    translate(-0.5*hexWidth,-0.5*hexHeight); //reset centre co-ordinate

  }
}

// which states are which colours
int colourFromNumber(int tempNumber){
  // Colours
  color cRed = color(255,0,0);
  color cOrange = color(255,165,0);
  color cGreen = color(50,205,50);

  color cWhite = color(255,255,255); // empty cell
  color cBlack = color(0,0,0);

  // Colours
  if (tempNumber == 1)  // valid and new
    return(cGreen);
  else if (tempNumber == 2) // invalid
    return(cRed);
  else if (tempNumber == 3) // valid but repeats
    return(cOrange);

  else if (tempNumber == -1)
    return(cWhite);

  else
    return(cBlack);
}

// calc arena size based on size and number of cells
int arenaWidth(){
  return((int)(1.5*hexWidth*(cols-0.2)));
}
int arenaHeight(){
  return((int)(0.75*4*(0.5*hexWidth)*(tan((radians(30))))*(rows+0.5)));
}



class ObjectShape{
  int[] objectCode = new int[50];
  int codeLength;
  int ID;
  int numberOfCells;
  int[] locations;

  int visitsToFirstEmpty = 0;

  ObjectShape(int tempID, int tempNumberOfCells){
    for (int i=0; i < objectCode.length; i++){
      objectCode[i] = 0;
    }

    numberOfCells = tempNumberOfCells+1;
    ID = tempID;

    locations = new int[20];

    for (int i = 0; i < locations.length; i++){
      locations[i] = -100;
    }

    locations[0] = 0;
  }

  // copy temp shape data
  void copyTempShape(){
    ID = tempShape.ID;
    numberOfCells = tempShape.numberOfCells;
    codeLength = tempShape.codeLength;
    visitsToFirstEmpty = tempShape.visitsToFirstEmpty;

    for(int i = 0; i < objectCode.length; i++){
      objectCode[i] = tempShape.objectCode[i];
    }

    for(int i = 0; i < locations.length; i++){
      locations[i] = tempShape.locations[i];
    }
  }

  void checkDataChain(){
    dataChainChecker();
  }
}
// given an initial centre co-ordinate of the 1st object cell
// find the x,y coordinate for each spiral location
// spiral location starts with 0 in the centre and spirals out increasing in value

void findXSpiralLocations(){
  int xTrack = cX;//((cols-1)/2); // relative x-coordinate based on current spiral location value
  int addJ = 0;

  for (int i = 1; i <= noOfRings; i++){ // i is ring number
```

```
    //boolean xAA;  // first value +2 rather than +1
    int xA = i;       // number of times to +1 to x-tracker
    int xB = i;   // number of times to +0 to x-tracker
    int xC = 2*i; // number of times to -1 from x-tracker
    int xD = i;   // number of times to +0 to x-tracker
    int xE = i;       // number of times to +1 to x-tracker

    for (int j = 1; j <= xA; j++){
      xTrack = xTrack+1;
      locationsX[j + addJ] = xTrack;
    }

    for (int j = xA+1; j <= xA + xB; j++){
      xTrack = xTrack;
      locationsX[j + addJ] = xTrack;
    }

    for (int j = xA + xB + 1; j <= xA + xB + xC; j++){
      xTrack = xTrack-1;
      locationsX[j + addJ] = xTrack;
    }

    for (int j = xA + xB + xC + 1; j <= xA + xB + xC + xD; j++){
      xTrack = xTrack;
      locationsX[j + addJ] = xTrack;
    }

    for (int j = xA + xB + xC + xD + 1; j <= xA + xB + xC + xD + xE; j++){
      xTrack = xTrack+1;
      locationsX[j + addJ] = xTrack;
    }

    addJ = xA + xB + xC + xD + xE + addJ;
  }
}

void findYSpiralLocations(){
  int yTrack = cY; //((rows-1)/2); // relative y-coordinate based on current spiral location value

  int addJ = 0;

  for (int i = 1; i <= noOfRings; i++){ // ring number
    int yA = 1;    // number of times to -1 from y-tracker
    int yB = i-1;        // number of times to +0 to y-tracker
    int yC = i*2;  // number of times to +1 to y-tracker
    int yD = i;    // number of times to +0 to y-tracker
    int yE = i*2;  // number of times to -1 from y-tracker

    for (int j = 1; j <= yA; j++){
      yTrack = yTrack-1; // -1 to tracker yA times
      locationsY[j + addJ] = yTrack;
    }

    for (int j = yA+1; j <= yA + yB; j++){
      yTrack = yTrack; // +0 to tracker yB times
      locationsY[j + addJ] = yTrack;
    }

    for (int j = yA + yB + 1; j <= yA + yB + yC; j++){
      yTrack = yTrack+1; // +1 to tracker yC times
      locationsY[j + addJ] = yTrack;
    }

    for (int j = yA + yB + yC + 1; j <= yA + yB + yC + yD; j++){
      yTrack = yTrack;    // +0 to tracker yD times
      locationsY[j + addJ] = yTrack;
    }

    for (int j = yA + yB + yC + yD + 1; j <= yA + yB + yC + yD + yE; j++){
      yTrack = yTrack-1;  // -1 from tracker yE times
      locationsY[j + addJ] = yTrack;
    }

    addJ = yA + yB + yC + yD + yE + addJ;
  }
}



void dataChainChecker(){
  DataChain dataChain = new DataChain();

  do{
    dataChain.checkDataChain();
  }while (dataChain.stop == false);

}

class DataChain{
  Sets[] sets;
  int maxSets = 1;
  int totalSets = 1;
  int currentSet;

  int[] dataChainNumbers;

  int lengthOfChain;
  int currentLink = 0;
  int numberWaiting = 0;
  int numberFound = 0;

  // Output information
```

```
  int noOfValidRepChains = 0;
  int[][] repChain;

  boolean stop = false;

  DataChain(){
    dataChainNumbers = new int[objectShape[globalID].codeLength];

    for(int i = 0; i < dataChainNumbers.length; i++){
      dataChainNumbers[i] = objectShape[globalID].objectCode[i];
    }

    orderDataChain();

    for(int i = 0; i < dataChainNumbers.length; i++){
      if(dataChainNumbers[i] == 2){
        maxSets++;
        totalSets *= 2;
      }
      if(dataChainNumbers[i] == 3){
        maxSets += 2;
        totalSets *= 3;
      }
      if(dataChainNumbers[i] == 4){
        maxSets += 3;
        totalSets *= 4;
      }
    }

    println("Max sets: " + maxSets + "   Total sets: " + totalSets + "    Number of Links: " +
dataChainNumbers.length);

    lengthOfChain = dataChainNumbers.length;

    sets = new Sets[maxSets];
    sets[maxSets-1] = new Sets(false, 0, 0, lengthOfChain); // boolean additional, int interpretedValue, int
copyThis, int lengthOfChain
    currentSet = maxSets-1;

    repChain = new int[10][lengthOfChain];

  }

  ////////////////////////////////////////////////////////////////////////////
  void checkDataChain(){
    int valueOfLink = dataChainNumbers[currentLink];

    if(valueOfLink == 1){
      sets[currentSet].positiveMove();

      sets[currentSet].believedDataChain[currentLink] = 1;
    }
    else if (valueOfLink == 2){
      // new potential set (1/1)
      sets[numberWaiting++] = new Sets(true,1,currentSet,lengthOfChain);

      sets[currentSet].negativeMove();
      sets[currentSet].positiveMove();

      sets[currentSet].believedDataChain[currentLink] = 2;
    }
    else if (valueOfLink == 3){
      // new potential set (1/1/1) and (1/2)
      sets[numberWaiting++] = new Sets(true,1,currentSet,lengthOfChain);

      // new potential set (2/1)
      sets[numberWaiting++] = new Sets(true,2,currentSet,lengthOfChain);

      sets[currentSet].negativeMove();
      sets[currentSet].negativeMove();
      sets[currentSet].positiveMove();

      sets[currentSet].believedDataChain[currentLink] = 3;
    }
    else if (valueOfLink == 4){
      // new potnetial set (2/2)
      sets[numberWaiting++] = new Sets(true,2,currentSet,lengthOfChain);

      // new potential set (1/3)
      sets[numberWaiting++] = new Sets(true,1,currentSet,lengthOfChain);

      // new potential set (3/1)
      sets[numberWaiting++] = new Sets(true,3,currentSet,lengthOfChain);

      sets[currentSet].negativeMove();
      sets[currentSet].negativeMove();
      sets[currentSet].negativeMove();
      sets[currentSet].positiveMove();

      sets[currentSet].believedDataChain[currentLink] = 4;
    }
    else if (valueOfLink == 5){
      sets[currentSet].negativeMove();
      sets[currentSet].negativeMove();
      sets[currentSet].negativeMove();
      sets[currentSet].negativeMove();
      sets[currentSet].positiveMove();

      sets[currentSet].believedDataChain[currentLink] = 5;
    }
    else {
      // invalid data-Chain
```

```
      println("ERROR dataChainChecker");
      exit();
    }

    currentLink++;

    // sets[currentSet] formed
    if(currentLink == dataChainNumbers.length){
      //
      sets[currentSet].printBelievedDataChain();

      // calc convexity value
      if(sets[currentSet].calculateConvexityValue() == 6){
        // check for repeating points (excluding last point)
        if(sets[currentSet].areThereRepeatPoints() == false){
          // calc if returns to start [0,0,0]
          if(sets[currentSet].checkForReturnToStart() == true){
            // VALID DATA-CHAIN
            println("VALID DATA-CHAIN");

            for(int i = 0; i < lengthOfChain; i++){
              repChain[noOfValidRepChains][i] = sets[currentSet].believedDataChain[i];
            }

            noOfValidRepChains++;
          }
          else{
            // invalid data-chain as does not return to start
            //println("INVALID: does not return to start");
          }
        }
        else{
          // invalid data-chain due to repeating points on trace
          //println("INVALID: repeating points on trace");
        }
      }
      else{
        // invalid data-chain due to convexity
        //println("INVALID: convexity != 6");
      }

      numberFound++;

      if(numberWaiting - 1 < 0){
        println("FINISHED, numberFound: " + numberFound);

        println("Number of valid rep-chains " + noOfValidRepChains);

        output.print(globalID + ";" + baseID + ";" + objectShape[globalID].numberOfCells + ";" + lengthOfChain +
";");

        output.print("{");
        for(int i = 0; i < dataChainNumbers.length-1; i++){
          output.print(dataChainNumbers[i] + ",");
        }
        output.print(dataChainNumbers[dataChainNumbers.length-1] +"};");

        output.print(numberFound + ";" + noOfValidRepChains + ";");

        for(int i = 0; i < noOfValidRepChains; i++){
          output.print("{");
          for(int j = 0; j < lengthOfChain-1; j++){
            output.print(repChain[i][j] + ",");
          }
          output.print(repChain[i][lengthOfChain-1] +"}");
        }

        output.print(";");

        boolean differentChain = false;
        for(int i = 0; i < dataChainNumbers.length; i++){
          if(dataChainNumbers[i] != repChain[0][i]){
            differentChain = true;
          }
        }
        if(differentChain){
          output.print("Complex;");
        }
        else{
          output.print("Simple;");
        }

        output.println(objectShape[globalID].visitsToFirstEmpty);

        stop = true;
      }
      else {
        //println("                              shift");

        // shift previous set to last slot
        sets[currentSet].currentDirection = sets[numberWaiting-1].currentDirection;
        sets[currentSet].linkWhenMade = sets[numberWaiting-1].linkWhenMade;
        sets[currentSet].currentCheck = sets[numberWaiting-1].currentCheck;

        for(int i = 0; i < sets[currentSet].totalNumbersOfDataChain; i++){
          sets[currentSet].aCheck[i] = sets[numberWaiting-1].aCheck[i];
          sets[currentSet].bCheck[i] = sets[numberWaiting-1].bCheck[i];
          sets[currentSet].cCheck[i] = sets[numberWaiting-1].cCheck[i];
        }

        for(int i = 0; i < 6; i++){
          sets[currentSet].directionValues[i] = sets[numberWaiting-1].directionValues[i];
```

```
        }

        for(int i = 0; i < dataChainNumbers.length; i++){
          sets[currentSet].believedDataChain[i] = sets[numberWaiting-1].believedDataChain[i];
        }

        numberWaiting = numberWaiting - 1;

        sets[currentSet].isValid = false;
        sets[currentSet].isTested = false;

        currentLink = sets[currentSet].linkWhenMade+1;
        //println(numberWaiting);
      }

    }
  }

  ////////////////////////////////////////////////////////////////////////////
  void orderDataChain(){
    // does dataChain have at least one 1
    int noOfOnes = 0;
    int lastOne = dataChainNumbers.length;

    for(int i = 0; i < dataChainNumbers.length; i++){
      if(dataChainNumbers[i] == 1){
        noOfOnes++;
        lastOne = i;
      }
    }


    if(noOfOnes < 1){ /// could this be 6?
      print("invalid DataChain");

      print("{");
      for(int i = 0; i < dataChainNumbers.length-1; i++){
        print(dataChainNumbers[i] + ",");
      }
      println(dataChainNumbers[dataChainNumbers.length-1] +"};");

      println("Obect Shape Infomation");
      println("datachainLength " + objectShape[globalID].codeLength);
      println("GlobalID" + globalID);
      println("BaseShape " + baseID);

      print("{");
      for(int i = 0; i < dataChainNumbers.length; i++){
        print( objectShape[globalID].objectCode[i] + ",");
      }
      println("}");

      for(int i = 0; i < objectShape[globalID].locations.length; i++){
        print( objectShape[globalID].locations[i] + ",");
      }

      exit();
    }
    else if(dataChainNumbers[dataChainNumbers.length-1] != 1){
      println("Reorder dataChain to end with a 1");
      // convertDataChain so there is a one at the end
      int[] tempChain = new int[dataChainNumbers.length];
      int shift = (dataChainNumbers.length-1) - lastOne;
      int newPos;

      for(int i = 0; i < dataChainNumbers.length; i++){
        if(shift + i > dataChainNumbers.length-1){
          newPos = shift + i - (dataChainNumbers.length);
        }
        else{
          newPos = shift + i;
        }

        tempChain[newPos] = dataChainNumbers[i];

