Honours Project Report: Evolving robot morphologies in a collective foraging task using NEAT-M

Jamie Hewland - HWLJAM002

October 2014

Supervisor: Geoff Nitschke

	Category	Min	Max	Chosen
1	Requirement Analysis and Design	0	20	0
2	Theoretical Analysis	0	25	0
3	Experiment Design and Execution	0	20	15
4	System Development and Implementation	0	15	10
5	Results, Findings and Conclusion	10	20	15
6	Aim Formulation and Background Work	10	15	15
7	Quality of Report Writing and Presentation	10	10	10
8	Adherence to Project Proposal and Quality of Deliverables	10	10	5
9	Overall General Project Evaluation	0	10	10
	TOTAL	80	80	80

Abstract

The neuro-evolution method know as Neuro-Evolution of Augmenting Topologies (NEAT) (Stanley and Miikkulainen, 2002) is extended to include the evolution of robot morphologies. This new method is called Neuro-Evolution of Augmenting Topologies and Morphologies (NEAT-M). We use the collective foraging task in a simulated environment to test the performance of evolved robotic agents. Of interest to our research is the level of cooperation which agents exhibit in this task. Our experiments show that cooperation is evolved using NEAT-M but that the addition of morphology evolution does not necessarily increase the level of cooperation.

Contents

List of Figures

List of Tables

1 Introduction

Evolutionary Robotics (ER) (Nolfi and Floreano, 2000) uses the concepts of Evolutionary Computation (EC) (Eiben and Smith, 2003) to create robotic agents that exhibit some form of Artificial Intelligence (AI) (Russell and Norvig, 2003). EC methods typically create control systems that exhibit a desired *behaviour*. Neuro-Evolution (NE) (Miikkulainen, 2010) is an EC method that has been shown to achieve high performance in control tasks with incomplete or noisy information which makes it a good fit for robotic control tasks.

We will explore some methods that evolve not only robot controllers but also the sensor and actuator configuration or *morphology* of robotic agents.

Robotic agents are typically tested in a simulated benchmark task in order to assess their performance. The simulation task that will be the focus of this research is the collective foraging or construction task which requires agents to seek out resources scattered about the environment and then to return those resources to a certain fixed destination.

By involving multiple agents concurrently in this foraging task, we hope to discover some collective behaviour that emerges through their interaction with one another. Such interaction could involve *cooperation* between agents. We aim to study the level of cooperation that evolves in the simulation and what effect our agent evolution scheme has on this level of cooperation.

2 Background

Artificial Intelligence (AI) (Russell and Norvig, 2003) provides the means to create autonomous, computer-driven systems that are able to solve complex problems. A number of techniques have been developed to create computer algorithms or procedures that are able to display some intelligent behaviour. Some of these techniques are based on simplified theories of Darwinian evolution and are known as *evolutionary algorithms* (EAs) (Bäck, 1996). EAs find useful algorithms or algorithm parameters by iteratively selecting and refining from a pool (or *population*) of candidate solutions. *Neuro-Evolution* (NE) takes these biologically-inspired concepts further by evolving *Artificial Neural Networks* (ANNs), a computational model inspired by the central nervous system.

Multiple robotic agents are placed in an environment and made to complete a task that will be used to measure the robots' performance. Such a system is often called a *multi-agent system* (MAS) (Wooldridge, 2009). One approach to deriving collective behaviour in a MAS is to have group behaviour such as cooperation emerge via the local interactions of robots.

Such tasks are typically carried out in a simulated virtual environment while the agent controllers are being developed. The controllers can then be "transplanted" into physical robots in a real-world simulation. Robotics simulations featuring controllers developed by EAs fall under the field of *Evolutionary Robotics* (ER) (Nolfi and Floreano, 2000).

We will begin by explaining the development of a control system for the agents in the simulation, provide some background on MASs, and finally cover the details of the simulation environment.

2.1 AI for Controller Design

2.1.1 Artificial Intelligence and Machine Learning

The field of AI was born out of the desire to create computer systems that are useful without explicit instruction from human controllers. Systems that do not follow a set of predetermined rules or some mathematical function but instead *behave* with some degree of autonomy are often necessary in order to find efficient solutions to challenging problems.

Machine Learning (ML) (Bishop, 2006) is a subfield of AI that is concerned with the construction of systems that can learn from data. ML can be used to produce behaviours either from a body of training data or through refinement in a learning environment.

2.1.2 Controller Design

When agents such as robots operate in a real-world environment, their controllers need to be able to account for a variety of circumstances and potentially noisy information. For a small set of situations it is possible for a hand-made procedure or set of heuristics to be created that will produce the desired behaviour. As the complexity of the task increases, however, it becomes increasingly difficult for a human to create controllers with good performance. For this reason, ML methods are often applied to controller design, so as to automate controller design for a given task.

2.1.3 Machine Learning methods

There are three main approaches to learning: supervised, unsupervised and reward-based learning. Each approach is defined by what kind of feedback is provided to the learning algorithm.

In supervised learning, an algorithm is "trained" by providing correct feedback for sets of inputs. Unsupervised learning is the opposite, no feedback is provided to the algorithm. In reward-based learning, the feedback to the algorithm is in the form of a *reward*, some value that signifies the quality of the algorithm. Supervised learning methods are generally not applied to MASs as the inherent complexity of the interactions between agents makes it challenging to provide feedback for every possible situation. Given this, the majority of work involving MASs has focused on reward-based and unsupervised methods.

Reward-based learning can be split into two subsets: *reinforcement learning* and *stochastic search methods*. Reinforcement Learning (RL) (Barto, 1998) estimates some value function and is useful in situations where feedback (in the form of penalties or rewards) can be provided after some sequence of actions performed in the environment. RL methods typically rely on dynamic programming concepts in order to estimate the value (or *utility*) of taking a given *action* from a certain *state* in the environment. This value is estimated based on previously received rewards. Due to the large state space created by the interactions between agents in MASs, it becomes challenging to map states to utilities using traditional RL methods. Still, there have been some uses of RL in MASs such as (Tan, 1993; Wiering et al., 1999; Lauer and Riedmiller, 2000).

Stochastic search methods directly learn behaviours without seeking to estimate a value function. Most literature in the stochastic search field concentrates on evolutionary computation. Evolutionary Computation (EC) (Eiben and Smith, 2003) has been shown to be an effective strategy to generate control parameters and decision rules for agents exhibiting collective behaviour (Baldassarre et al., 2003).

An *Evolutionary Algorithm* (EA) is a subfield of EC that involves the adaptation of solutions using processes based on abstract concepts of Darwinian evolution. Collections of candidate solutions (often called *populations* of *individuals*) are refined through processes of selection, recombination and mutation at each iteration (or "generation") of the algorithm. The processes that are applied to the solutions are known as *genetic operators*. At each iteration, the population is refined until some termination condition is reached. Usually this occurs when either a certain amount of time has elapsed or a sufficiently fit individual has been found. The basic structure of this iterative process can be seen in figure 1.

There are several different classes of evolutionary algorithms, each distinguished by the data structure used to represent the solutions. *Genetic algorithms* (GAs) (Holland, 1975) and *Evolutionary Strategies* (ESs) (Eigen, 1973) are usually applied to search multi-dimensional state spaces. The former typically uses a bit-string to represent individuals while the latter makes use of real-valued vectors. *Evolutionary programming* (EP) (Fogel

```
INITIALISE population with random candidate solutions
EVALUATE each candidate
REPEAT UNTIL (TERMINATION CONDITION is satisfied)
        1. SELECT parents
        2. RECOMBINE pairs of parents
        3. MUTATE the resulting offspring
        4. EVALUATE new candidates
        5. SELECT individuals for the next generation
```

Figure 1: The general structure (or pseudo-code) for an evolutionary algorithm. Adapted from (Yao, 1999).

et al., 1966) evolves state tables for finite state machines (FSMs). *Genetic programming* (GP) (Koza, 1992) evolves actual computer programs by refining their expression trees.

2.2 Neuro-Evolution (NE)

Neuro-Evolution (NE) (Miikkulainen, 2010) applies evolutionary algorithms to the adaptation of artificial neural networks. Thus, to explain NE it is necessary to explain the workings of artificial neural networks.

2.2.1 Artificial Neural Networks

Artificial Neural Networks (ANNs) (McCulloch and Pitts, 1943) were inspired by the biological brains of organisms in nature. In an ANN, artificial nodes are connected together via links to form a network that models a biological neural network. The nodes in an ANN are analogous to neurons in the brain and the terms *node* and *neuron* are used interchangeably in ANN literature. The links between the nodes are similar to synapses between neurons in the brain although the term *synapse* is used less often in the context of ANNs.

The individual nodes function much like neurons in the brain: each has a number of inputs and a single output that "fires" a signal depending on the values of the inputs. Each node in an ANN has a weight assigned to each of its inputs. A weighted sum is calculated with the *transfer function* as shown in the equation,

$$net = \sum_{i \in I} w_i x_i \tag{1}$$

where w_i is the weight for the ith input, and x_i is the ith input.

The result of the weighted sum is then fed to an *activation function.* This is typically a differentiable Sigmoid function such as the logistic function shown in the equation,

$$\varphi(t) = \frac{1}{1 + e^{-\mu t}} \tag{2}$$

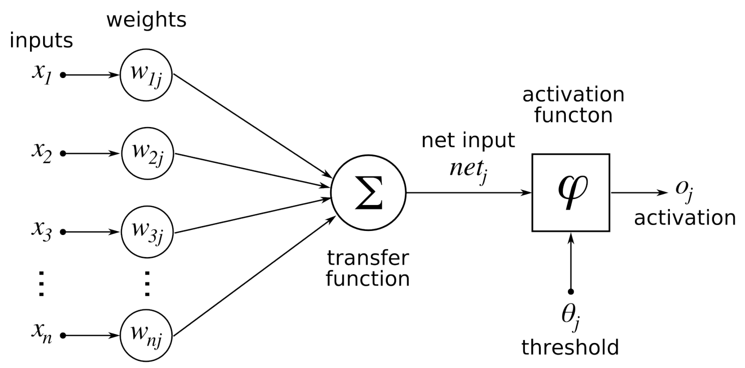


Figure 2: The internal structure of the jth neuron or "node" (Burgmer, 2005).

where t is the weighted sum from the transfer function, and μ is the slope constant. The overall structure of a single neuron can be seen in figure 2.

Multiple nodes are connected together to form a network. In the simplest networks, input nodes receive external stimulus and feed directly into the output nodes. More complex networks place additional nodes between the inputs and outputs. Each level of indirection between the input and output nodes is known as a *hidden layer*. Nodes in hidden layers are known as *hidden nodes*. Any network with at least one hidden layer is known as a *multi-layer network*.

Another important concept with regard to ANNs is the concept of *connectedness* which describes how nodes in a network are connected to one another. A *fully-connected* network is one in which all the nodes in a given layer are connected to every node in the next layer. An example of a fully-connected ANN with a single hidden layer is shown in figure 3. A connection from a node in a certain layer to a node in the same layer or a previous layer is known as a *recurrent connection*.

Cybenko (1989) (and later, Hornik (1991)) showed that a fully-connected network with a single hidden layer is a *universal function approximator* in that it is able to approximate any continuous function.

This behaviour is useful for complex control tasks as it becomes possible to approximate complex control functions that map a set of sensor inputs to a set of control outputs.

2.2.2 Weight adaptation methods

Artificial neural networks provide a framework for building a control system with adaptive behaviour for robotic agents. It is still necessary to design an appropriate structure for the network as well as decide on the weights for each input of each node. Various methods have been developed in order to "learn" these parameters. Neuro-Evolution applies evolutionary algorithms to adapt ANNs to solve problems.

A key benchmark for the performance of a control strategy is the inverted pendulum problem. In this problem, a controller must balance a pendulum which has its centre of mass above its pivot point. The problem can have different configurations with varying complexity and is often considered a surrogate for more complex real world control problems. In its simplest form, it is typically implemented as the "cart and pole" problem where a pole (the pendulum) is mounted to a pivot point on a cart that can be moved

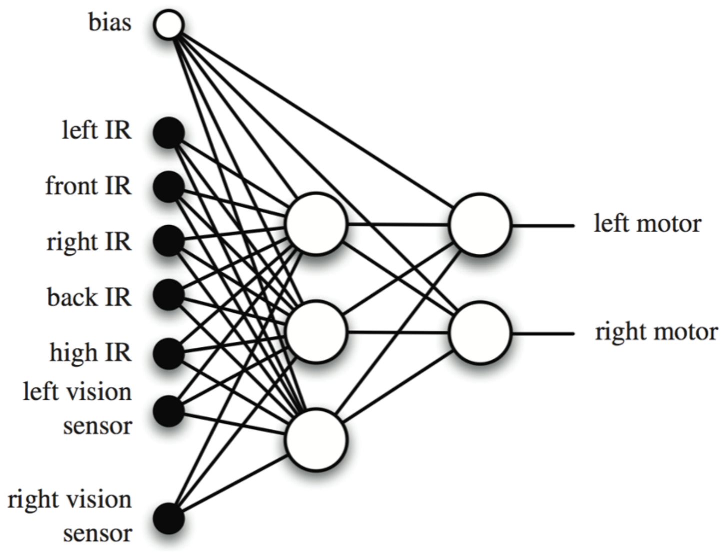


Figure 3: The neural network architecture used by Waibel et al. (2009) to control their robot agents. The inputs to the network can be seen on the left, while the outputs to the motor drives are on the right. The nodes between the inputs and outputs are the *hidden nodes*. The network is *fully connected* as each node connects to every node in the next layer.

horizontally. The control system must control the movement of the cart so that the pole is kept upright. Gomez et al. (2006) showed a NE method that outperformed a variety of other ML methods on several different variations of the inverted pendulum problem.

NE in its simplest form features genes that consist of the weights for nodes in a neural network (Yao, 1999). In this configuration, the weights of the neural network are refined towards their optimal values each generation of the evolutionary algorithm.

The most basic NE approaches use a genetic algorithm. Early systems used a bitstring representation of the concatenated connection weights of each node (Caudell and Dolan, 1989; Srinivas and Patnaik, 1991). Other systems used a real-number representation of the weights in the neural network (Montana and Davis, 1989; Fogel et al., 1990). These weight adaptation methods assume that the architecture of the ANN is predefined and fixed during the evolution of connection weights.

2.2.3 Topology Adaptation Methods

Adapting the weights of the ANN is only one half of the problem. Designing the structure or *topology* of the network itself is often a non-trivial task. The number of input and output nodes are generally defined by the inputs and outputs of the system but most ANNs feature at least some hidden nodes. Deciding on the number of hidden nodes needed and whether or not extra hidden layers are needed is not a simple task. Several methods have been developed to evolve both the topologies and weights of networks. Stanley and Miikkulainen (2002) refer to these networks as *Topology and Weight Evolving Artificial Neural Networks* (TWEANNs).

The answer to the question of whether evolving both weights and topology is advantageous remains elusive. Since a fully connected ANN has been proven to be a general function approximator (Cybenko, 1989), it seems that expending effort to test extra topologies would be a waste.