      }


      print("Reordered dataChain: {");
      for(int i = 0; i < dataChainNumbers.length-1; i++) {
        print(tempChain[i] + ", ");
      }
      println(tempChain[dataChainNumbers.length-1] + "}");



      for(int i = 0; i < dataChainNumbers.length; i++){
        dataChainNumbers[i] = tempChain[i];
      }
    }
  }


  ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////
  class Sets{

    int[]directionValues = {0,0,0,0,0,0};
    int currentDirection;
    int linkWhenMade;
    boolean isValid;
    boolean isTested;
```

```
int[] believedDataChain;

// Checkset variables
int totalNumbersOfDataChain = 100;
int[] aCheck = new int[totalNumbersOfDataChain];
int[] bCheck = new int[totalNumbersOfDataChain];
int[] cCheck = new int[totalNumbersOfDataChain];
int currentCheck = 1;

Sets(boolean additional, int interpretedValue, int copyThis, int lengthOfChain){
  believedDataChain = new int[lengthOfChain];
  for(int i = 0; i < lengthOfChain; i++){
    believedDataChain[i] = 0;
  }

  currentDirection = 0;
  isValid = false;
  isTested = false;

  if(additional){
    // copy root set
    //println("                                COPY");
    currentDirection = sets[copyThis].currentDirection;
    currentCheck = sets[copyThis].currentCheck;

    for(int i = 0; i < lengthOfChain; i++){
      believedDataChain[i] = sets[copyThis].believedDataChain[i];
    }

    for(int i = 0; i < directionValues.length; i++){
      directionValues[i] = sets[copyThis].directionValues[i];
    }

    for(int i = 0; i < totalNumbersOfDataChain; i++){
      aCheck[i] = sets[copyThis].aCheck[i];
      bCheck[i] = sets[copyThis].bCheck[i];
      cCheck[i] = sets[copyThis].cCheck[i];
    }

    // perform alternate move
    if(interpretedValue == 1){
      positiveMove();

      believedDataChain[currentLink] = 1;

    }
    else if (interpretedValue == 2){
      negativeMove();
      positiveMove();

      believedDataChain[currentLink] = 2;
    }
    else if (interpretedValue == 3){
      negativeMove();
      negativeMove();
      positiveMove();

      believedDataChain[currentLink] = 3;
    }

    linkWhenMade = currentLink;


    // Check set only
    aCheck[0] = 0;
    bCheck[0] = 0;
    cCheck[0] = 0;

  }

}

void printNow(){
  print("  CurrentDirection: " + currentDirection + "  ");

  for(int i = 0; i < directionValues.length; i++){
    print(directionValues[i] + " ");
  }
  println(" ");
}

void printBelievedDataChain(){
  //print("{");
  //for(int i = 0; i < believedDataChain.length-1; i++){
    //print(believedDataChain[i] + ",");
  //}
  //print(believedDataChain[believedDataChain.length-1] +"};");
}

int calculateConvexityValue(){
  int convexityValue = 0;

  for(int i = 0; i < believedDataChain.length; i++){
    if(believedDataChain[i] == 1)
      convexityValue += 1;
    if(believedDataChain[i] == 3)
      convexityValue -= 1;
    if(believedDataChain[i] == 4)
      convexityValue -= 2;
    if(believedDataChain[i] == 5)
      convexityValue -= 3;
```

```
      }

      //println("Convexity Value: " + convexityValue);
      return convexityValue;
    }

    boolean areThereRepeatPoints(){
      for(int i = 0; i < currentCheck-1; i++){ // need to ignore final point
        //println(aCheck[i] + " " + bCheck[i] + " " + cCheck[i]);
        for(int j = 0; j < i; j++){
          println("     " + aCheck[j] + " " + bCheck[j] + " " + cCheck[j]);
          if(aCheck[i] == aCheck[j] && bCheck[i] == bCheck[j] && cCheck[i] == cCheck[j]){
            //println("Repeated Point");
            return true;
          }
        }
      }
      return false;
    }

    boolean checkForReturnToStart(){
      if(aCheck[currentCheck-1] == 0 && bCheck[currentCheck-1] == 0 && cCheck[currentCheck-1] == 0){
        //println("Has returned to start");
        return true;
      }
      else{
        //println("Did not return to start");
        return false;
      }
    }

    void positiveMove(){
      directionValues[currentDirection]++;
      currentDirection++;

      if(currentDirection == directionValues.length){
        currentDirection = 0;
      }

      // find difference in direction values
      aCheck[currentCheck] = directionValues[0] - directionValues[3];
      bCheck[currentCheck] = directionValues[1] - directionValues[4];
      cCheck[currentCheck] = directionValues[2] - directionValues[5];
      currentCheck++;
      //println("Current check: " + currentCheck);
    }

    void negativeMove(){
      directionValues[currentDirection]++;
      currentDirection--;

      if(currentDirection < 0){
        currentDirection = 5;
      }

      // find difference in direction values
      aCheck[currentCheck] = directionValues[0] - directionValues[3];
      bCheck[currentCheck] = directionValues[1] - directionValues[4];
      cCheck[currentCheck] = directionValues[2] - directionValues[5];
      currentCheck++;

      //println("Current check: " + currentCheck);
    }
  }
}
void objectCode(){
  int currentCellX;
  int currentCellY;
  int order = 0;
  int tally = 0;
  boolean stay;
  CellBit[][] cellBit;
  cellBit = new CellBit[cols][rows];
  int firstEmptyX=0;
  int firstEmptyY=0;

  int visitsToFirst=1;
  int maxVisitsToFirst=0; // this number is affected when the first cell is a shared-link if; 2 then twice; 3
twice or thrice; 4 twice;

  //start at centre location
  currentCellX = cX;
  currentCellY = cY;

  // reset all code bit values on tempShape to 0;
  for (int i = 0; i < tempShape.objectCode.length; i++){
    tempShape.objectCode[i] = 0;
  }

  // check in direction 1 till empty cell occurs
  do{
    stay = true;
    int checkX = find2X(0,currentCellX);
    int checkY = find2Y(0,currentCellY);

    if(cell[checkX][checkY].cellState == -1){ // cell is empty
      cellBit[checkX][checkY] = new CellBit(checkX,checkY,order); // create first surrounding cell

      // note locations of first empty cell
      firstEmptyX = checkX;
      firstEmptyY = checkY;
```

```
      visitsToFirst = 1;

      // determine if first empty cell is a shared-link
      // check each of the six surrounding cells to determine if object cells or not
      int[] surroundingFirstEmpty = {0,0,0,0,0,0};
      if(cell[checkX + 1][checkY - 1].cellState == -1) // is empty
        surroundingFirstEmpty[0] = 1;
      if(cell[checkX + 1][checkY    ].cellState == -1) // -1 is empty
        surroundingFirstEmpty[1] = 1;
      if(cell[checkX    ][checkY + 1].cellState == -1) // -1 is empty
        surroundingFirstEmpty[2] = 1;
      if(cell[checkX - 1][checkY + 1].cellState == -1) // -1 is empty
        surroundingFirstEmpty[3] = 1;
      if(cell[checkX - 1][checkY    ].cellState == -1) // -1 is empty
        surroundingFirstEmpty[4] = 1;
      if(cell[checkX    ][checkY - 1].cellState == -1) // -1 is empty
        surroundingFirstEmpty[5] = 1;
      // compare each cell to the next cell including first to last
      for(int i = 0; i < surroundingFirstEmpty.length; i++){
        if(surroundingFirstEmpty[i] != surroundingFirstEmpty[(i+1)%6]){
          maxVisitsToFirst++; // add up the number of times this changes
        }
      }

      // divide this number by two, this is the maxVisitsToFirst
      maxVisitsToFirst = maxVisitsToFirst / 2;

      // update tempShape
      int contacts = cellBit[checkX][checkY].contacts;
      tempShape.update(order,contacts);

      order++;
      stay = false;
    }
    else{
      currentCellX = checkX;
      currentCellY = checkY;
    }
  }while(stay);

  // check next direction of currentCell
  int checkDirection = 0;
  do{
    stay = true; /// new bit
    int checkX = find2X(checkDirection,currentCellX);
    int checkY = find2Y(checkDirection,currentCellY);

    // if cell is empty
    if (cell[checkX][checkY].cellState == -1){
      if (cellBit[checkX][checkY] != null){ // has already been marked

        if (cellBit[checkX][checkY].order == order - 1){ // if cell is the cell marked exactly previously
          checkDirection++; //check next direction
        }
        else if (checkX == firstEmptyX && checkY == firstEmptyY){
          if(visitsToFirst == maxVisitsToFirst){
            stay = false; // gets back to first checked empty cell after completion, the shape has been been
marked
          }
          else{// add new cell as the first cell is a shared link and the search is not completed
            cellBit[checkX][checkY] = new CellBit(checkX,checkY, order);

            // update tempShape
            int contacts = cellBit[checkX][checkY].contacts;
            tempShape.update(order,contacts);
            order++;
            checkDirection++;
            visitsToFirst++;
          }
        }
        else { // add new cell
          cellBit[checkX][checkY] = new CellBit(checkX,checkY, order);

          // update tempShape
          int contacts = cellBit[checkX][checkY].contacts;
          tempShape.update(order,contacts);
          order++;
          checkDirection++;
        }

      }
      else {
        cellBit[checkX][checkY] = new CellBit(checkX,checkY, order);

        // update tempShape
        int contacts = cellBit[checkX][checkY].contacts;
        tempShape.update(order,contacts);
        order++;
        checkDirection++;
      }
    }
    else{ // if cell is part of the object
      currentCellX = checkX; // make this the currentCell
      currentCellY = checkY;

      checkDirection+=4; // update check direction
    }

    if (checkDirection > 5)
      checkDirection-=6; // make sure check direction is in range

    //tally++;
```

```
  }while(stay);//(tally <100); // this needs to be done in a better way

  // Check to see if code is new
  tempShape.visitsToFirstEmpty = maxVisitsToFirst;
  tempShape.findLength();
  tempShape.compare();

}


// temporary store for data
class CellBit{
  int contacts;
  int order;
  int x;
  int y;
  boolean valid = false;

  CellBit(int tempX, int tempY, int tempOrder){
    x = tempX;
    y = tempY;
    order = tempOrder;
    valid = true;

    contacts = 0;

    if (cell[x+1][y-1].cellState == 0) // NE
      contacts++;
    if (cell[x+1][y  ].cellState == 0) // E
      contacts++;
    if (cell[x  ][y+1].cellState == 0) // SE
      contacts++;
    if (cell[x-1][y+1].cellState == 0) // SW
      contacts++;
    if (cell[x-1][y  ].cellState == 0) // W
      contacts++;
    if (cell[x  ][y-1].cellState == 0) // NW
      contacts++;
  }

}

//////////////////
int find2X(int i, int x){
  int tempX = 0;

  if (i == 0)
   tempX = x+1;
  else if (i == 1)
   tempX = x+1;
  else if (i == 2)
   tempX = x;
  else if (i == 3)
   tempX = x-1;
  else if (i == 4)
   tempX = x-1;
  else // (i == 5)
   tempX = x;

  return(tempX);
}

int find2Y(int i, int y){
  int tempY = 0;

  if (i == 0)
   tempY = y-1;
  else if (i == 1)
   tempY = y;
  else if (i == 2)
   tempY = y+1;
  else if (i == 3)
   tempY = y+1;
  else if (i == 4)
   tempY = y;
  else // (i == 5)
   tempY = y-1;

  return(tempY);
}



class TempShape{
  int[] objectCode = new int[50];
  int codeLength;
  int ID;
  int numberOfCells;
  int[] locations;

  int visitsToFirstEmpty = 0;

  TempShape(int tempID, int tempNumberOfCells){
    for (int i=0; i < objectCode.length; i++){
      objectCode[i] = 0;
    }

    numberOfCells = tempNumberOfCells+1;
    ID = tempID;

    locations = new int[20];
```

```
    for (int i = 0; i < locations.length; i++){
      locations[i] = -100;
    }

    locations[0] = 0;
  }

  // change the states of the cells that contain a object cell up to
  // a certain point (the locations not updated do not appear)
  void markCells(int markColour, int upto){
    int xPlace;
    int yPlace;

    //cell[cX][cY].cellState = markColour;

    for (int i = 0; i <= upto; i++){
      if (locations[i] == -100){
        println("TempShape ERROR");
        exit(); // ERROR
      }

      xPlace = locationsX[locations[i]];
      yPlace = locationsY[locations[i]];
      cell[xPlace][yPlace].cellState = markColour;
    }
  }

  // find length of object code
  void findLength(){
    int count = 0;

    for (int i = 0; i < objectCode.length; i++){
      if (objectCode[i] != 0){
        count++;
      }
    }
    codeLength = count;
  }

  // UPDATE /////////////////////////////////////////////////////////////
  // update each bit of code
  void update(int codePosition, int value){
    objectCode[codePosition] = value;
  }

  // VALID OBJECT ///////////////////////////////////////////////////////////
  // is object shape valid (are all pieces touching, are any pieces overlapping)
  boolean validObject(int cellCheck){
    if (globalID <= 0){
      return(true);
    }

    if (locations[0] != 0){
      // Nothing at centre
      return(false);
    }

    // is any empty group of empty cells completely surrounded by shape cells
////////////////////////////////////////////
    if (checkForHollow){
      // refresh all cells
      for (int i = 0; i < cols; i++) {
        for (int j = 0; j < rows; j++) {
          cell[i][j].cellState = -1;      // -1 empty
        }
      }

      // mark object shape cells
      markCells(-2,cellCheck);            // -2 object

      // infect corner cells
      if (cell[0][0].cellState != -2)
        cell[0][0].cellState = 0;
      if (cell[cols-1][0].cellState != -2)
        cell[cols-1][0].cellState = 0;
      if (cell[0][rows-1].cellState != -2)
        cell[0][rows-1].cellState = 0;
      if (cell[cols-1][rows-1].cellState != -2)
        cell[cols-1][rows-1].cellState = 0;

      // spread infection
      boolean stable = false;
      int numberInfected = 0;
      int oldNumberInfected = 0;
      do{
        // check all cells (excluding outermost rows and columns
        for (int i = 1; i < cols-1; i++){
          for (int j = 1; j < rows-1; j++){
            //if cell is empty
            if (cell[i][j].cellState == -1){
              // check surroundings
              int[] sensed = new int[6];
              int tempCount = 0;

              sensed[tempCount++] = cell[i+1][j-1].cellState; // NE
              sensed[tempCount++] = cell[i+1][j  ].cellState; // E
              sensed[tempCount++] = cell[i  ][j+1].cellState; // SE
              sensed[tempCount++] = cell[i-1][j+1].cellState; // SW
              sensed[tempCount++] = cell[i-1][j  ].cellState; // W
              sensed[tempCount++] = cell[i  ][j-1].cellState; // NW

              for (int k = 0; k < sensed.length; k++){
```

```
              if(sensed[k] == 0 && cell[i][j].cellState == -1){ // if sense infected AND not currently infected
                cell[i][j].cellState = 0; // become infected
                numberInfected++; // count infected
              }
            }
          }
        }
      }

    // compare to old infected
    if (numberInfected == oldNumberInfected){
      // constant amount of infected
      stable = true;
    }

    oldNumberInfected = numberInfected;

    // when stable are any cells left un-infected and not object cells
    if (stable){
      // count number of empty (-1) cells
      for (int i = 1; i < cols-1; i++){
        for (int j = 1; j < rows-1; j++){
          //if cell is empty
          if (cell[i][j].cellState == -1){
            // shape is hollow and therefore invalid
            return(false);
          }
        }
      }
    }

      //stable = true;
    }while(!stable);

  }

  // is current cell touching other cell
  int xTemp = locationsX[tempShape.locations[cellCheck]];
  int yTemp = locationsY[tempShape.locations[cellCheck]];
  int contacts = 0;

  // REFRESH CELLS
  for (int i = 0; i < cols; i++) {
    for (int j = 0; j < rows; j++) {
      cell[i][j].cellState = -1;
    }
  }

  // mark cells up to but not including the current object cell
  markCells(-2,cellCheck);

  // does the current cell touch one of these lower numbered object cells
  if (cell[xTemp+1][yTemp-1].cellState == -2)  // NE
    contacts++;
  if (cell[xTemp+1][yTemp  ].cellState == -2) // E
    contacts++;
  if (cell[xTemp  ][yTemp+1].cellState == -2) // SE
    contacts++;
  if (cell[xTemp-1][yTemp+1].cellState == -2) // SW
    contacts++;
  if (cell[xTemp-1][yTemp  ].cellState == -2) // W
    contacts++;
  if (cell[xTemp  ][yTemp-1].cellState == -2) // NW
    contacts++;

  // if sharing a cell return false
  for(int i = 0; i < cellCheck; i++){
    if (locations[i] == locations[cellCheck]){
       // sharing cell - false shape
      return(false);
    }
  }

  if (contacts > 0){
    return(true);
  }
  else{
    return(false);
  }

}

// COMPARE//////////////////////////////////////////////////////////////////
// compares current shape to any previous object shapes
boolean compare(){
  boolean sameAs = false;

  for (int i = ID-1; i >= 0; i--){
    //if(numberOfCells != objectShape[i].numberOfCells){
      // does not match
      // check next object
    //}

    if (codeLength != objectShape[i].codeLength){
      // does not match
      // check next object
    }
    else {  // number of cells match and code length match
      // compare chains
      int[] tempA = new int[codeLength];
      int[] tempB = new int[codeLength];
```

```
        // transfer to tempA and tempB
        for (int j = 0; j < tempA.length; j++){
          tempA[j] = objectCode[j];
          tempB[j] = objectShape[i].objectCode[j];
        }

        // compare tempA to tempB
        int shiftCount = 0;
        for (int k = 0; k < tempA.length; k++){
          int similarity = 0;
          for (int j = 0; j < tempA.length; j++){
            if(tempA[j] == tempB[j]){
              // check next
              similarity++;
            }
            else{
              // shift tempB and re-check
              // leave loop
              similarity = 0;
              j = tempA.length; // force to leave loop
            }
          }

          // check similarity
          if (similarity == tempA.length){
            // not new shape
            sameAs = true;
          }
          else { // shift the tempB array
            if(shiftCount == tempA.length){
              // checked every combination with existing object
              // check next object
            }
            else {
              int tempFirst = tempB[0];

              for (int j = 0; j < tempB.length-1; j++){
                tempB[j] = tempB[j+1];
              }
              tempB[tempB.length-1] = tempFirst;
              shiftCount++;

            }
          }

        }// shift
      }// equal number of cells
    }// next object

    // if it does not match any previous shapes the ID is increased
    if (sameAs){
      return(true);
    }
    else {
      return(false);
    }

  }

  // COPYOBJECTSHAPE ////////////////////////////////////////////////////////////
  // copy temp shape data
  void copyObjectShape(int tempID){
    ID = objectShape[tempID].ID;
    numberOfCells = objectShape[tempID].numberOfCells;
    codeLength = objectShape[tempID].codeLength;

    for(int i = 0; i < objectCode.length; i++){
      objectCode[i] = objectShape[tempID].objectCode[i];
    }

    for(int i = 0; i < locations.length; i++){
      locations[i] = objectShape[tempID].locations[i];
    }
  }

}
```

Appendix D: Possible States for hBots and State-Relationships

Each of the first fifteen object shapes are listed, with the exception of object shape ID 11 which is omitted as it includes a number five in its data-chain. The state relationships are also included. By looking at a hBots own state, and the states of its lowest state neighbour and highest state neighbours the original hBots new state can be found. For each of the new states there is indication whether or not this state is achievable with a hBot in contact with a specific object shape. If the state is achievable it is market with a tick.

Own	Low	High	New	ID0	ID1	ID2	ID3	ID4	ID5	ID6	ID7	ID8	ID9	ID10	ID11	ID12	ID13	ID14
			1	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓
			2		✓	✓	✓	✓	✓	✓	✓		✓	✓		✓	✓	✓
			3				✓			✓	✓	✓	✓	✓		✓	✓	
1	1	1	4	✓	✓		✓	✓		✓	✓	✓	✓	✓		✓	✓	✓
1	1	2	5		✓	✓	✓	✓	✓	✓	✓		✓	✓		✓	✓	✓
1	1	3	6				✓			✓	✓	✓	✓	✓		✓	✓	
1	2	2	7				✓		✓					✓		✓		
1	2	3	8										✓				✓	
1	3	3	9															
2	1	1	10		✓	✓	✓		✓	✓	✓		✓	✓		✓	✓	
2	1	2	11					✓		✓	✓			✓		✓		✓
2	1	3	12											✓		✓		
2	2	2	13															✓
2	2	3	14															
2	3	3	15															
3	1	1	16				✓			✓	✓	✓	✓				✓	
3	1	2	17											✓		✓		
3	1	3	18															
3	2	2	19															
3	2	3	20															
3	3	3	21															
4	4	4	22	✓														
4	4	5	23															
4	4	6	24															
4	5	5	25		✓			✓						✓		✓		✓
4	5	6	26				✓			✓	✓		✓	✓		✓	✓	
4	6	6	27									✓						
5	4	4	28															
5	4	5	29															
5	4	6	30															
5	4	10	31		✓		✓						✓	✓		✓	✓	
5	4	11	32					✓		✓	✓			✓		✓		✓
5	4	12	33											✓		✓		
5	5	5	34															
5	5	6	35															
5	5	10	36			✓			✓	✓	✓							
5	5	11	37							✓	✓							
5	5	12	38															
5	6	6	39															
5	6	10	40							✓	✓							
5	6	11	41															
5	6	12	42															
5	10	10	43															
5	10	11	44															
5	10	12	45															
5	11	11	46															
5	11	12	47															
5	12	12	48															
6	4	4	49															
6	4	5	50															
6	4	6	51															
6	4	16	52				✓			✓	✓	✓	✓				✓	
6	4	17	53											✓		✓		
6	4	18	54															
6	5	5	55															
6	5	6	56															
6	5	16	57							✓	✓							
6	5	17	58															
6	5	18	59															
6	6	6	60															
6	6	16	61															
6	6	17	62															

6	6	18	63															
6	16	16	64															
6	16	17	65															
6	16	18	66															
6	17	17	67															
6	17	18	68															
6	18	18	69															
7	10	10	70				✓		✓									
7	10	11	71											✓		✓		
7	10	12	72															
7	11	11	73															
7	11	12	74															
7	12	12	75															
8	10	10	76															
8	10	11	77															
8	10	12	78															
8	10	16	79										✓				✓	
8	10	17	80															
8	10	18	81															
8	11	11	82															
8	11	12	83															
8	11	16	84															
8	11	17	85															
8	11	18	86															
8	12	12	87															
8	12	16	88															
8	12	17	89															
8	12	18	90															
8	16	16	91															
8	16	17	92															
8	16	18	93															
8	17	17	94															
8	17	18	95															
8	18	18	96															
9	16	16	97															
9	16	17	98															
9	16	18	99															
9	17	17	100															
9	17	18	101															
9	18	18	102															
10	5	5	103		✓	✓				✓	✓							
10	5	7	104				✓		✓					✓		✓		
10	5	8	105										✓				✓	
10	7	7	106															
10	7	8	107															
10	8	8	108															
11	5	5	109															
11	5	7	110															
11	5	8	111															
11	5	11	112					✓		✓	✓			✓		✓		
11	5	13	113															✓
11	5	14	114															
11	7	7	115															
11	7	8	116															
11	7	11	117											✓		✓		
11	7	13	118															
11	7	14	119															
11	8	8	120															
11	8	11	121															
11	8	13	122															
11	8	14	123															
11	11	11	124															
11	11	13	125															
11	11	14	126															
11	13	13	127															
11	13	14	128															
11	14	14	129															
12	5	5	130															
12	5	7	131															
12	5	8	132															
12	5	17	133											✓		✓		
12	5	19	134															
12	5	20	135															
12	7	7	136															
12	7	8	137															
12	7	17	138															
12	7	19	139															
12	7	20	140															
12	8	8	141															
12	8	17	142															
12	8	19	143															
12	8	20	144															
12	17	17	145															
12	17	19	146															
12	17	20	147															
12	19	19	148															
12	19	20	149															
12	20	20	150															
13	11	11	151															✓

13	11	13	152															
13	11	14	153															
13	13	13	154															
13	13	14	155															
13	14	14	156															
14	11	11	157															
14	11	13	158															
14	11	14	159															
14	11	17	160															
14	11	19	161															
14	11	20	162															
14	13	13	163															
14	13	14	164															
14	13	17	165															
14	13	19	166															
14	13	20	167															
14	14	14	168															
14	14	17	169															
14	14	19	170															
14	14	20	171															
14	17	17	172															
14	17	19	173															
14	17	20	174															
14	19	19	175															
14	19	20	176															
14	20	20	177															
15	17	17	178															
15	17	19	179															
15	17	20	180															
15	19	19	181															
15	19	20	182															
15	20	20	183															
16	6	6	184				✓			✓	✓	✓						
16	6	8	185										✓				✓	
16	6	9	186															
16	8	8	187															
16	8	9	188															
16	9	9	189															
17	6	6	190															
17	6	8	191															
17	6	9	192															
17	6	12	193											✓		✓		
17	6	14	194															
17	6	15	195															
17	8	8	196															
17	8	9	197															
17	8	12	198															
17	8	14	199															
17	8	15	200															
17	9	9	201															
17	9	12	202															
17	9	14	203															
17	9	15	204															
17	12	12	205															
17	12	14	206															
17	12	15	207															
17	14	14	208															
17	14	15	209															
17	15	15	210															
18	6	6	211															
18	6	8	212															
18	6	9	213															
18	6	18	214															
18	6	20	215															
18	6	21	216															
18	8	8	217															
18	8	9	218															
18	8	18	219															
18	8	20	220															
18	8	21	221															
18	9	9	222															
18	9	18	223															
18	9	20	224															
18	9	21	225															
18	18	18	226															
18	18	20	227															
18	18	21	228															
18	20	20	229															
18	20	21	230															
18	21	21	231															
19	12	12	232															
19	12	14	233															
19	12	15	234															
19	14	14	235															
19	14	15	236															
19	15	15	237															
20	12	12	238															
20	12	14	239															
20	12	15	240															

20	12	18	241															
20	12	20	242															
20	12	21	243															
20	14	14	244															
20	14	15	245															
20	14	18	246															
20	14	20	247															
20	14	21	248															
20	15	15	249															
20	15	18	250															
20	15	20	251															
20	15	21	252															
20	18	18	253															
20	18	20	254															
20	18	21	255															
20	20	20	256															
20	20	21	257															
20	21	21	258															
21	18	18	259															
21	18	20	260															
21	18	21	261															
21	20	20	262															
21	20	21	263															
21	21	21	264															

Appendix E: Advance SHM Program with GA

This is the program, written in Processing, that was utilised for the experimentation in Chapter 8 and Chapter 9. In the tests for Chapter 8, the aspects regarding the GA were ignored by changing the variables. In Chapter 9 the complete program was used including the genetic algorithm to solve the state rule behaviours problem for 11 different scenarios involving two different object shape types. The same program is used for the base-line generic method and the random method by adjusting the variables to suit.