Gruau et al. (1996) argued for the evolution of both topology and weights by claiming that evolving the structure of a network saves the time spent by humans finding a particular topology that is suited to the problem at hand. Kazuhiro Ohkura and Matsumura (2011) showed a TWEANN that outperformed an algorithm without adapting topologies in a cooperative food-foraging problem. As will be discussed later, the food-foraging problem is relevant to this research and thus the focus of this research will be on TWEANNs.

Evolving the structure of networks in addition to the weights presents several technical challenges. For example, it becomes necessary to determine whether two parent genes are compatible when recombining networks with different topologies. Also, it is difficult to determine how much testing is necessary to prove the effectiveness of a given topology.

Yao and Liu (1997) developed EPNet, one of the early TWEANN approaches. EPNet is based on evolutionary programming. It applies a sequence of different mutation operations to every ANN each generation. The mutation operators either add or delete connections or nodes in the network or adapt the connection weights. After each operator is applied the ANN is retested and if it has improved no further mutation operations are applied. This means that EPNet requires a large population size and is relatively computationally expensive. EPNet completely avoids the problem of trying to combine incompatible genes by not using any recombination operators.

One of the more well-known systems for evolving neural networks was proposed by Stanley and Miikkulainen (2002), called *Neuro-Evolution of Augmenting Topologies* (NEAT). This system has been used for a variety of tasks and a number of derivative techniques have been developed such as *rtNEAT* (Stanley et al., 2005), *HyperNEAT* (Stanley et al., 2009) and *MM-NEAT* (Schrum and Miikkulainen, 2014).

Stanley and Miikkulainen isolated three core challenges with TWEANNs:

1. Disparate topologies are often difficult or impossible to recombine.
2. New genes which introduce changes to the topology of a network often require a few generations to properly optimise the new topology.
3. TWEANNs should not produce networks with unnecessarily complex networks.

NEAT proposes several solutions to these problems. NEAT protects topological innovation through *speciation*. Speciation, also known as *niching*, forces genomes of a certain *species* to "share" fitness with each other. Genomes of a certain species are similar by the measure used to find similar genomes for crossover. Speciation essentially splits the population into several smaller sub-populations of similar genomes. These sub-populations are then individually selected from. A *threshold speciation* scheme is used, where a value, δ, is assigned to the structural difference between two genomes and if the value is larger than a threshold value, δ_t, the two genomes are said to be of different species. Calculating the structural difference between two genomes, δ, is achieved using the equation,

$$\delta = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 \cdot \overline{W} \tag{3}$$

where c_1, c_2, and c_3 are constant coefficients, N is the number of genes in the larger genomes, E is the difference in length of the two genomes or the *extraneous* difference, D is the number of links with differing innovations or the *disjoint* difference, and $\overline{W}$ is the vector of the difference in weight values.

This equation compares the connection genes of the two genomes by counting the number of genes, D, that do not share the same innovation number, the difference in length, E, of the two genomes, and the difference in the link weight values, $\overline{W}$, of the genes with matching innovation numbers. Thus, NEAT's speciation of genomes takes into account both the topology of the resulting networks as well as the network's link weights.

In many TWEANN systems the initial population is initialised with random topologies in order to ensure topological diversity. A common aim for TWEANNs is to produce minimal topologies, it is desired that networks should only be as complex as is necessary. In order to achieve this aim, NEAT takes a different approach when initialising the population. NEAT starts with a population of networks with "minimal" topologies and creates more complex networks through the evolutionary process. Stanley describes this process as *complexification* (Stanley, 2004).

Another problem that is common to many forms of NE (not only TWEANNs) is known as the *Competing Conventions Problem* (Whitley et al., 1990; Radcliffe, 1993), also known as the *Permutations Problem.* Often there is more than one way to express a solution to a weight optimisation problem with a neural network. A network with n hidden nodes can be represented in $n!$ different ways by rearranging the order of the hidden nodes. This concept is illustrated in figure 4 where two possible networks are shown that compute the same function but are represented differently. Additionally, when these two networks are recombined, the "information" in one of the hidden nodes is lost. Recombining the hidden node representations $[A, B, C]$ and $[C, B, A]$ could result in the new network $[C, B, C]$ which has lost the information in A while duplicating the information in C. The competing conventions problem is complicated further when network topology is also evolved. TWEANN networks can represent similar solutions using different topologies or even with genomes of different lengths, making it difficult to determine when two networks compute the same function.

NEAT tackles the competing conventions problem using *historical markings* in its genotype representation. NEAT tracks an "innovation number" for each link in the network. Common historical markings on two genes signify that they share an origin and are therefore similar and can be recombined. Figure 5 shows the genome encoding scheme used by NEAT.

Stanley and Miikkulainen (2004) have also shown NEAT to be suited to the adaptation of robotic control schemes.

2.3 Multi-Agent Systems

2.3.1 Definition

Multi-agent systems (Wooldridge, 2009) are best defined as a subfield of *distributed systems*, where a collection of entities cooperate in order to solve problems. The combination

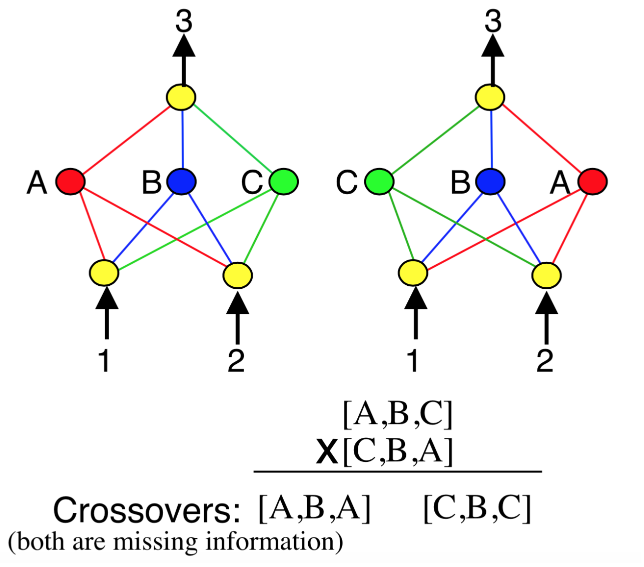


Figure 4: The two networks depict the same effective function while their hidden nodes appear in a different order (Stanley and Miikkulainen, 2002).

of the fields distributed systems and artificial intelligence is known as *distributed artificial intelligence* (DAI) (Ferber, 1999). DAI itself can be divided into two areas, *distributed problem solving* and *multi-agent systems* (MASs). Multi-agent systems are concerned with the manner in which agents, behaving with some degree of autonomy, interact to create some kind of joint behaviour.

Panait and Luke (2005) attempt to provide a broad definition of the concepts of *agent* and *multi-agent* in their review of the field of cooperative multi-agent learning. They define an agent as a computational mechanism that exhibits a high degree of autonomy, performing actions in its environment based on information (sensor inputs) received from the environment. The definition of a multi-agent environment is more complex than merely the presence of multiple agents. Panait and Luke provide the additional constraint on the environment that agents may not at any given time know everything about the world that other agents know. This includes the internal state of the other agents themselves. It is argued that this constraint is important because it ensures that agents cannot simply act in sync, as though they were merely part of a single master controller.

2.3.2 Multi-Agent Learning

Once again, Panait and Luke (2005) define multi-agent learning in broad terms; it is the application of machine learning to problems involving multiple agents. Additionally, they argue that the field is worth separating from ordinary machine learning for two reasons. Firstly, because multi-agent learning deals with problem domains involving multiple agents, the search space can be very large. Due to the interaction of those agents small changes in learned behaviour can result in unpredictable changes in the resulting macro-level ("emergent") properties of the simulation as a whole. Secondly, multi-agent learning sometimes involves multiple learners, each learning and adapting in the context of others. This introduces complexity to the learning process which is difficult to model.