```
//////////////////////////////////////////////////
// Creates Object Shapes and Data-Chains        //
//////////////////////////////////////////////////

PrintWriter output;

// runComparison
int maxCells = 10;

boolean showDisplay = false;

// Cell variables
Cell[][] cell;
int cols = 31;//objectsPerRow + ((objectsPerRow+1)*gapsize);//21;
int rows = 31;//objectsPerRow + ((objectsPerRow+1)*gapsize);//21;
float hexWidth = 15;
int cX = ((cols-1)/2);
int cY = ((rows-1)/2);

// types of shapes
int[] numberOfEachShape = new int[30];
boolean checkForHollow = true;

// general
int locationsX[]; // stores the x-coordinates of the spiral location value relative to the centre cell
int locationsY[]; // stores the y-coordinates of the spiral location value relative to the centre cell
int noOfRings = 12;
int lowestCellInRing[]; // stores to the lowest spiral location value for each ring
int falseCounter = 0; // counts the times that a false result is returned (to many falses force an update of
previous object cell)
int displayColour = 0;

ObjectShape[] objectShape;
int currentCell = 0;
int globalID = 0;
int currentRing = 0;

TempShape tempShape;

boolean newBaseObjShape = false;
boolean firstNewObject = true;
int baseID = 0;

boolean updateHighestCellPosition = false;

boolean halt = false;

int fileNumber = 1;
int limitID = 1000;

void setup(){
  output = createWriter("DataChainsCheckedPart" + fileNumber + ".txt");
  output.println("Object Shape ID; Number of Cells; Length of Data-Chain; Base Shape ID; Data-Chain;Number of
Branches;Number of valid Rep-Chains;Rep-Chains;Simple or Complex Shape;Required Visits to First Empty Cell in
forming data-chain");

  size(arenaWidth(),arenaHeight()); // function is in Cell
  background(0);

  // create cells (x,y)
  cell = new Cell[cols][rows];
  for (int i = 0; i < cols; i++) {
    for (int j = 0; j < rows; j++) {
      cell[i][j] = new Cell(i,j);
    }
  }

  // calculate relative spiral locations
  locationsX = new int[1000];
  locationsY = new int[1000];
  findXSpiralLocations();
  findYSpiralLocations();

  locationsX[0] = ((cols-1)/2); // x coordinate of 1st cell
  locationsY[0] = ((rows-1)/2); // y coordinate of 1st cell
```

```
  // calculate lowest cell number in each ring
  lowestCellInRing = new int[noOfRings];
  lowestCellInRing[0] = 0;
  for (int i = 1; i < lowestCellInRing.length; i++){
    lowestCellInRing[i] = (3*i*i) - (3*i) + 1;
  }

  // create objectShape
  objectShape = new ObjectShape[100000];
  objectShape[0] = new ObjectShape(0,0);

  tempShape = new TempShape(globalID,currentCell);

  // fill numberOfEachShape array
  for (int i = 0; i < numberOfEachShape.length; i++){
    numberOfEachShape[i] = 0;
  }

}

void draw(){

  if(globalID == limitID){
    limitID += 10000;
    fileNumber++;
    output.flush(); // Writes the remaining data to the file
    output.close(); // Finishes the file
    output = createWriter("DataChainsCheckedPart" + fileNumber + ".txt");
  }

  if(halt){
    //delay(250);
    halt = false;
  }

  refreshCells();
  tempShape.markCells(0,currentCell);


  // is object valid
  if (tempShape.validObject(currentCell)){
    ////////////////////////////////////////////////////////////////////////////// VALID OBJECT
    falseCounter = 0;

    refreshCells();
    tempShape.markCells(0,currentCell);
    objectCode();// create objectCode

    // does shape already exist
    if (tempShape.compare()){
      // update highest cell position
      updateHighestCellPosition = true;

      // display
      displayTheCells(3); // orange (exising shape)
    }
    else{
      // Display
      displayTheCells(1); // green (valid shape)
      halt = true;

      ////////////////////////////////////////////////////////////////////// SAVE SHAPE
      // save shape to object shapes
      objectShape[globalID] = new ObjectShape(globalID, currentCell);
      objectShape[globalID].copyTempShape();

      /////////////// check the data-chain
      objectShape[globalID].checkDataChain();

      // record number of shapes for each cell allowance
      numberOfEachShape[currentCell+1]++;

      // increase globalID
      globalID++;

      // update base shape and reset cell position
      tempShape = new TempShape(globalID,currentCell);
      tempShape.copyObjectShape(baseID);
      tempShape.ID = globalID;
      tempShape.numberOfCells = currentCell+1;

      tempShape.locations[currentCell] = 0;

      tempShape.locations[currentCell] = objectShape[globalID-1].locations[currentCell]+1;

    }

  }
  //////////////////////////////////////////////////////////////////////////// INVALID OBJECT
  else{
    // update highest cell position
    updateHighestCellPosition = true;

    displayTheCells(2);  // red (invalid shape)
  }

  // update highest cell position
  if (updateHighestCellPosition){
    updateHighestCellPosition = false;

    if (falseCounter == (lowestCellInRing[currentRing+1] - lowestCellInRing[currentRing])){
      // load new base object shape
```

```
      newBaseObjShape = true;
      falseCounter = 0;
    }

    tempShape.locations[currentCell]++;

    // update currentRing
    for (int i = 0; i < noOfRings; i++){
      if(tempShape.locations[currentCell] >= lowestCellInRing[i]){
        currentRing = i;
      }
    }

    // check current ring number, if move to next ring reset false counter
    if (tempShape.locations[currentCell] == lowestCellInRing[currentRing]){ //lowestCellInRing[currentRing + 1]
      falseCounter = 0;
    }

    falseCounter++;
  }

  ////////////////////////////////////////////////////////////////////////////// CREATE NEW BS
  if(newBaseObjShape){
    if (!firstNewObject){
      baseID++;
    }
    else{
      firstNewObject = false;
    }

    newBaseObjShape = false;
    tempShape.copyObjectShape(baseID);
    tempShape.numberOfCells++;
    currentCell = tempShape.numberOfCells-1;
    tempShape.locations[currentCell] = 0;
    tempShape.ID = globalID;

  }

  // compare the shapes
  if (currentCell == maxCells){
    println("FLUSH with currentCell: " + currentCell);
    output.flush(); // Writes the remaining data to the file
    output.close(); // Finishes the file
    exit();
  }
}

void refreshCells(){
  // REFRESH CELLS
  for (int i = 0; i < cols; i++) {
    for (int j = 0; j < rows; j++) {
        cell[i][j].cellState = -1;
    }
  }
}

void displayTheCells(int displayColour){
  if(showDisplay){
    refreshCells();
    tempShape.markCells(displayColour,currentCell);

    // DISPLAY GRID OF CELLS
    for (int i = 0; i < cols; i++) {
      for (int j = 0; j < rows; j++) {
        if (cell[i][j] != null)
          cell[i][j].display(j);
      }
    }


    refreshCells();
    tempShape.markCells(0,currentCell);
  }
}

class ActionRules{
  // for each state the hBot can be in (other than 0) it has three options
  // 1. disolve object
  // 2. HIGH probability of moving away from object (revert to state zero)
  // 3. LOW probability of moving away from object (revert to state zero)
  int[] actionRuleList = new int[stateRules.levelThreeMax];

  // the 41 state rules in the order that they appear in the genome.
  // see StateRelationshipsPoosible spreadsheet
  int[] geneToStateRule =
{1,4,22,25,26,27,5,31,32,33,36,37,40,6,52,53,57,7,70,71,8,79,2,10,103,104,105,11,112,113,117,12,133,13,151,3,16,1
84,185,17,193};

  ActionRules(){
    for(int i = 0; i < actionRuleList.length; i++){
      actionRuleList[i] = -1000;
    }

    for(int i = 0; i < geneToStateRule.length; i++){
      actionRuleList[geneToStateRule[i]] = genome[curSamp].bitList[i];
    }
  }

  void updateActionRules(){
    for(int i = 0; i < actionRuleList.length; i++){
      actionRuleList[i] = -1000;
```

```
    }

    for(int i = 0; i < geneToStateRule.length; i++){
      actionRuleList[geneToStateRule[i]] = genome[curSamp].bitList[i];
    }
  }

  void allUpdateActionRules(){
    for(int i = 0; i < maxPopulation; i++){
      for(int j = 0; j < geneToStateRule.length; j++){
        actionRuleList[geneToStateRule[j]] = genome[i].bitList[j];
      }
    }
  }


  int callRule(int state){
    if(state == 0)
      return(0);
    else {
      if(actionRuleList[state] == -1000){
        println("Error in call rule, action rules");
        exit();
        return(-1000);
      }
      return(actionRuleList[state]);
    }
  }
}

////////////////////////////////////////////////////////////////////////////////
// A Cell Object ///////////////////////////////////////////////////////////////
class Cell {
  // A cell object knows about its location in the grid
  float x,y; //x,y location of centre point
  //color colour = cWhite;
  int cellState = -1;
  float hexHeight = 4 * (0.5 * hexWidth) * (tan((radians(30))));

  // Cell Constructor
  Cell(float tempX, float tempY){
    x = (tempX*hexWidth)+2;
    y = (tempY*hexHeight*0.75)+1;
  }

  void display(int j) {

    rectMode(CENTER);


    noStroke();
    //strokeWeight(2);
    fill(colourFromNumber(cellState));



    float altTransX = x + (j*0.5*hexWidth) + (0.5*hexWidth);
    float altTransY = y + (0.5*hexHeight);

    ellipse(altTransX,altTransY,0.8*hexHeight,0.8*hexHeight); // draw circle (quicker)
  }
}

// which states are which colours
int colourFromNumber(int tempNumber){
  // Colours
  color cRed = color(255,0,0);
  color cOrange = color(255,165,0);
  color cYellow = color(255,255,0);
  color cGreen = color(50,205,50);
  color cBlue = color(0,0,255);
  color cPurple = color(128,0,128);

  color cWhite = color(255,255,255); // empty cell
  color cGrey = color(47,79,79); // object
  color cSilver = color(135,135,135); //agent
  color cBlack = color(0,0,0);

  // Colours
  if (tempNumber == 0) // state 0
    return(cSilver);
  else if (tempNumber == 1)  // state 1
    return(cGreen);
  else if (tempNumber == 2) // state 2
    return(cBlue);
  else if (tempNumber == 3)
    return(cRed);
  else if (tempNumber == 4)
    return(cOrange);
  else if (tempNumber == 5)
    return(cYellow);
  else if (tempNumber == 6)
    return(cPurple);

  else if (tempNumber == -1)
    return(cWhite);
  else if (tempNumber == -2)
    return(cGrey);
  else if (tempNumber == -3)
    return(cBlack);

  else
```

```
    return(cPurple);
}

// calc arena size based on size and number of cells
int arenaWidth(){
  return((int)(1.5*hexWidth*(cols-0.2)));
}
int arenaHeight(){
  return((int)(0.75*4*(0.5*hexWidth)*(tan((radians(30))))*(rows+0.5)));
}

void createBoundary(){
  // create walls (remove top left triangle of rhombus to create hexagon)
  for (int j = 0; j < cols/2; j++){
    for (int i = 0; i < (cols/2)+2 - j; i++){
      cell[i][j].cellState = -3;
    }
  }

  // create walls (remove bottom right triangle of rhombus to create hexagon)
  for (int j = 0; j < (cols-1)/2; j++){
    for (int i = cols - j; i < cols; i++){
      cell[i - 2][j + ((cols-1)/2)].cellState = -3;
    }
  }

  // create walls (provide boarder round edges)
  for (int i = 0; i < cols; i++) {
    for (int j = 0; j < cols; j++) {
      if (i < 2 || i > cols-3 || j < 2 || j > cols-3)
        cell[i][j].cellState = -3;
    }
  }

}
// create parent pool through tournament ranking
void createParentPool(){
  PrintWriter checkingGA;
  checkingGA = createWriter("check/checkingGA" + curGen + ".txt");

  boolean[] selectedAsParent =  new boolean[maxPopulation];

  for(int i = 0; i < selectedAsParent.length; i++){
    selectedAsParent[i] = false;
  }

  int[] randomForRank = new int[4];

  // start
  for(int curParent = 0; curParent < maxPopulation/2; curParent++){
    // select four different random values
    do {
      randomForRank[0] = (int)random(0,maxPopulation);
    } while(selectedAsParent[randomForRank[0]] == true);

    do {
      randomForRank[1] = (int)random(0,maxPopulation);
    } while(selectedAsParent[randomForRank[1]] == true || randomForRank[1] == randomForRank[0]);

    do {
      randomForRank[2] = (int)random(0,maxPopulation);
    } while(selectedAsParent[randomForRank[2]] == true || randomForRank[2] == randomForRank[0] || randomForRank[2]
== randomForRank[1]);

    do {
      randomForRank[3] = (int)random(0,maxPopulation);
    } while(selectedAsParent[randomForRank[3]] == true || randomForRank[3] == randomForRank[0] || randomForRank[3]
== randomForRank[1] || randomForRank[3] == randomForRank[2]);


    // calculate fitness value
    float[] fitness = new float[4];

    for(int i = 0; i < fitness.length; i++){
      int temp = randomForRank[i];                                    // (59+58)-(57+56)
      float firstLastFirstLast = (readAndWrite.collectedData[temp][59] + readAndWrite.collectedData[temp][58]) -
(readAndWrite.collectedData[temp][57] + readAndWrite.collectedData[temp][56]);
      float diffCorrectIncorrect = readAndWrite.collectedData[temp][45] - readAndWrite.collectedData[temp][50];
      float allOrSome = 0; // 1 if only correct removed, -1 if only incorrect removed, 0 otherwise

      if(readAndWrite.collectedData[temp][45] > 0 && readAndWrite.collectedData[temp][50] == 0){
        allOrSome = 1;
      }
      if(readAndWrite.collectedData[temp][50] > 0 && readAndWrite.collectedData[temp][45] == 0){
        allOrSome = -1;
      }
      fitness[i] = (504000 * allOrSome) + (84000 * diffCorrectIncorrect) + firstLastFirstLast;

      // temp for check
      checkingGA.print("Random" + i + ":" + randomForRank[i]);
      checkingGA.print("  fitness: " + fitness[i]);
      checkingGA.println(" allOrSome: " + allOrSome + "  diffCorrectIncorrect: " + diffCorrectIncorrect + "
firstLastFirstLast: " + firstLastFirstLast);
    }


    // rank fitness values
    int selectThisLineForPool = 0;

    float maxFitness = max(fitness);

    for(int i = 0; i < fitness.length; i++){
```

```
      if(fitness[i] == maxFitness){
        selectThisLineForPool = randomForRank[i];
        selectedAsParent[selectThisLineForPool] = true;
        i = fitness.length; // make sure only a single value is updated in the case that two or more share the
same fitness
      }
    }

    checkingGA.println("Parent selected: " + selectThisLineForPool);

    // copy selected sample
    for(int i = 0; i < 41; i++){

      println("curParent: " + curParent);
      println("i: " + i);
      println("selectThisLineForPool: " + selectThisLineForPool);


      println("A: " + parents[curParent][i]);

      parents[curParent][i] = readAndWrite.collectedData[selectThisLineForPool][i];
      checkingGA.print(readAndWrite.collectedData[selectThisLineForPool][i] + ",");
    }
    checkingGA.println("");
    checkingGA.println("");


  }
  // end
  checkingGA.flush(); // Writes the remaining data to the file
  checkingGA.close(); // Finishes the file
}

void createOffspring(){
  PrintWriter checkCrossover;
  checkCrossover = createWriter("check/checkingCrossover" + curGen + ".txt");

  int[] timesSelected = new int[maxPopulation/2];

  for(int i = 0; i < timesSelected.length; i++){
    timesSelected[i] = 0;
  }

  int parentA = -1;
  int parentB = -1;
  int[] viableList = new int[maxPopulation/2]; // list of currently viable parents 0,1,2,3,4,5,6,7,8,9
  int numberOfViable = maxPopulation/2; // number of parents that are viable
  int countNoUses; // count number of 0 uses of parent

  for(int i = 0; i < viableList.length; i ++){
    viableList[i] = i;
  }
  viableList = sort(viableList);
  viableList = reverse(viableList);


  // start
  for(int currentGenomeToUpdate = 0; currentGenomeToUpdate < maxPopulation; currentGenomeToUpdate =
currentGenomeToUpdate + 2){

    // pick two different parents at random
    // check parents have not been used twice already

    // count number with no uses as parent
    // Parent A
    countNoUses = 0;

    for(int i = 0; i < maxPopulation/2; i++){
      if(timesSelected[i] == 0){
        countNoUses++;
      }
    }

    if(numberOfViable == 3 && countNoUses == 1){ // stop a single parent been left to breed with itself
      for(int i = 0; i < maxPopulation/2; i++){
        if(timesSelected[i] == 0){
          parentA = i;
          timesSelected[parentA]++;
        }
      }
    }
    else{
      parentA = viableList[(int)random(0,numberOfViable)]; // select random parent from available list
      println("ParentA: " + parentA);
      timesSelected[parentA]++; // count the number of times the parent has been used

      if(timesSelected[parentA] == 2){ // if the parent has been used twice, remove it from the available list
        // remove from list
        for(int i = 0; i < viableList.length; i++){
          if(viableList[i] == parentA){
            viableList[i] = -1; // set the unviable option to -1
          }
        }

        numberOfViable--; // reduce the number of viable options
        viableList = sort(viableList);
        viableList = reverse(viableList); // arrange list so unviable options (those with -1) are listed last

        for(int i = 0; i < viableList.length; i++){
          print(viableList[i] + ",");
        }
        println("");
```

```
      println("numberOfViable " + numberOfViable);
    }
  }

  //Parent B
  countNoUses = 0;

  for(int i = 0; i < maxPopulation/2; i++){
    if(timesSelected[i] == 0){
      countNoUses++;
    }
  }

  if(numberOfViable == 3 && countNoUses == 1){ // stop a single parent been left to breed with itself
    for(int i = 0; i < maxPopulation/2; i++){
      if(timesSelected[i] == 0){
        parentB = i;
        timesSelected[parentB]++;
      }
    }
  }
  else{
    do{
      parentB = viableList[(int)random(0,numberOfViable)];
    }while(parentB == parentA); // check parent B is not the same as parent A

    println("ParentB: " + parentB);
    timesSelected[parentB]++;

    if(timesSelected[parentB] == 2){
      // remove from list
      for(int i = 0; i < viableList.length; i++){
        if(viableList[i] == parentB){
          viableList[i] = -1;
        }
      }
      numberOfViable--;
      viableList = sort(viableList);
      viableList = reverse(viableList);

      for(int i = 0; i < viableList.length; i++){
        print(viableList[i] + ",");
      }
      println("");
      println("numberOfViable " + numberOfViable);
    }
  }

  // select two different random points, for cross over
  int crossoverA;
  int crossoverB;

  crossoverA = (int)random(0,genome[0].bitList.length-2); // bitListLength - 2, cross over happens after this
value

  do{
    crossoverB = (int)random(0,genome[0].bitList.length-2);
  } while (crossoverB == crossoverA); // cross over A and B cannot be the same

  // produce two offspring and store
  boolean switchFirstParent = false;

  for(int i = 0; i < genome[0].bitList.length; i++){
    if(switchFirstParent == false){
      genome[currentGenomeToUpdate   ].bitList[i] = (int)parents[parentA][i];
      genome[currentGenomeToUpdate +1].bitList[i] = (int)parents[parentB][i];
    }
    else{
      genome[currentGenomeToUpdate   ].bitList[i] = (int)parents[parentB][i];
      genome[currentGenomeToUpdate +1].bitList[i] = (int)parents[parentA][i];
    }

    if(i == crossoverA || i == crossoverB){
      // switch
      if(switchFirstParent == false){
        switchFirstParent = true;
      }
      else{
        switchFirstParent = false;
      }
    }
  }

  // tempory print out check below
  checkCrossover.println("ParentA: " + parentA);

  for(int i = 0; i < genome[0].bitList.length; i ++){
    checkCrossover.print(parents[parentA][i] + ",");
  }
  checkCrossover.println("");


  checkCrossover.println("ParentB: " + parentB);

  for(int i = 0; i < genome[0].bitList.length; i ++){
    checkCrossover.print(parents[parentB][i] + ",");
  }
  checkCrossover.println("");


  checkCrossover.println("crossoverA: " + crossoverA + "  crossoverB: " + crossoverB);
```

```
    checkCrossover.println("ChildA: ");

    for(int i = 0; i < genome[0].bitList.length; i ++){
      checkCrossover.print(genome[currentGenomeToUpdate   ].bitList[i] + ",");
    }
    checkCrossover.println("");


    checkCrossover.println("ChildB: ");

    for(int i = 0; i < genome[0].bitList.length; i ++){
      checkCrossover.print(genome[currentGenomeToUpdate +1].bitList[i] + ",");
    }
    checkCrossover.println("");
    checkCrossover.println("");
    checkCrossover.println("");
    // end temporary print out
  }
  // end
  checkCrossover.flush(); // Writes the remaining data to the file
  checkCrossover.close(); // Finishes the file
}

void mutation(){
  PrintWriter checkMutation;
  checkMutation = createWriter("check/checkMutation" + curGen + ".txt");

  int mutationRate = genome[0].bitList.length;

  // each bit in each genome has a probability of mutating to one of the other option
  for(int i = 0; i < maxPopulation; i++){

    // printold
    for(int j = 0; j < genome[0].bitList.length; j++){
      checkMutation.print(genome[i].bitList[j] + ",");
    }
    checkMutation.println("");

    for(int j = 0; j < genome[0].bitList.length; j++){
      // check if mutation happens
      if(random(mutationRate) < 1){
        // if mutation happens, check current value of bit
        int currentBitValue = genome[i].bitList[j];
        int newBitValue;
        do{
          // switch bit with an equal probability to on the other other bit values.
          newBitValue = (int)random(1,4); // returns 1,2,3

        }while (newBitValue == currentBitValue);

        genome[i].bitList[j] = newBitValue;

        // print change
        checkMutation.println("Change bit: " + j + " from " + currentBitValue + " to " + newBitValue);
      }
    }

    // print new
    for(int j = 0; j < genome[0].bitList.length; j++){
      checkMutation.print(genome[i].bitList[j] + ",");
    }
    checkMutation.println("");
    checkMutation.println("");
  }
  checkMutation.flush(); // Writes the remaining data to the file
  checkMutation.close(); // Finishes the file
}

class Genome{
  int ID;
  int parentA;
  int parentB;
  int mutationRate;

  int totalCorrect;
  int totalWrong;
  int medianCorrect;
  int medianWrong;

  int[] bitList = new int[41];

  Genome(){
    if(typeOfTest == 1) {// hand solved solutions
      handSolutionList();
    }
    else if(typeOfTest == 2){ // random
      for(int i = 0; i < bitList.length; i++){
        bitList[i] = (int)(random(1,4)); // insert random 1,2,3
      }
    }
    else if(typeOfTest == 3){ // genetic algorithm
      for(int i = 0; i < bitList.length; i++){
        bitList[i] = (int)(random(1,4)); // insert random 1,2,3
      }
    }
  }


  void handSolutionList(){

    if(findThisShape == 5 || findThisShape == 6 || findThisShape == 8 || findThisShape == 9 || findThisShape ==
10 || findThisShape == 14){
```

```
    }
    else{
      println("Error wrong shape type");
      exit();
    }
    if(ignoreThisShape == 5 || ignoreThisShape == 6 || ignoreThisShape == 8 || ignoreThisShape == 9 ||
ignoreThisShape == 10 || ignoreThisShape == 14){

    }
    else{
      println("Error wrong shape type");
      exit();
    }

    int[] find5Ignore6   = {3,2,2,2,2,2,3,2,2,2,2,2,2,2,2,2,1,1,2,2,2,3,3,2,1,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2};
    int[] find5Ignore8   = {3,2,2,2,2,2,1,2,2,2,1,2,2,2,2,2,2,1,1,2,2,2,1,1,2,1,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2};
    int[] find5Ignore9   = {3,2,2,2,2,2,3,2,2,2,1,2,2,2,2,2,2,1,1,2,2,2,3,3,2,1,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2};
    int[] find5Ignore10  = {3,2,2,2,2,2,2,3,2,2,2,1,2,2,2,2,2,2,2,3,1,2,2,2,3,3,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2};
    int[] find5Ignore14  = {3,2,2,2,2,2,2,3,2,2,2,1,2,2,2,2,2,2,2,1,1,2,2,2,3,1,2,1,2,2,2,2,2,2,2,2,2,2,2,2,2};

    int[] find6Ignore5   = {3,1,2,2,1,2,3,2,1,2,2,1,1,1,1,2,1,2,2,2,2,2,2,3,3,1,2,2,1,1,2,2,2,2,2,2,1,1,1,2,2,2};
    int[] find6Ignore8   = {3,3,2,2,1,2,1,2,1,2,1,1,1,3,2,2,1,2,2,2,2,2,2,1,1,1,2,2,1,1,2,2,2,2,2,2,3,3,2,2,2,2};
    int[] find6Ignore9   = {3,3,2,2,2,2,3,2,1,2,1,1,1,1,3,2,2,1,2,2,2,2,2,3,3,1,2,2,1,1,2,2,2,2,2,2,3,3,1,2,2,2};
    int[] find6Ignore10  = {3,3,2,2,2,2,2,3,2,2,2,1,1,1,3,1,2,1,2,2,2,2,2,3,3,1,2,2,3,2,2,2,2,2,2,2,3,1,1,2,2,2};
    int[] find6Ignore14  = {3,3,2,2,1,2,2,3,2,2,2,1,1,1,1,1,2,1,2,2,2,2,2,2,3,1,1,2,2,3,1,2,2,2,2,2,2,1,1,1,2,2,2};

    int[] find8Ignore5   = {3,1,2,2,2,1,2,2,2,2,2,2,2,1,1,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,1,1,1,2,2,2};
    int[] find8Ignore6   = {3,3,2,2,2,2,1,2,2,2,2,2,2,2,2,3,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,3,3,2,2,2,2};
    int[] find8Ignore9   = {3,3,2,2,2,2,1,2,2,2,2,2,2,2,2,3,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,3,3,1,2,2,2};
    int[] find8Ignore10  = {3,3,2,2,2,2,1,2,2,2,2,2,2,2,2,3,1,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,3,1,1,2,2,2};
    int[] find8Ignore14  = {3,3,2,2,2,2,1,2,2,2,2,2,2,2,2,1,1,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,1,1,1,2,2,2};

    int[] find9Ignore5   = {3,1,2,2,1,2,3,1,2,2,2,2,2,1,1,2,2,2,2,2,1,1,3,3,2,2,1,2,2,2,2,2,2,2,2,1,1,2,1,2,2};
    int[] find9Ignore6   = {3,3,2,2,2,2,2,3,1,2,2,2,2,2,2,3,2,2,2,2,2,2,1,1,3,3,2,2,1,2,2,2,2,2,2,2,2,3,3,2,1,2,2};
    int[] find9Ignore8   = {3,3,2,2,1,2,1,1,2,2,2,2,2,2,3,2,2,2,2,2,2,1,1,1,1,2,2,1,2,2,2,2,2,2,2,2,3,3,2,1,2,2};
    int[] find9Ignore10  = {3,3,2,2,2,2,2,3,2,2,2,2,2,2,3,1,2,2,2,2,2,1,1,3,3,2,2,1,2,2,2,2,2,2,2,2,3,1,2,1,2,2};
    int[] find9Ignore14  = {3,3,2,2,1,2,3,1,2,2,2,2,2,2,1,1,2,2,2,2,2,2,1,1,3,1,2,2,1,2,2,2,2,2,2,2,2,1,1,2,1,2,2};

    int[] find10Ignore5  = {3,1,2,1,1,2,3,1,1,1,2,2,2,1,2,1,2,3,2,1,2,2,3,3,2,2,2,1,1,2,1,1,1,2,2,1,2,2,2,1,1};
    int[] find10Ignore6  = {3,3,2,1,2,2,3,1,2,1,2,2,2,3,2,1,2,1,2,1,2,2,3,3,2,1,2,3,2,2,1,1,1,2,2,3,2,2,2,1,1};
    int[] find10Ignore8  = {3,3,2,1,1,2,1,1,1,1,2,2,2,3,2,1,2,1,2,1,2,2,1,1,2,1,2,1,1,2,1,1,1,2,2,3,2,2,2,1,1};
    int[] find10Ignore9  = {3,3,2,1,2,2,3,2,1,1,2,2,2,3,2,1,2,1,2,1,2,2,3,3,2,1,2,1,1,2,1,1,1,2,2,3,2,2,2,1,1};
    int[] find10Ignore14 = {3,3,2,2,1,2,3,1,2,1,2,2,2,1,2,1,2,1,2,1,2,2,3,1,2,1,2,3,1,2,1,1,1,2,2,1,2,2,2,1,1};

    int[] find14Ignore5  = {3,1,2,1,2,2,3,2,1,2,2,2,2,2,2,2,2,2,2,2,2,2,2,3,2,2,2,2,1,2,1,2,2,2,1,1,2,2,2,2,2,2};
    int[] find14Ignore6  = {3,3,2,1,2,2,3,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,3,2,2,2,2,3,2,1,2,2,2,1,1,2,2,2,2,2,2};
    int[] find14Ignore8  = {3,3,2,1,2,2,1,2,1,2,2,2,2,2,2,2,2,2,2,2,2,2,1,2,2,2,2,1,2,1,2,2,2,1,1,2,2,2,2,2,2};
    int[] find14Ignore9  = {3,3,2,1,2,2,3,2,1,2,2,2,2,2,2,2,2,2,2,2,2,2,3,2,2,2,2,1,2,1,2,2,2,1,1,2,2,2,2,2,2};
    int[] find14Ignore10 = {3,3,2,2,2,2,3,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,3,2,2,2,2,3,2,1,2,2,2,1,1,2,2,2,2,2,2};


    if(findThisShape == 5){
      if(ignoreThisShape == 6){
        for(int i = 0; i < bitList.length; i++){
          bitList[i] = find5Ignore6[i];
        }
      }

      else if(ignoreThisShape == 8){
       for(int i = 0; i < bitList.length; i++){
          bitList[i] = find5Ignore8[i];
        }
      }
      else if(ignoreThisShape == 9){
        for(int i = 0; i < bitList.length; i++){
          bitList[i] = find5Ignore9[i];
        }
      }
      else if(ignoreThisShape == 10){
        for(int i = 0; i < bitList.length; i++){
          bitList[i] = find5Ignore10[i];
        }
      }
      else if(ignoreThisShape == 14){
        for(int i = 0; i < bitList.length; i++){
          bitList[i] = find5Ignore14[i];
        }
      }