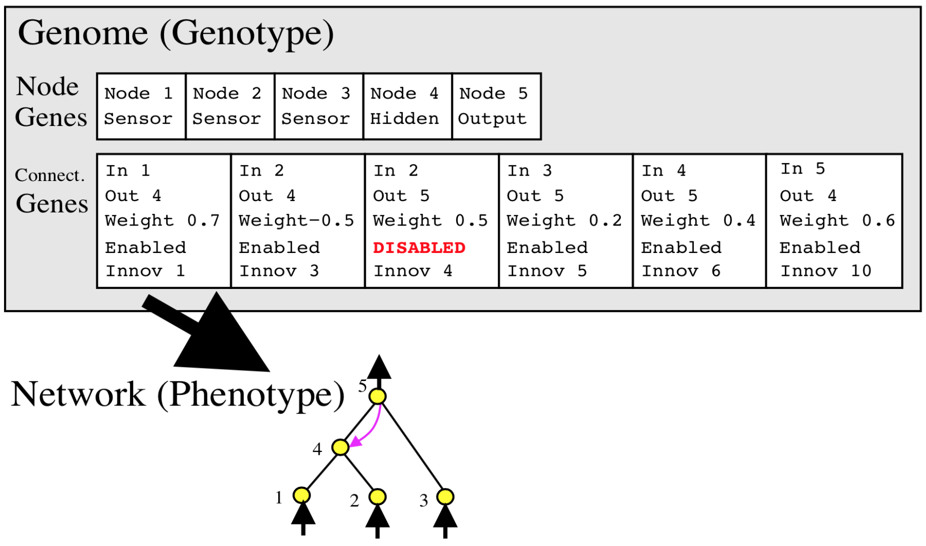

Genome (Genotype)
Node Genes
Node 1 Sensor
Node 2 Sensor
Node 3 Sensor
Node 4 Hidden
Node 5 Output
Connect. Genes
In 1 Out 4 Weight 0.7 Enabled Innov 1
In 2 Out 4 Weight-0.5 Enabled Innov 3
In 2 Out 5 Weight 0.5 DISABLED Innov 4
In 3 Out 5 Weight 0.2 Enabled Innov 5
In 4 Out 5 Weight 0.4 Enabled Innov 6
In 5 Out 4 Weight 0.6 Enabled Innov 10
Network (Phenotype)
5
4
1
2
3

Figure 5: The NEAT genome is composed of a list of node genes and a list of connection (link) genes. Each connection gene has an *innovation number* which is used as a historical marking (Stanley and Miikkulainen, 2002).

It is also necessary to define what is meant by *cooperative* multi-agent learning. There are many different approaches to multi-agent learning and hence it is difficult to provide a strict definition of what is meant by cooperation as there are many differing implementations. At the very least, we can define cooperation in terms of the *intent* of the experimenter. It is sufficient for the problem and the learning system to be constructed with the intention of encouraging cooperation among the agents, even if cooperation is not necessarily achieved.

2.3.3 Multi-Agent Teams

Waibel et al. (2009) distinguish between types of multi-agent teams along two axes: the genetic composition of the team and the level of selection within those teams. The genetic composition of a team can be seen to be either *homogeneous* (every agent in the team is a "clone" of a single genome), or *heterogeneous* (every agent features a unique genome). Team selection may operate either on teams (team-level selection) or on individuals (individual-level selection).

Distinguishing between these different types of teams is not always clear cut. For example, Stanley et al. (2005) used "squads" of agents within groups that clustered genetically similar agents together, resulting in partially heterogeneous teams. Another example of hybrid team learning is A. Hara (1999), who suggest an automated grouping technique called *Automatically Defined Groups* (ADG). ADG aims to automatically find the optimum number of groups and their compositions.

Waibel et al. consistently found homogeneous teams with individual-level selection to perform better than heterogeneous teams with team-level selection. They applied a genetic algorithm-based NE approach to developing controllers for multi-agent robots performing a cooperative foraging task.

2.3.4 Co-Evolutionary Algorithms (CEAs)

Panait and Luke's (2005) review of MASs focuses specifically on cooperative systems. They are careful to distinguish between the two types of co-evolutionary algorithms: *competitive* and *cooperative* co-evolution. In competitive co-evolution, individuals benefit at the expense of their peers. On the other hand, in cooperative co-evolution, individuals succeed or fail together. In general, in a CEA the fitness of an individual is *context-sensitive* as it is dependent on the individual's interaction with other individuals in the population.

There are a few strategies for implementing cooperation in an optimisation problem. One approach is to split the problem into sub-components and then to assign each sub-component to a separate portion of the population. Potter and De Jong (1994, 2000) have done extensive work on this approach.

Another approach to cooperative co-evolution is to create tasks that require multiple agents to complete. For instance, in Waibel et al.'s (2009) experiment, multiple agents needed to work together in order to push large objects to a target destination. This approach is attractive because developing cooperation does not complicate the evolutionary algorithm implementation. Cooperation is *required* by the environment in order for the genome to achieve high task performance (fitness).

2.4 Evolutionary Robotics

2.4.1 Robots

Several studies have used either virtual or real robots to perform multi-agent simulations. These implementations provide a testbed for multi-agent systems. Showing that a multi-agent controller is robust and able to adapt to noisy conditions is an important step in showing the real-world usefulness of such a controller.

Often these robots feature multiple short-range infra-red (IR) proximity detectors Waibel et al. (2009); Quinn (2001); Quinn et al. (2003); Nolfi et al. (1994). This is probably due to a number of factors. IR detectors are generally accurate at detecting other objects while also being relatively low-definition and so simplifying the design of the control systems that use the detectors as inputs. They are also relatively inexpensive devices. Other common sensor types include ultrasonic sensors and low-resolution camera sensors. A number of different robots featured in various experiments can be seen in figure 6.

Robotics are an attractive approach for many experimenters as the real-life implementation of an algorithm shows its robustness and relevance. The use of robotics also allows experimenters to test realistic inputs and outputs with their algorithms.

One of the most popular models of robot in these experiments is the Khepera robot by the K-Team Corporation. This robot is currently available in its third generation, the Khepera III. Kheperas have been used in (Nolfi et al., 1994; Stolzmann, 1999; Godjevac and Steele, 1999; Low et al., 2002; Buason et al., 2005) and many more. These robots, however, are expensive devices and so many experimenters have instead simulated these robots in a virtual environment.

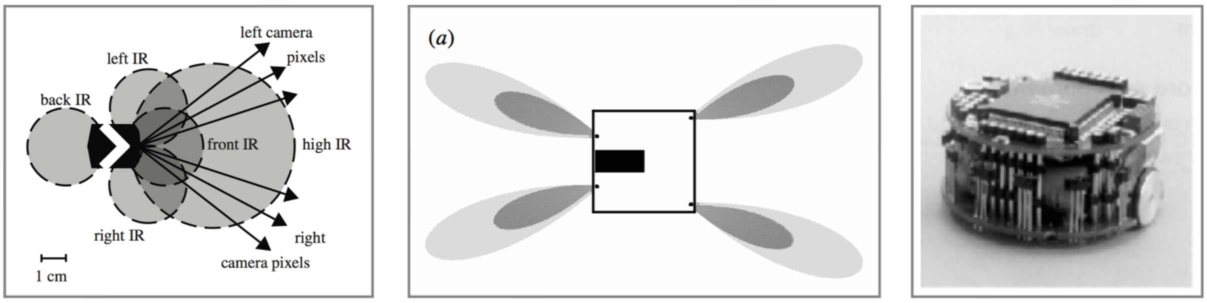


Figure 6: From left to right — the sensor configuration of Waibel et al.'s (2009) robots, the range of the infra-red sensors of Quinn et al.'s (2003) robots, and the Khepera robots used by Nolfi et al. (1994)

There are several virtual simulator platforms available, often called *agent-based simulation models* (ABMs). Railsback et al. (2006) performed an extensive review and performance test of ABMs. Examples of available platforms include NetLogo, Swarm, MASON and Repast. MASON and Repast are the two systems that have undergone the most development in the years since the study of Railsback and it is possible to configure both using Java.