    }

    if(findThisShape == 6){
      if(ignoreThisShape == 5){
        for(int i = 0; i < bitList.length; i++){
          bitList[i] = find6Ignore5[i];
        }
      }
      else if(ignoreThisShape == 8){
        for(int i = 0; i < bitList.length; i++){
          bitList[i] = find6Ignore8[i];
        }
      }
      else if(ignoreThisShape == 9){
        for(int i = 0; i < bitList.length; i++){
          bitList[i] = find6Ignore9[i];
        }
      }
      else if(ignoreThisShape == 10){
        for(int i = 0; i < bitList.length; i++){
          bitList[i] = find6Ignore10[i];
        }
```

```
  }
  else if(ignoreThisShape == 14){
    for(int i = 0; i < bitList.length; i++){
      bitList[i] = find6Ignore14[i];
    }
  }
}

if(findThisShape == 8){
  if(ignoreThisShape == 5){
    for(int i = 0; i < bitList.length; i++){
      bitList[i] = find8Ignore5[i];
    }
  }
  else if(ignoreThisShape == 6){
    for(int i = 0; i < bitList.length; i++){
      bitList[i] = find8Ignore6[i];
    }
  }
  else if(ignoreThisShape == 9){
    for(int i = 0; i < bitList.length; i++){
      bitList[i] = find8Ignore9[i];
    }
  }
  else if(ignoreThisShape == 10){
    for(int i = 0; i < bitList.length; i++){
      bitList[i] = find8Ignore10[i];
    }
  }
  else if(ignoreThisShape == 14){
    for(int i = 0; i < bitList.length; i++){
      bitList[i] = find8Ignore14[i];
    }
  }
}

if(findThisShape == 9){
  if(ignoreThisShape == 5){
    for(int i = 0; i < bitList.length; i++){
      bitList[i] = find9Ignore5[i];
    }
  }
  else if(ignoreThisShape == 6){
    for(int i = 0; i < bitList.length; i++){
      bitList[i] = find9Ignore6[i];
    }
  }
  else if(ignoreThisShape == 8){
    for(int i = 0; i < bitList.length; i++){
      bitList[i] = find9Ignore8[i];
    }
  }
  else if(ignoreThisShape == 10){
    for(int i = 0; i < bitList.length; i++){
      bitList[i] = find9Ignore10[i];
    }
  }
  else if(ignoreThisShape == 14){
    for(int i = 0; i < bitList.length; i++){
      bitList[i] = find9Ignore14[i];
    }
  }
}
if(findThisShape == 10){
  if(ignoreThisShape == 5){
    for(int i = 0; i < bitList.length; i++){
      bitList[i] = find10Ignore5[i];
    }
  }
  else if(ignoreThisShape == 6){
    for(int i = 0; i < bitList.length; i++){
      bitList[i] = find10Ignore6[i];
    }
  }
  else if(ignoreThisShape == 8){
    for(int i = 0; i < bitList.length; i++){
      bitList[i] = find10Ignore8[i];
    }
  }
  else if(ignoreThisShape == 9){
    for(int i = 0; i < bitList.length; i++){
      bitList[i] = find10Ignore9[i];
    }
  }
  else if(ignoreThisShape == 14){
    for(int i = 0; i < bitList.length; i++){
      bitList[i] = find10Ignore14[i];
    }
  }
}
if(findThisShape == 14){
  if(ignoreThisShape == 5){
    for(int i = 0; i < bitList.length; i++){
      bitList[i] = find14Ignore5[i];
    }
  }
  else if(ignoreThisShape == 6){
    for(int i = 0; i < bitList.length; i++){
      bitList[i] = find14Ignore6[i];
    }
  }
  else if(ignoreThisShape == 8){
```

```
        for(int i = 0; i < bitList.length; i++){
          bitList[i] = find14Ignore8[i];
        }
      }
      else if(ignoreThisShape == 9){
        for(int i = 0; i < bitList.length; i++){
          bitList[i] = find14Ignore9[i];
        }
      }
      else if(ignoreThisShape == 10){
        for(int i = 0; i < bitList.length; i++){
          bitList[i] = find14Ignore10[i];
        }
      }
    }
  }
}

class HBot{
  int x,y;
  int xOld, yOld;
  int state = 0;
  int actionState = 0;
  int nextState = 0;
  int stateLevel = 0;
  int[] sensed = new int[6];
  boolean[] possDirections = new boolean[6]; // is it possible to move in this direction

  int lowMoveProb = 1; // random(100) < value then do
  int highMoveProb = 10;//10; // random(100) < value then do

  /*
        |5|0|
       |4|x|1|
        |3|2|
  */

  HBot(int tempA){
    x = locationsX[tempA];
    y = locationsY[tempA];

    for(int i = 0; i < sensed.length; i++){
      sensed[i] = 0;
    }
  }

  void move(int currentBot){
    if(state == 0){
      xOld = x;
      yOld = y;

      // all directions are possible
      for (int i = 0; i < 6; i++){
        possDirections[i] = true;
      }

      // stop travel in direction of any object cell in inner sensor ring
      for(int i = 0; i < 6; i++){
        if (sensed[i] == -2 || sensed[i] == -3){ // if object cell or boundary cell
          possDirections[i] = false;
        }
      }

      int direction = 0;
      int[] checkAll = {0,0,0,0,0,0};
      boolean leaveLoop = false;

      do{
        direction = (int)(random(6)); // pick random direction
        checkAll[direction] = 1; // note direction has been selected
        int totalCheck = 0;
        for (int i = 0; i < checkAll.length; i++){
          totalCheck += checkAll[i]; // tally the number of directions selected
        }

        if (totalCheck == 6){ // if all directions selected
          leaveLoop = true;
          direction = -1; // if all directions are not possible stay still
        }

      }while (leaveLoop == false && possDirections[direction] == false); // if direction not possible loop

      if(direction != -1 && possDirections[direction] == false){
        println("bad move by hBot");
        exit();
      }

      // move in the selected direction
      if (direction == 0){ // NE
        x += 1;
        y -= 1;
      }
      else if (direction == 1){ // E
        x += 1;
      }
      else if (direction == 2){ // SE
        y += 1;
      }
      else if (direction == 3){ // SW
        x -= 1;
        y += 1;
      }
```

```
    else if (direction == 4){ // W
      x -= 1;
    }
    else if (direction == 5){ // NW
      y -= 1;
    }
    else{
      x = x;
      y = y;
      // stay still
      //println("Stay still");
    }

    // if movement isn't possible because of other hBot stay still
    // check for bots on top of each other
    for (int i = 0; i < noOfBots; i++){
      if(i != currentBot){ // is this another hBot
        if(hBot[i].x == x && hBot[i].y == y){ // is there an agent in this cell
          //println("hBot in that cell!   " + direction); // move back to original possition
          x = xOld;
          y = yOld;
        }
      }
    }
  }
}

void sense(){
  int tempCount = 0;

  sensed[tempCount++] = cell[x+1][y-1].cellState; // NE
  sensed[tempCount++] = cell[x+1][y  ].cellState; // E
  sensed[tempCount++] = cell[x  ][y+1].cellState; // SE
  sensed[tempCount++] = cell[x-1][y+1].cellState; // SW
  sensed[tempCount++] = cell[x-1][y  ].cellState; // W
  sensed[tempCount++] = cell[x  ][y-1].cellState; // NW
}

void findNextState(){
 // if in contact with another agent change to state 1
  //state = 0;
  int noOfObjCont = 0;
  int noOfhBotCont = 0;
  int stateContA = -1;
  int stateContB = -1;
  boolean switchAB = false;

  // count object side contacts
  for(int i = 0; i < 6; i++){
    if (sensed[i] == -2){ // -2 is object
      noOfObjCont++;
    }
  }

  // update state relative to number of side contacts allowed by rule system
  if (state == 0){
    if (noOfObjCont == 1)
      nextState = 1;
    else if (noOfObjCont == 2 && stateRules.levelOneMax >= 2)
      nextState = 2;
    else if (noOfObjCont == 3 && stateRules.levelOneMax >= 3)
      nextState = 3;
    else if (noOfObjCont == 4 && stateRules.levelOneMax >= 4)
      nextState = 4;
    else if (noOfObjCont == 5 && stateRules.levelOneMax >= 5)
      nextState = 5;
  }

  // check for neighbouring agents and their states relative to the hBots current level
  for(int i = 0; i < 6; i++){
    if (sensed[i] != -2 && sensed[i] != -1 && sensed[i] != 0 && state != 0 && state < maximumState){
      if (findLevelFromState(sensed[i]) == stateLevel){
        if (switchAB == false){
          stateContA = sensed[i];
          switchAB = true;
          noOfhBotCont++;
        }
        else {
          stateContB = sensed[i];
          noOfhBotCont++;
        }
      }
      else if (findLevelFromState(sensed[i]) > stateLevel){
        if (switchAB == false){
          stateContA = findStateAtSameLevel(stateLevel,sensed[i]);
          switchAB = true;
          noOfhBotCont++;
        }
        else {
          stateContB = findStateAtSameLevel(stateLevel,sensed[i]);
          noOfhBotCont++;
        }
      }
    }
  }

  if (state != 0 && (stateContA != -1 || stateContB != -1)){
    if (noOfhBotCont == 2){
      if (stateContA <= stateContB){
        nextState = stateRules.ruleList[state][stateContA][stateContB];
      }
      else{
```

```
          nextState = stateRules.ruleList[state][stateContB][stateContA];
        }
      }
    }

    // reverts to zero state (search) if not touching an object
    if (noOfObjCont == 0){
      nextState = 0;
    }


  }

  void changeState(){
    state = nextState;
    stateLevel = findLevelFromState(state);
    //if(stateLevel > 0){
    //  print(state + "[" + stateLevel + "]");
    //}
    actionState = actionRules.callRule(state);
    //if(stateLevel > 0){
    //  print("(" + actionState + "),");
    //}
  }

  void act(){
    if(actionState == 0){
      //do nothing
    }
    else if (actionState == 1){
      //remove object in contact with
      removeObject(x,y);
    }
    else if (actionState == 2){
      // revert to state 0
      // high probability of state 0
      if (random(100) < highMoveProb)
        state = 0;
    }
    else if (actionState == 3){
      // Slim chance to revert to state 0)
      // low probability of state 0
      if (random(100) < lowMoveProb)
        state = 0;
    }
    else{
      println("ERROR in act, hBot");
    }
  }
}

class Objects{
  int x,y; //location
  int noOfCells;
  int type; // each type is a different possible parter based on the number of cells used
  int rotation; // each object has 6 different rotations
  int otherXY[]; // 0 - 18 the cells that surround any central cell
  boolean deleted = false;

  int osID; // object shape identifying number

  Objects(int ID){
    x = (int)(cols/2);
    y = (int)(cols/2);

    //x = tempX;
    //y = tempY;

    // find x and y positions
    if(ID < 6){
      x = locationsX[(3*outerObjRing*outerObjRing) + (3*outerObjRing) - (outerObjRing*ID)];
      y = locationsY[(3*outerObjRing*outerObjRing) + (3*outerObjRing) - (outerObjRing*ID)];
    }
    else{
      x = locationsX[(3*innerObjRing*innerObjRing) + (3*innerObjRing) - (innerObjRing*(ID-6)) + (innerObjRing/2)];
      y = locationsY[(3*innerObjRing*innerObjRing) + (3*innerObjRing) - (innerObjRing*(ID-
6)) + (innerObjRing/2)];
    }

    // change shape on odd and even
    if(ID%2 == 0){
      osID = findThisShape;
    }
    else{
      osID = ignoreThisShape;
    }

    noOfCells = noOfCellsFromOsID(osID); // number of cells in object

    rotation = (int)(random(6));

    //cellLocations
    otherXY = new int[noOfCells-1];
    cellLocations();
    rotation();
  }

  void markCells(){
    cell[x][y].cellState = -2; // -2 is cGrey object

    for (int i = 0; i < noOfCells - 1; i++){
      int tempX = findX(otherXY[i]);
```

```
    int tempY = findY(otherXY[i]);
    cell[tempX][tempY].cellState = -2;
  }
}

//////////////////////////
void cellLocations(){
  // one cell allowance
  if (osID == 0){
      // no other cells
      // object shape ID 0
  }
  // two cell allowance
  else if (osID == 1){
      //objectShape ID 1
      otherXY[0] = 0;
  }
  // three cell allowance
  else if (osID == 2){ // cluster
    // object shape ID 2
    otherXY[0] = 0;
    otherXY[1] = 1;
  }
  else if (osID == 3){ // curve
    // object shape ID 3
    otherXY[0] = 0;
    otherXY[1] = 4;
  }
  else if (osID == 4){ // straight
    // object shape ID 4
    otherXY[0] = 0;
    otherXY[1] = 3;
  }

  // 4 four cell allowance
  else if (osID == 14){ // straight
    // object shape ID 14
    otherXY[0] = 0;
    otherXY[1] = 3;
    otherXY[2] = 13;
  }
  else if (osID == 12){ // clockwise kink
    // object shape ID 12
    otherXY[0] = 0;
    otherXY[1] = 2;
    otherXY[2] = 11;
  }
  else if (osID == 10){ // anti-clockwise kink
    // object shape ID 10
    otherXY[0] = 0;
    otherXY[1] = 4;
    otherXY[2] = 15;
  }
  else if (osID == 13){ // wiggle 1
    // object shape ID 13
    otherXY[0] = 0;
    otherXY[1] = 2;
    otherXY[2] = 12;
  }
  else if (osID == 9){ // wiggle 2
    // object shape ID 9
    otherXY[0] = 0;
    otherXY[1] = 4;
    otherXY[2] = 14;
  }
  else if (osID == 6){ // cherry left
    // object shape ID 6
    otherXY[0] = 0;
    otherXY[1] = 3;
    otherXY[2] = 4;
  }
  else if (osID == 7){ // cherry right
    // object shape ID 7
    otherXY[0] = 0;
    otherXY[1] = 2;
    otherXY[2] = 3;
  }
  else if (osID == 5){ // cluster
    // object shape ID 5
    otherXY[0] = 0;
    otherXY[1] = 1;
    otherXY[2] = 5;
  }
  else if (osID == 11){ // curve
    // object shape ID 11
    otherXY[0] = 0;
    otherXY[1] = 2;
    otherXY[2] = 10;
  }
  else if (osID == 8){ // three-way
    // object shape ID 8
    otherXY[0] = 0;
    otherXY[1] = 2;
    otherXY[2] = 4;
  }
}

// RotateObjects
void rotation(){
  for (int i = 0; i < noOfCells-1; i++){
    if (otherXY[i] < 6){ // inner ring
      otherXY[i] += rotation;
```

```
        if (otherXY[i] >= 6)
          otherXY[i] -= 6; // keep within inner ring
      }
      else if (otherXY[i] >= 6){
        otherXY[i] += (rotation*2);

        if (otherXY[i] >= 18)
          otherXY[i] -= 12;
      }
    }
  }

  //////////////////
  int findX(int i){
    int tempX = 0;

    if (i == 0)
     tempX = x+1;
    else if (i == 1)
     tempX = x+1;
    else if (i == 2)
     tempX = x;
    else if (i == 3)
     tempX = x-1;
    else if (i == 4)
     tempX = x-1;
    else if (i == 5)
     tempX = x;

    else if (i == 6)
     tempX = x+1;
    else if (i == 7)
     tempX = x+2;
    else if (i == 8)
     tempX = x+2;
    else if (i == 9)
     tempX = x+2;
    else if (i == 10)
     tempX = x+1;
    else if (i == 11)
     tempX = x;
    else if (i == 12)
     tempX = x-1;
    else if (i == 13)
     tempX = x-2;
    else if (i == 14)
     tempX = x-2;
    else if (i == 15)
     tempX = x-2;
    else if (i == 16)
     tempX = x-1;
    else //(i == 17)
     tempX = x;

    return(tempX);
  }

  int findY(int i){
    int tempY = 0;

    if (i == 0)
     tempY = y-1;
    else if (i == 1)
     tempY = y;
    else if (i == 2)
     tempY = y+1;
    else if (i == 3)
     tempY = y+1;
    else if (i == 4)
     tempY = y;
    else if (i == 5)
     tempY = y-1;

    else if (i == 6)
     tempY = y-2;
    else if (i == 7)
     tempY = y-2;
    else if (i == 8)
     tempY = y-1;
    else if (i == 9)
     tempY = y;
    else if (i == 10)
     tempY = y+1;
    else if (i == 11)
     tempY = y+2;
    else if (i == 12)
     tempY = y+2;
    else if (i == 13)
     tempY = y+2;
    else if (i == 14)
     tempY = y+1;
    else if (i == 15)
     tempY = y;
    else if (i == 16)
     tempY = y-1;
    else //(i == 17)
     tempY = y-2;

    return(tempY);
  }
}
```

```
int noOfCellsFromOsID(int id){
  int temp = 0;

  if (id <= 0)
    temp = 1;
  else if (id <= 1)
    temp = 2;
  else if (id <= 4)
    temp = 3;
  else if (id <= 14)
    temp = 4;

  return(temp);
}

void removeObject(int hX, int hY){
  boolean checkIt = false; // make sure shape hasn't already been deleted this time step
  int IDofDeletedObj = -1;

  // finds which object to remove when hBot is trying to remove an object
  for(int i = 0; i < noOfObjects; i++){
    // first check object's central x,y position
    for(int k = 0; k < 6; k++){
      if(find2Y(k,hY) == objects[i].y && find2X(k,hX) == objects[i].x){
        //println("remove " + i);
        if(objects[i].deleted == false){
          objects[i].deleted = true;
          halt = true;
          checkIt = true;
          IDofDeletedObj = objects[i].osID; // get the object shape type
        }
      }
    }

    for(int j = 0; j < objects[i].otherXY.length; j++){
      int position = objects[i].otherXY[j];
      int posY = objects[i].findY(position);
      int posX = objects[i].findX(position);

      for(int k = 0; k < 6; k++){
        if(find2Y(k,hY) == posY && find2X(k,hX) == posX){
          if(objects[i].deleted == false){
            //println("remove " + i);
            objects[i].deleted = true;
            halt = true;
            checkIt = true;
            IDofDeletedObj = objects[i].osID; // get the object shape type
          }
        }
      }
    }
  }
  if(checkIt){
    if(IDofDeletedObj == findThisShape){
      println("CORRECT SHAPE REMOVED");
      //outputCheckFind.print(timeSteps + ",");

      if(correctRemoved == 0){
        firstCorrectRemoved = timeSteps;
      }
      lastCorrectRemoved = timeSteps;

      noOfObjRemoved++;
      correctRemoved++;
    }
    else if(IDofDeletedObj == ignoreThisShape){
      println("INCORRECT SHAPE REMOVED");
      //outputCheckIgnore.print(timeSteps + ",");

      if(incorrectRemoved == 0){
        firstIncorrectRemoved = timeSteps;
      }
      lastIncorrectRemoved = timeSteps;

      noOfObjRemoved++;
      incorrectRemoved++;
    }
    else{
      println("ERROR Non-existant shape removed");
      exit();
    }
  }
  //println("NoOfObjectsRemoved: " + noOfObjRemoved);
}

//////////////////
int find2X(int i, int x){
  int tempX = 0;

  if (i == 0)
   tempX = x+1;
  else if (i == 1)
   tempX = x+1;
  else if (i == 2)
   tempX = x;
  else if (i == 3)
   tempX = x-1;
  else if (i == 4)
   tempX = x-1;
  else // (i == 5)
   tempX = x;
```

```
  return(tempX);
}

int find2Y(int i, int y){
  int tempY = 0;

  if (i == 0)
   tempY = y-1;
  else if (i == 1)
   tempY = y;
  else if (i == 2)
   tempY = y+1;
  else if (i == 3)
   tempY = y+1;
  else if (i == 4)
   tempY = y;
  else // (i == 5)
   tempY = y-1;

  return(tempY);
}

class ReadAndWrite{
  PrintWriter output;
  int numberOfLines;
  int numberOfColumns;

  float[] eachNumber;// = new int[numberOfColumns]; // length of numbers i.e. 100 would be 3, 4567 would be 4.

  float[][] collectedData;// = new int[numberOfLines][numberOfColumns]; //[number of lines][number of columns]

  ReadAndWrite(){
    numberOfLines = maxPopulation;
    numberOfColumns = 60;

    eachNumber = new float[numberOfColumns]; // length of numbers i.e. 100 would be 3, 4567 would be 4.

    collectedData = new float[numberOfLines][numberOfColumns]; //[number of lines][number of columns]

    collectedData[curSamp][0] = 20;
  }

  void readFile(){
    ////////////////////////////////////////////////////
    //INPUT
    String name;
    if(typeOfTest == 1)
      name = "baseline";
    else if(typeOfTest == 2)
      name = "random";
    else //(typeOfTest == 3)
      name = "geneticAlgorithm";

    String inputFile = "results/Find" + findThisShape + "Ignore" + ignoreThisShape + name + "Generation" + curGen
+ ".txt";

    // clean out each number
    for(int j = 0; j < eachNumber.length; j++){
      eachNumber[j] = 0;
    }

    // read from current file
    String lines[] = loadStrings(inputFile);

    //println(lines.length);

    for(int i = 0; i < numberOfLines; i++){
      // convert string into a char array
      char[] eachChar = new char[lines[i].length()];


      for (int j = 0; j < lines[i].length(); j++){
        eachChar[j] = lines[i].charAt(j);
      }

      // find the numbers
      int currentDigit = 0;
      int[] tempHold = new int[30];
      int tempHoldPosition = 0;
      int decimalAt = -1;
      int commaAt = 0; // must start at 0

      for (int j = 0; j < eachChar.length; j++){

        if(eachChar[j] == ','){
          commaAt = j; // update latest comma position

          //output collected
          int tempValue = 0;

          for(int k = 0; k < tempHoldPosition; k++){
            //println("tempHoldPosition = " + j + "    tempHold[j] = " + tempHold[j]);
            float temp = pow(10,decimalAt - k - 1); // 0.1, 1, 10, 100 etc based on where the digit is
            tempValue += (temp * tempHold[k]);
          }
          //println("Total " + tempValue);
          eachNumber[currentDigit] = tempValue;
          currentDigit++;
          tempHoldPosition = 0;
        }
        else{
```

```
          if(eachChar[j] == '.'){
            // there is a decimal point
            decimalAt = j - commaAt - 1; // find the relative position of decimal point
          }
          else{
            tempHold[tempHoldPosition] = (int)eachChar[j] - 48;
            tempHoldPosition++;
          }
        }
      }

      for (int j = 0; j < eachNumber.length; j++){
        collectedData[i][j] = eachNumber[j];
      }

      // problem reading first value, possible solution
      collectedData[i][0] = (int)eachChar[0] - 48;

    }

  }

  void writeFile(){
    // print to currentfile
    for(int i = 0; i < eachNumber.length; i++){
      output.print(collectedData[curSamp][i] + ",");
    }
    output.println("");
  }

  void writeAllFile(){
    // print to currentfile
    for(int i = 0; i < numberOfLines; i++){
      for(int j = 0; j < eachNumber.length; j++){
        output.print(collectedData[i][j] + ",");
      }
      output.println("");
    }
  }

  void startFile(){
    // set up text file
    String name;
    if(typeOfTest == 1)
      name = "baseline";
    else if(typeOfTest == 2)
      name = "random";
    else //(typeOfTest == 3)
      name = "geneticAlgorithm";

    String outputFile = "results/Find" + findThisShape + "Ignore" + ignoreThisShape + name + "Generation" +
curGen + ".txt";
    output = createWriter(outputFile);
  }

  void finishFile(){
    // Write a list of the variables used

    output.println(" ");

    for(int i = 0; i < genome[0].bitList.length; i++){
      output.print("Bit " + i + ",");
    }

    output.print("totalCorrect,maxCorrect,minCorrect,meanCorrect,medianCorrect");
    output.print(",totalIncorrect,maxIncorrect,minIncorrect,meanIncorrect,medianIncorrect");
    output.print(",totalSteps,maxSteps,minSteps,meanSteps,medianSteps");
    output.println(",meanFirstCorrectRemoved,meanLastCorrectRemoved,meanFirstIncorrectRemoved,meanLastIncorrectRe
moved");

    output.println(" ");
    output.println("Variables");
    output.println("Find Object Shape: " + findThisShape);
    output.println("Ignore Object Shape: " + ignoreThisShape);
    output.println("Max number of time-steps: " + maxTimeSteps);
    output.println("Number of test repeats: " + maxTests);
    output.println("Number of hBots: " + noOfBots);
    output.println(hour() + ":" + minute() + " " + day() + "/" + month() + "/" + year());


    // close current file
    output.flush(); // Writes the remaining data to the file
    output.close(); // Finishes the file

  }

  void convertDataForOutput(){
    // steps
    // total, max, min, mean, median
    float totalCorrect = 0;
    float totalIncorrect = 0;
    float totalSteps = 0;

    float totalFirstCorrectRemoved = 0;
    float totalLastCorrectRemoved = 0;
    float totalFirstIncorrectRemoved = 0;
    float totalLastIncorrectRemoved = 0;

    int maxCorrect = 0;
    int maxIncorrect = 0;
    int maxSteps = 0;
```

```
int minCorrect = 6;
int minIncorrect = 6;
int minSteps = maxTimeSteps;

int[] tempCorrect = new int[maxTests];
int[] tempIncorrect = new int[maxTests];
int[] tempSteps = new int[maxTests];

// calculate total time steps (number 2)
for (int i = 0; i < maxTests; i++){
  // find totals
  totalCorrect = totalCorrect + records[i][0];
  totalIncorrect = totalIncorrect + records[i][1];
  totalSteps = totalSteps + records[i][2];
  totalFirstCorrectRemoved = totalFirstCorrectRemoved + records[i][3];
  totalLastCorrectRemoved = totalLastCorrectRemoved + records[i][4];
  totalFirstIncorrectRemoved = totalFirstIncorrectRemoved + records[i][5];
  totalLastIncorrectRemoved = totalLastIncorrectRemoved + records[i][6];

  // find maximums
  if(records[i][0] > maxCorrect){
    maxCorrect = records[i][0];
  }
  if(records[i][1] > maxIncorrect){
    maxIncorrect = records[i][1];
  }
  if(records[i][2] > maxSteps){
    maxSteps = records[i][2];
  }

  // find minimums
  if(records[i][0] < minCorrect){
    minCorrect = records[i][0];
  }
  if(records[i][1] < minIncorrect){
    minIncorrect = records[i][1];
  }
  if(records[i][2] < minSteps){
    minSteps = records[i][2];
  }

  // copy across values to be sorted
  tempCorrect[i] = records[i][0];
  tempIncorrect[i] = records[i][1];
  tempSteps[i] = records[i][2];