2.4.2 Morphology evolution

The configuration of sensors and actuators on a robot is called the *morphology* of the robot. When using simulated robots it becomes possible to experiment with new sensor morphologies easily as no physical hardware reconfiguration is necessary. The design of the robot's sensor configuration becomes another set of parameters that need to be either carefully chosen by the experimenter or optimised by an evolutionary algorithm.

While the evolution of robot controllers is an established field with a large quantity of research, only a few attempts have been made to extend the scope of evolutionary algorithms to include the evolution of the robot's morphology.

Some of the earliest work on sensor evolution was done by Cliff et al. (1992b). The authors were interested in evolving their robots' vision capabilities. They argued for the concurrent evolution of sensor morphology and control networks, going so far as to claim (Cliff et al., 1992a) that "...separating morphology from control is a measure which is difficult to justify from an evolutionary perspective, and potentially misleading." The robot's two photoreceptor sensors were evolved by adapting their positions and view angles. This can be seen in more detail in figure 7. It is also worth noting that Cliff et al. simulated their robots in a virtual environment using basic ray-tracing to simulate the camera sensors as well as a hand-made physics engine to simulate collisions. The robots were made to complete a fairly simple task of avoiding the enclosing wall in a cylindrical environment while covering as much of the environment's area as possible.

Lee et al. (1996) evolved robot sensor morphologies featuring a single simulated robot in an obstacle avoidance task. The directions of the robot's infrared proximity sensors and the robot's controller were evolved simultaneously. Two different EA techniques were used: a GP was used for the controller and a GA was used for the sensor positions. The genome representation was a concatenation of the GP's program and the GA's real-valued

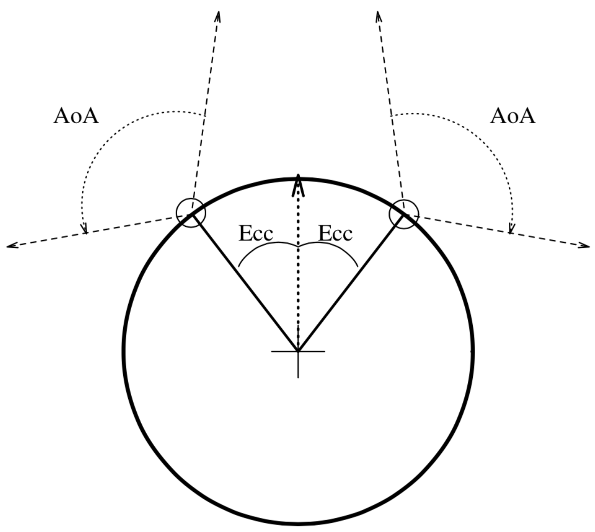


Figure 7: A top-down view of Cliff et al.'s (1992b) robots showing the adjustable parameters for the vision sensors. The parameters are: the angle of acceptance (labeled "AoA") — the view angle or field-of-view, and the eccentricity (labeled "Ecc") — the angle of the camera away from the robot forward direction.

vector. Separate genetic operators were applied to each part of the genome. The authors noted that in many cases a hand-designed robot morphology would perform sufficiently, but for more complicated tasks the additional evolution of the robot morphology may provide an alternative solution.

Later work by Buason and Ziemke (2003) evolved the view angle (field of view) and range of a camera sensor on Khepera robots in a predator-prey competitive co-evolution (CCE) task. Further work by Buason et al. (2005) evolved additional sensor parameters such as camera direction and experimented with not only evolving the agents in the simulation but also their environment. All of Buason's experiments featured a predator-prey task with two agents involved in the simulation. Additionally, the controller topology remained fixed during evolution. Sensor configuration and controller link weights were evolved simultaneously. An evolved predator-prey pair can be seen in figure 8.

It should be noted that adjusting a sensor's *position* is very different from adjusting a sensor's *characteristics*. Buason and Ziemke's technique is less likely to be feasible than Lee et al.'s outside of a simulated environment as they evolve a camera sensor's field-of-view and range. These sensor characteristics would likely be limited, if not *defined*, by the camera lens system's angle of view, aperture, focal-length, zoom level and other properties.

2.5 Collective Behaviour Tasks

2.5.1 Definition

There are a large number of problem domains to which multi-agent systems can be applied. Some kind of task is required to test agent performance. Generally a task is chosen that is simple enough that the search space is not too large — but not too simple

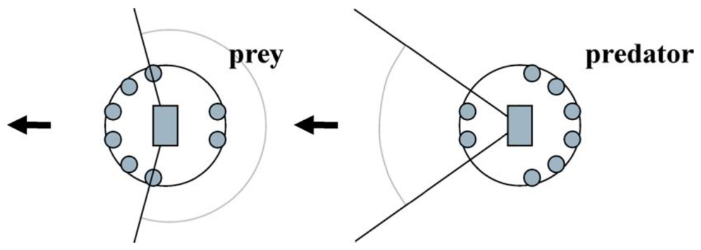


Figure 8: Illustration of Buason et al.'s (2005) evolved predator-prey robots. The predator (right) evolved its camera to face backwards and to move in the same direction, allowing it to keep track of the prey while chasing it. The prey evolved to have its camera face backwards while it moves forward.

that it may be trivially accomplished by a heuristic solution. The task is designed such that a collective behaviour is needed from a team (or teams) of agents in order to perform well at the task. Some common tasks include predator-prey pursuit (Buason and Ziemke, 2003), box pushing (Zhang and Cho, 1999), cooperative navigation (Balch, 1997), and soccer playing (Luke et al., 1998).

2.5.2 Gathering and Collective Construction

The foraging task generally involves a large map that agents (robots) roam to forage for resources (generally pucks or cans). The task is to find the items and to carry them to specially designated areas. This task may be configured in many different ways; the number of sources of resources, the number of destinations, whether the environment is enclosed or not, and more. Østergaard et al. (2001) provide a review of literature that focuses on the foraging task. The authors point out that the majority of research has focused on more complex configurations of the foraging task in order to make the control and analysis problems more interesting.

2.5.3 Example implementations

Waibel et al. (2009) implemented a robotic foraging simulation. Their robots were equipped with a pair of light sensors that were able to detect the colour of walls. The wall adjacent to the single "sink" location was coloured differently to the other walls, making it possible for the robots to easily identify the area they should head to after picking up resources. The authors also created a simulation in which multiple agents were required to cooperate in order to push larger resources back to the sink location. The simulation environment can be seen in figure 9.

Goldberg and Matarić (2000) used a different approach. The sink location was in a single corner of the environment. The robots used by the authors were very sophisticated in comparison to most other related research. Robots kept track of their heading direction and would effectively "remember" the direction home.

Balch (1999) refers to *multi-foraging* where robots are required to collect different types of objects and deliver them to different locations according to type. The authors investigate the relationship between performance and the behavioural homogeneity of

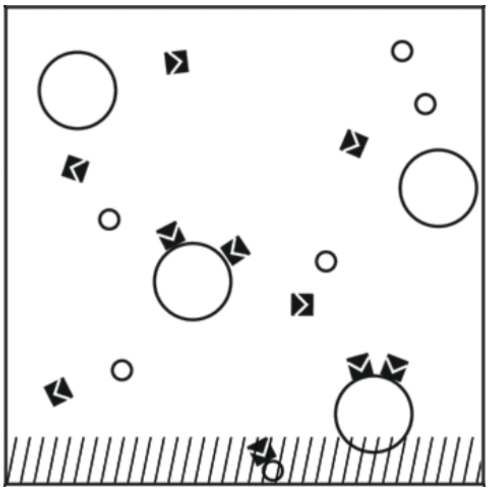


Figure 9: The experimental set up for one of Waibel et al.'s (2009) tasks. The square robots must cooperate in order to push the large circular resources towards the target area at the bottom of the scene.

agents by testing three different strategies. The three strategies were: *homogeneous*, where each agent is able to deliver all types of objects; *specialize-by-color*, where each robot specialises in collecting one type of object; and *territorial*, where most of the robots collect objects and drop them off near a delivery area while a single agent sorts the objects and completes delivery. They found that the homogeneous strategy performed best.

3 Neuro-Evolution Methods

Neuro-Evolution of Augmenting Topologies (NEAT) (Stanley and Miikkulainen, 2002) was chosen as the algorithm to evolve robot agents. A new evolutionary method was developed to extend NEAT to evolve both the robot's controller and the robot's morphology. The proposed method is called *Neuro-Evolution of Augmenting Topologies and Morphologies* (NEAT-M).

3.1 Neuro-Evolution of Augmenting Topologies (NEAT)

We will explain the NEAT method in detail here as the method for morphology evolution makes use of much of the same logic. The morphology evolution scheme builds directly on top of the framework provided by NEAT.

3.1.1 Initialisation and Selection

The initial population is created with all genomes having an identical, minimal morphology. This set of nodes includes the input and output nodes and the bias node. No hidden nodes exist initially. Links are randomly created with a certain initial connection density.

Each generation a new population is created and tested. This population is created in a number of ways. A certain portion of the top genomes of each species each generation is preserved through to the next generation (this is also known as *elitism*). The other genomes are created through mutation. The genomes to be mutated for the next generation are chosen using a truncation selection process. This selection process randomly chooses a genome from a portion of the top genomes of each species until the population is full. The population size remains constant meaning that this is a *Steady-State Genetic Algorithm* (SSGA).

3.1.2 Mutation

The mutation operators can be split into two types: link weight mutators and network topology mutators.

Link weight mutators adjust the values of the input weights for nodes. These work either by applying a Gaussian perturbation to the weight value or by resetting the weight value to a new random value. The link weights are always clamped to their valid range. Links are chosen from the network either by choosing a fixed number of links randomly or by randomly selecting a small portion of the links in the network.

There are three kinds of network topology mutators: the add link mutator, the remove link mutator, and the add node mutator. The add and remove link operators are relatively simple, they simply add or remove a random link in the network. The add link operator makes sure not to add a link that duplicates another link or that links into an input node or the bias node. The remove link operator checks that the network it is operating on is not too small before removing a link. It also makes sure to remove hidden nodes that have no links leading into or out of them after a link is deleted.

The add node operator is a somewhat more complicated. This operator finds a link in the network and splits the link, inserting a node in the middle. The link is split by first disabling the link and then creating two new links that connect the new node between the input and output of the old link. The link into the new node is given the weight of the old link and the link out of the new node is given the maximum possible weight value. Stanley and Miikkulainen (2002) explain this approach as being designed to prevent large "nonlinearities" in the network when a new node is added.

3.2 Neuro-Evolution of Augmenting Topologies and Morphologies (NEAT-M)

The morphology adaptation method was designed in an ad-hoc fashion and builds on top of the features of NEAT. The simplest approach was taken to adding sensor information to the NEAT genome — encoding extra information in the input nodes. Three additional information fields were added to the input node genes: sensor type, bearing, and orientation. NEAT-M adds two new genetic operators: one that evolves sensor position (bearing and orientation), and one that adds new sensors to the morphology.

3.2.1 Sensors

Sensor types: The two types of sensors used were the Khepera IR proximity and ultrasonic sensors. It would be possible to add new types of sensors as long as the sensor type could be configured using the existing parameters. The sensor types were restricted to the two sensor types available on a standard Khepera III so that it would be possible to compare the evolved morphology with a real-life robot morphology.

Sensor parameters: The abstraction used for sensors in the simulations is best described by figure 11. The bearing and orientation sensor parameters were chosen to be evolved as these are the parameters that could be most easily changed in an actual robot. Adjustment of the sensor bearing and orientation could be achieved by simply attaching sensors to different positions on the robot's body. This approach is very similar to that taken by Lee et al. (1996) in that it evolves sensor position. It is less similar to Buason and Ziemke's (2003) approach which evolves the sensor characteristics.

3.2.2 Mutation

Sensor position mutation: The sensor position mutation operator was implemented very similarly to the NEAT link weight mutation operators. Two operators were introduced: a position perturbation operator and a position reset operator. The perturbation operator shifted the sensor positions according to random values chosen from a Gaussian distribution. The bearing value is wrapped when perturbed beyond its valid range while the orientation value is clamped to its valid range. The reset operator simply moves a sensor to a new random position. These operators are applied by choosing either one or two sensors randomly.

Sensor addition mutation: Since adding a sensor is effectively adding a node to a network, NEAT's infrastructure for adding nodes could be reused. NEAT's speciation system and crossover system meant that new nodes could be added to the network without reimplementing those systems. The sensor addition mutation operator works by adding a single new input node to the network with a randomly chosen type and randomly initialised position. An important aim when adding a new sensor to the network is to affect a change to the behaviour of the network but not to completely change the behaviour and destroy the network's usefulness. This is similar to Stanley and Miikkulainen's (2002) aim to limit "nonlinearity" when adding a new hidden node to the network.

The links connecting the new sensor to the rest of the network are created in a different manner to those created for a new hidden node. Links from the new input node are instead created using a process similar to the way links are initialised in a new genome. The links are created randomly but with a certain connection density. The connection density was chosen to be a relatively low value so as not to add too many links to the network. Adding many links would result in a network that is very different to other networks in the population and so would never be found to be of the same species as other networks.

Link weights are chosen so as to have an effect on the network but not to cause too drastic a change. If these links were initialised with a connection weight of 0, the addition of the new sensor would have no effect on the network (at least until the sensor weight is mutated). Initialising the new links with a large weight could result in a large change to the network's behaviour. The sensor addition operator was designed to be configured with a link weight mutation operator. The links are initialised with weight values of 0 and then the weight mutation operator is applied to the links to ensure that the sensor has some effect on the network. It is useful to be able to configure different weight mutation operators when adding different types of sensors. When adding a sensor with a short range and narrow field-of-view, for the majority of the time the sensor will not provide any signal. Setting a small link weight for this sensor would mean that it would only have a very small effect on the behaviour of the network. Setting a large link weight gives the sensor more potential to change the network's behaviour. The opposite is true for sensors which have a long range and wide field-of-view. For ultrasonic sensors, it is necessary to be more conservative when adding new links to the network so as not to completely distort the network's behaviour.

3.2.3 Speciation

The speciation system needed to be adapted to account for sensor positions. This was done by simply adding extra terms to the speciation equation (equation 3). The updated speciation equation is given by,

$$\delta = \delta_n + c_4 \cdot \overline{B} + c_5 \cdot \overline{O} \tag{4}$$

where δ_n is the result of the NEAT speciation equation (equation 3), c_4 and c_5 are constant factors, $\overline{B}$ is the vector of differences in sensor bearings, and $\overline{O}$ is the vector of differences in sensor orientations.