}

float meanCorrect = totalCorrect/maxTests;
float meanIncorrect = totalIncorrect/maxTests;
float meanSteps = totalSteps/maxTests;

float meanFirstCorrectRemoved = totalFirstCorrectRemoved / maxTests;
float meanLastCorrectRemoved = totalLastCorrectRemoved / maxTests;
float meanFirstIncorrectRemoved = totalFirstIncorrectRemoved / maxTests;
float meanLastIncorrectRemoved = totalLastIncorrectRemoved / maxTests;

float medianCorrect = -1;
float medianIncorrect = -1;
float medianSteps = -1;

// sort arrays
tempCorrect = sort(tempCorrect);
tempIncorrect = sort(tempIncorrect);
tempSteps = sort(tempSteps);

// median correct
if(maxTests %2 == 0) {// even
  float tempA = tempCorrect[(maxTests/2)-1];
  float tempB = tempCorrect[(maxTests/2)];

  medianCorrect = (tempA+tempB)/2;
}
else{ // odd
  medianCorrect = tempCorrect[((maxTests-1)/2)];
}

// median incorrect
if(maxTests %2 == 0) {// even
  int tempA = tempIncorrect[(maxTests/2)-1];
  int tempB = tempIncorrect[(maxTests/2)];

  medianIncorrect = (tempA+tempB)/2;
}
else{ // odd
  medianIncorrect = tempIncorrect[((maxTests-1)/2)];
}

// median steps
if(maxTests%2 == 0) {// even
  int tempA = tempSteps[(maxTests/2)-1];
  int tempB = tempSteps[(maxTests/2)];

  medianSteps = (tempA+tempB)/2;
}
else{ // odd
  medianSteps = tempSteps[((maxTests-1)/2)];
}


// copy across values
// add genome bit list (length 41)
```

```
    for(int i = 0; i < genome[curSamp].bitList.length; i++){
      collectedData[curSamp][i] = genome[curSamp].bitList[i];
    }

    int counterTemp = genome[curSamp].bitList.length;

    collectedData[curSamp][counterTemp++] = totalCorrect;
    collectedData[curSamp][counterTemp++] = maxCorrect;
    collectedData[curSamp][counterTemp++] = minCorrect;
    collectedData[curSamp][counterTemp++] = meanCorrect;
    collectedData[curSamp][counterTemp++] = medianCorrect;

    collectedData[curSamp][counterTemp++] = totalIncorrect;
    collectedData[curSamp][counterTemp++] = maxIncorrect;
    collectedData[curSamp][counterTemp++] = minIncorrect;
    collectedData[curSamp][counterTemp++] = meanIncorrect;
    collectedData[curSamp][counterTemp++] = medianIncorrect;

    collectedData[curSamp][counterTemp++] = totalSteps;
    collectedData[curSamp][counterTemp++] = maxSteps;
    collectedData[curSamp][counterTemp++] = minSteps;
    collectedData[curSamp][counterTemp++] = meanSteps;
    collectedData[curSamp][counterTemp++] = medianSteps;

    collectedData[curSamp][counterTemp++] = meanFirstCorrectRemoved;
    collectedData[curSamp][counterTemp++] = meanLastCorrectRemoved;
    collectedData[curSamp][counterTemp++] = meanFirstIncorrectRemoved;
    collectedData[curSamp][counterTemp++] = meanLastIncorrectRemoved;
    //println("CounterTemp: " + counterTemp);

  }

}
int[][] records = new int[maxTests][7]; // 7 is for correctRemoved, incorrectRemoved, timeSteps, first and last
etc

void resetTest(){
  // hBots back to starting position

  for(int i = 0; i < noOfBots; i++){
    hBot[i].x = locationsX[i];
    hBot[i].y = locationsY[i];

    for(int k = 0; k < 6; k++){
      hBot[i].sensed[k] = 0;
    }
  }


  // shapes reset
  for(int i = 0; i < objects.length; i++){
    objects[i].deleted = false;

    objects[i].rotation = (int)(random(6));
    objects[i].rotation();
  }

  // Record Results for averaging later
  // correctRemoved,incorrectRemoved,timeSteps
  records[currentTest][0] = correctRemoved;
  records[currentTest][1] = incorrectRemoved;
  records[currentTest][2] = timeSteps-1; // need to minus one as time-step is added before this is run

  records[currentTest][3] = firstCorrectRemoved;
  records[currentTest][4] = lastCorrectRemoved;
  records[currentTest][5] = firstIncorrectRemoved;
  records[currentTest][6] = lastIncorrectRemoved;

  // other
  currentTest++;
  timeSteps = 0;
  noOfObjRemoved = 0;
  correctRemoved = 0;
  incorrectRemoved = 0;

  firstCorrectRemoved = maxTimeSteps;
  firstIncorrectRemoved = maxTimeSteps;
  lastCorrectRemoved = 0;
  lastIncorrectRemoved = 0;

  first = true; // check this
  //outputCheckFind.println("");
  //outputCheckIgnore.println("");
  //println("Current Test " + currentTest);
}

class Rule{
  int id = 0;
  int own = 0;
  int neighbourLow = 0;
  int neighbourHigh = 0;
  //int outputState = 0;

  int stateLevel = 0;

  Rule(int tempID){
    id = tempID;
    //outputState = id + 4;
  }

  void update(int i, int j, int k){
    own = i;
```

```
    neighbourLow = j;
    neighbourHigh = k;
    //stateLevel = levelTemp;
  }
}

// given an initial centre co-ordinate of the 1st object cell
// find the x,y coordinate for each spiral location
// spiral location starts with 0 in the centre and spirals out increasing in value
int[] locationsX = new int[(3*outerObjRing*outerObjRing) + (3*outerObjRing)+1];
int[] locationsY = new int[(3*outerObjRing*outerObjRing) + (3*outerObjRing)+1];
int noOfRings = outerObjRing;


void findXSpiralLocations(){
  int xTrack = ((cols-1)/2); // relative x-coordinate based on current spiral location value
  int addJ = 0;
  locationsX[0] = xTrack; // set first


  for (int i = 1; i <= noOfRings; i++){ // i is ring number
    //boolean xAA;  // first value +2 rather than +1
    int xA = i;       // number of times to +1 to x-tracker
    int xB = i;   // number of times to +0 to x-tracker
    int xC = 2*i; // number of times to -1 from x-tracker
    int xD = i;   // number of times to +0 to x-tracker
    int xE = i;       // number of times to +1 to x-tracker

    for (int j = 1; j <= xA; j++){
      xTrack = xTrack+1;
      locationsX[j + addJ] = xTrack;
    }

    for (int j = xA+1; j <= xA + xB; j++){
      xTrack = xTrack;
      locationsX[j + addJ] = xTrack;
    }

    for (int j = xA + xB + 1; j <= xA + xB + xC; j++){
      xTrack = xTrack-1;
      locationsX[j + addJ] = xTrack;
    }

    for (int j = xA + xB + xC + 1; j <= xA + xB + xC + xD; j++){
      xTrack = xTrack;
      locationsX[j + addJ] = xTrack;
    }

    for (int j = xA + xB + xC + xD + 1; j <= xA + xB + xC + xD + xE; j++){
      xTrack = xTrack+1;
      locationsX[j + addJ] = xTrack;
    }

    addJ = xA + xB + xC + xD + xE + addJ;
  }
}

void findYSpiralLocations(){
  int yTrack = ((rows-1)/2); // relative y-coordinate based on current spiral location value
  locationsY[0] = yTrack; // set first

  int addJ = 0;

  for (int i = 1; i <= noOfRings; i++){ // ring number
    int yA = 1;    // number of times to -1 from y-tracker
    int yB = i-1;        // number of times to +0 to y-tracker
    int yC = i*2;  // number of times to +1 to y-tracker
    int yD = i;    // number of times to +0 to y-tracker
    int yE = i*2;  // number of times to -1 from y-tracker

    //println(yA +" "+ yB +" "+ yC +" "+ yD +" "+ yE);

    for (int j = 1; j <= yA; j++){
      yTrack = yTrack-1; // -1 to tracker yA times
      locationsY[j + addJ] = yTrack;
    }

    for (int j = yA+1; j <= yA + yB; j++){
      yTrack = yTrack; // +0 to tracker yB times
      locationsY[j + addJ] = yTrack;
    }

    for (int j = yA + yB + 1; j <= yA + yB + yC; j++){
      yTrack = yTrack+1; // +1 to tracker yC times
      locationsY[j + addJ] = yTrack;
    }

    for (int j = yA + yB + yC + 1; j <= yA + yB + yC + yD; j++){
      yTrack = yTrack;    // +0 to tracker yD times
      locationsY[j + addJ] = yTrack;
    }

    for (int j = yA + yB + yC + yD + 1; j <= yA + yB + yC + yD + yE; j++){
      yTrack = yTrack-1;  // -1 from tracker yE times
      locationsY[j + addJ] = yTrack;
    }

    addJ = yA + yB + yC + yD + yE + addJ;
  }
}

class StateRules{
```

```
Rule[] rule;

int[] possibleList = {1,2,3,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0}; // 0 is empty
int nMin = 1; // the lowest in the possible list
int nMax = max(possibleList); // the highest in the possible list
int countRule = 4; // start at 4 because rule 1,2 and 3 are pre-defined

int[][][] ruleList = new int[21+1][21+1][21+1]; // own, low, high (rules start at one)


int maxVal = 22;

int levelOneMax = 3; // maximum number of side contacts permisable for single hBot
int levelTwoMax = 21; // max state for centre hBot of three
int levelThreeMax = 264; // max state for centre hBot of five

StateRules(){
  rule = new Rule[264+1]; // plus 1 as rule list starts at 1 and not 0


  for (int i = 4; i < rule.length; i++){
    rule[i] = new Rule(i);
  }

  int own = 1; // the hBots current state
  int nLow = 1; // the lowest neighbour
  int nHigh = 1; // the highest value neighbour

  while(countRule < rule.length){
    boolean highCheck = true;
    boolean lowCheck = true;

    if (nLow <= nHigh){
      rule[countRule++].update(own,nLow,nHigh);
      println("[" + own + "][" + nLow + "][" + nHigh + "]  new: " + rule[countRule-1].id);
    }

    nHigh++;

    while(highCheck){
      if (nHigh > nMax){
        nLow++;
        nHigh = nLow;
        highCheck = false;
      }
      else{
        // is nHigh on the possibleList
        boolean onList = false;
        for (int i = 0; i < possibleList.length; i++){
          if(possibleList[i] == nHigh){
            onList = true;
          }
        }
        if (!onList){
          nHigh++;
        }
        else{
          highCheck = false;
        }
      }
    }

    while(lowCheck){
      if(nLow > nMax){
        own++;

        updatePossibleList(own);

        nLow = nMin;
        nHigh = nMin;
        lowCheck = false;
      }
      else{
        // is nLow on the possibleList
        boolean onList = false;
        for (int i = 0; i < possibleList.length; i++){
          if(possibleList[i] == nLow){
            onList = true;
          }
        }
        if (!onList){
          nLow++;
        }
        else{
          lowCheck = false;
        }
      }
    }
  }


  // create rules reference list
  // set all rules to 0
  for(int i = 0; i < 22; i++){
    for(int j = 0; j < 22; j++){
      for(int k = 0; k < 22; k++){
        ruleList[i][j][k] = -100;
      }
    }
  }
```

```
    for(int i = 0; i < rule.length; i++){
      if(rule[i] != null){
        ruleList[rule[i].own][rule[i].neighbourLow][rule[i].neighbourHigh] = rule[i].id;
      }
      else{
        int a = 0;
        //println("This is null " + i);
      }
    }


  } // end of setup

  ////////////////////////////////////////////////////////////////////////////
  int returnNewState(int tOwn, int tLow, int tHigh){
    if(ruleList[tOwn][tLow][tHigh] != -100){
      return(ruleList[tOwn][tLow][tHigh]);
    }
    else{
      println("ERROR with returnNewState");
      exit();
      return(0);
    }
  }

  ////////////////////////////////////////////////////////////////////////////
  void updatePossibleList(int ownTemp){
    int pointer = 0;
    if (ownTemp >= 4){

      // reset possible list
      for (int i = 0; i < possibleList.length; i++){
        possibleList[i] = 0;
      }

      // find root state, and neighbour values
      int root = rule[ownTemp].own;
      int nOne = rule[ownTemp].neighbourLow;
      int nTwo = rule[ownTemp].neighbourHigh;

      // check rules to see if the root matches either neighbour of the other rule
      for (int i = 4; i < rule.length; i++){
        if(root == rule[i].neighbourLow || root == rule[i].neighbourHigh){
          if (nOne == rule[i].own || nTwo == rule[i].own){
                // this is an acceptable option
                possibleList[pointer++] = rule[i].id; // add to list
          }
        }
      }

      nMin = possibleList[0];
      nMax = max(possibleList); // the highest in the possible list

      if (nMin == 0){
        println("ERROR min can't be zero");
        exit();
      }
      if (possibleList[possibleList.length-1] != 0){
        println("ERROR last in list must be zero");
        exit();
      }
    }
  }
}

int findStateAtSameLevel(int levelRequired, int stateAt){
  int counter = 0;
  boolean leave = false;


  do{
    for(int i = 0; i < stateRules.maxVal; i++){
       for(int j = 0; j < stateRules.maxVal; j++){
         for(int k = 0; k < stateRules.maxVal; k++){
           if(stateRules.ruleList[i][j][k] != -100){ // undefined rules are minus one-hundred
             if(stateRules.ruleList[i][j][k] == stateAt){
               stateAt = i;
             }
           }
         }
        }
      }
     counter++;
     if(counter > 10){
       leave = true;
     }
  } while((findLevelFromState(stateAt) != levelRequired) && leave == false);

  if (leave){
      println("Error Stuck In Do While Loop Find State at Same Level");
      exit();
  }

  return(stateAt);
}

int findLevelFromState(int stateAt){
  if (stateAt <= 0)
    return(0);
  else if (stateAt <= stateRules.levelOneMax)
    return(1);
```

```
  else if (stateAt <= stateRules.levelTwoMax)
    return(2);
  else if (stateAt <= stateRules.levelThreeMax)
    return(3);

  else{
    println("Error in findLevelFromState");
    exit();
    return(0);
  }

}
```