The sensor bearing, $\overline{B}$, and orientation, $\overline{O}$, difference vectors are calculated by first normalising the sensor bearing and orientation to values between -1.0 and 1.0.

4 Experiments and Results

The development work for the project was split into two key parts:

1. The simulator platform that emulated the environment in which tests would be run. This part was developed by all the group members.
2. The NE method that would be used to develop the simulated robotic agents and the experiment design that would be used to evaluate the effectiveness of the method. This part was developed individually.

Overall, the project required a significant amount of software development. We reused existing software where available but ultimately were required to write a large amount of code to link together and adapt various libraries in order to create a realistic simulation.

4.1 Simulator Platform

We decided to use the MASON (Luke et al., 2005; Sean Luke and Panait, 2014) ABM as the basis for our simulator platform. We chose MASON because it had several advantages over other systems:

- It is written in the Java programming language, a language which all team members had previous experience with.
- It is one of the few ABMs that is actively maintained.
- MASON is specifically designed for Multi-Agent Simulations.
- It has a comprehensive user manual (Luke, 2014).

MASON provided the underlying structure to our simulator by specifying a system for placing and drawing objects in the scene. Additionally, MASON provided the following functionality for our simulator platform:

- A graphical user interface (GUI) to visualise the simulation.
- Detailed controls over the simulation such as the ability to play or pause the simulation, speed or slow the simulation and to adjust the graphical scale.
- A command line interface to the simulation when running simulations without the GUI.

MASON provided a general framework for our simulator but two key additional features were needed. Firstly, MASON does not provide a physics engine which we needed to accurately simulate the interactions between our robot agents. Secondly, MASON does not provide any elements of the evolutionary algorithms that would be needed to evolve our agents. There is no scoring system and no specific inputs or outputs for the robot agents are provided.

In order to create a fully-featured simulator, we integrated a physics engine with MASON. After assessing a number of different libraries we decided to use JBox2D (Murphy, 2014), a Java implementation of the Box2D library (Catto, 2014) which is a popular and

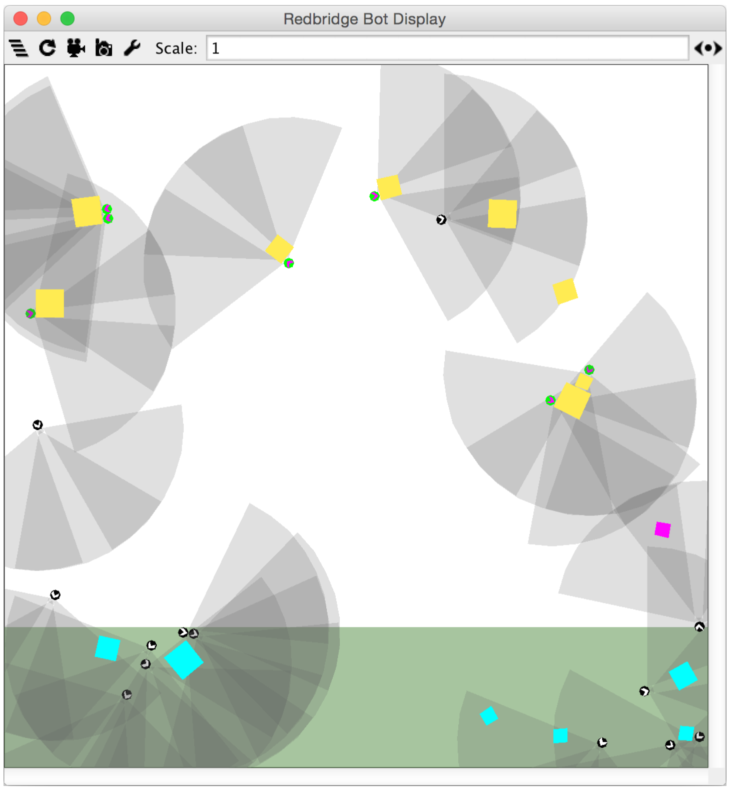


Figure 10: The GUI for the simulator platform. The agents are the green and black circles. The square objects are resources and are coloured yellow if they are uncollected, cyan if collected, and magenta if they have been collected but were pushed out of the target area. The transparent grey shapes are the robots' sensor fields.

widely-used C++ 2D physics engine. Jbox2D provided most of the tools needed to simulate the physical environment for the robots to be tested in. The GUI for the completed simulator platform can be seen in figure 10.

4.1.1 Environment Configuration

The environment was a square field enclosed on all sides by walls. The rectangular target area where agents would drag foraged resources was set up along the length of one of the four walls. The basic configuration of the environment is specified in table 1.

Environment dimensions	20 x 20m
Target area dimensions	20 x 4m

Table 1: The dimensions of the environment and target area.

Resources were marked as collected if they had passed entirely into the target area and were marked as uncollected if any part of the resource moved out of the target area.

4.1.2 Resource Configuration

A mix of different sizes of resources was chosen in order to test the cooperation of agents. Three different sizes of square resources were used. Larger and heavier resources are difficult or impossible for single agents to push, thus requiring multiple agents to move to the target area. The dimensions and masses of each of these sizes are listed in table 2.

Size	Dimensions	Mass	Number of robots required to move
Small	40 x 40cm	1kg	1
Medium	60 x 60cm	3kg	2
Large	80 x 80cm	6kg	3

Table 2: The dimensions and masses of the different resource configurations.

4.1.3 Robot Configuration

The virtual robots were designed to roughly approximate the Khepera III robots in terms of physical size, shape and weight. The physics engine made it possible for us to also emulate the forces of two wheels with roughly the same power, size and positions as the Khepera III. The robots are circular which is similar to the Khepera III's shape. The robots' basic specifications are listed in table 3.

Body diameter	30cm
Mass	0.7kg
Wheel diameter	6cm
Engine torque	0.075nm
Maximum speed	0.5m/s

Table 3: The specifications of the robot agents. These values are modelled after the Khepera III robot.

The robots' movement is entirely controlled via the drive strengths to the two wheels. The wheels are controlled with two continuous scalar values between -1.0 and 1.0. A full range of motion can be achieved in this way. For example, a robot can be driven forward with full force by powering both wheels at 1.0 or backwards by powering at -1.0. To turn left, the left wheel is powered at a lower value than the right. Turning right is achieved in a similar manner.

4.1.4 Robot Sensor Configuration

Agent sensors were modelled as conical fields emanating from the outer circumference of the agent. All sensors (with one exception) had a number of common configuration parameters as shown in figure 11. The possible parameter configurations are described in table 4.

These parameters could be used to create a variety of sensors but three sensor types in particular were developed. These sensors were modelled after the sensors present in the Khepera III robot in its standard configuration.

The first was an infra-red (IR) proximity sensor. The Khepera III uses Vishay TCRT5000 proximity sensors (Lambercy and Tharin, 2013) which are quoted as having a maximum range of 30cm (Corporation, 2014) by the makers of the robot. This number is likely

Parameter	Valid range	Unit	Description
Bearing	$[-\pi, \pi]$	Radians	The position of the sensor on the robot's circumference rotated from the robot's heading.
Orientation	$[-\pi/2, \pi/2]$	Radians	The offset of the sensor's heading direction relative to directly radially outward from the agent.
Range	$(0, 10^{12})$	Meters	The maximum range (distance) of the sensor's field.
Field of view	$(0, \pi]$	Radians	The angle between the edges of the sensor's field.

Table 4: The different parameters that describe a sensor configuration. All sensors in the simulation $with the exception of the bottom proximity sensor$ could be abstracted to a set of these parameters.

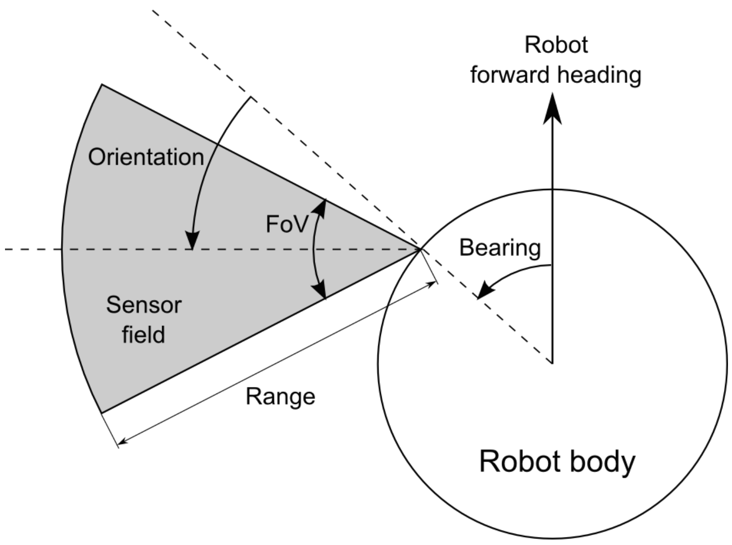


Figure 11: A top-down view of the robots in the simulation. The sensors were configured as conical fields extending from the edge of the robot.

an upper bound on the sensor range. The sensor manufacturer's datasheet (Vis, 2009) quotes a nominal operating range of 2.5mm (as can be seen in figure 12). IR proximity sensors essentially work by emitting a pulse of infra-red light and measuring the amount of that light that is reflected back. This means that these sensors can be sensitive to environmental factors such as the amount of ambient infra-red light and the reflectivity of the objects which they should be detecting.

The second type of sensor is the ultrasonic sensor. This sensor pairs an ultrasonic transmitter and receiver. It works by emitting a short ultrasonic sound burst and then recording and analysing the sound waves that are reflected off objects in the scene. Again, this type of sensor is sensitive to noise in the environment and to the characteristics of the objects to be detected. The ultrasonic sensor in the Khepera III is made up of the Midas 400ST100 transmitter and 400SR100 receiver (Lambercy and Tharin, 2013). Additional signal processing is performed on the output of these sensors in order to resolve useful information. In the Khepera III, each processed sensor measurement provides a set of readings for each recorded echo. Each echo should correspond to an object in the

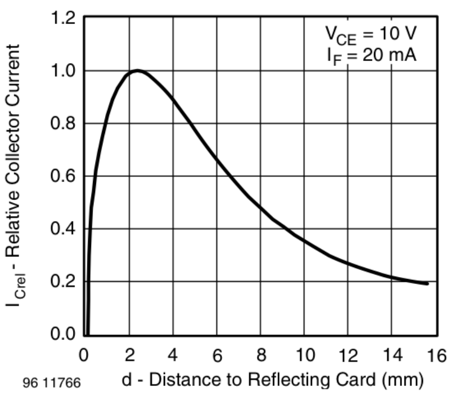


Figure 12: Relative collector current versus distance for the Vishay TCRT5000 sensor. (Vis, 2009)

sensor's field. Each reading contains an estimated distance in centimetres to the object, an amplitude, and a timestamp. The datasheet for the Midas parts (Mid) do not provide much information about the expected range or characteristics of a sensor that implements the transceiver/receiver pair. The Khepera user manual (Lambercy and Tharin, 2013) quotes the ultrasonic sensors as having a range of up to 4 meters. It also notes that the ultrasonic sensors have poor sensitivity when tracking objects very close to the robot. Objects within 20cm of the sensor can be detected but their distance cannot. Objects in the 20-40cm range may not be accurately tracked.

The third and final sensor type that was implemented was another implementation of the proximity sensor. The Khepera III features two proximity sensors that face the ground. The user manual quotes these sensors as being useful for line-following applications. We instead used these sensors to detect when the robot was positioned in the target area. If the ground in the target area were painted a different colour to the ground in the rest of the environment then these bottom-facing sensors could potentially be used to accurately detect the presence of the target area.

The overall morphology of the Khepera robot was closely imitated including the shape, size, mass, top speed and positioning of the wheels.

4.1.5 Guidance Heuristic

The robots feature an extra sensor that is not used as input to the controller but is instead used by a heuristic. The sensor is a small sensor with a rectangular field (10 x 20cm) that is positioned just in front of the robot. When this sensor detects a resource, the heuristic will attempt to "pick up" the resource. If the robot is properly aligned with the resource, this will result in a physical connection or joint being made between the agent and the resource. The heuristic will then take complete control of the robot and attempt to guide the robot together with the resource back to the target area. Once the resource is within the target area and marked as collected, the joint between the robot and the resource is removed and control of the robot is returned to the evolved controller.

This heuristic was chosen as it was decided that having the robots learn the other behaviours needed to complete the task — such as collision avoidance and resource tracking

— was enough to test the effectiveness of the evolutionary methods and measure the level of cooperation between agents. New, longer range sensors would need to be developed in order for the robots to be able to detect the target area from far-away parts of the environment.

4.1.6 Other Configurations

The simulation ran at an update rate of 10 steps per simulated second. In other words, agents sampled the environment at a maximum rate of 10 samples per second.

The physics engine operated at a faster rate in order to achieve reasonable accuracy. It was configured to update object velocities 60 times per second and object positions 30 times per second.

In order to limit the robots' ability to easily move heavy resources about the scene, simplified friction was implemented between resources and the ground as well as between robots and the ground.

4.2 Experiment Setup

Three different experiments were created to test differing levels of cooperation by changing the configuration of the environment. The evolving morphology method, NEAT-M, was compared to a fixed morphology based on the Khepera III robot. The fixed morphology agents' controllers were evolved using the standard NEAT method. Further testing of the morphologies produced by NEAT-M was also carried out.

4.2.1 Environment Configuration

Three different experiments were created by configuring the environment in different ways. Since the different sizes of resources required different levels of cooperation to push, mixing different quantities of each size produces tasks that test cooperation to a greater or lesser extent. The three configurations are listed in table 5.

	Small resources	Medium resources	Large resources
Experiment 1	15	0	0
Experiment 2	5	5	5
Experiment 3	0	5	10

Table 5: Each experiment featured a different mix of resource types. Experiments ranged from not featuring any cooperation to requiring cooperation.

Experiment 1 does not test cooperation as none of the resources require cooperation to move. Experiment 2 contains a mix of small and larger resources meaning that agents can either act selfishly or cooperate. The greatest fitness can be achieved by agents that do both, but cooperation is not necessarily required in order to achieve some fitness. Experiment 3 contains only large resources and so cooperation is always required to achieve high performance.

In all three experiments 20 agents were placed in the environment.

4.2.2 NEAT Implementation

A number of machine learning libraries contain implementations of NEAT. The Encog Machine Learning library (Heaton, 2014) was chosen as it is one of the most actively maintained Java-based libraries. It was also determined to be flexible enough to serve as a basis for new machine learning algorithms. The authors of Encog used (Stanley and Miikkulainen, 2002; Whiteson et al., 2005; Gauci and Stanley, 2007) as references when implementing NEAT.

NEAT, like most other machine learning methods, has a number of parameters that can be tuned to extract the best performance for the task at hand. Encog provides a pre-configured implementation of NEAT but this configuration is not tuned for multi-agent simulations. Throughout the configuration of the NEAT system, the original C++ NEAT code by Stanley (2001), which contains many detailed implementation notes, was referenced for parameters. Additionally, the NEAT Users' Group (Stanley, 2013) provides a number of guidelines for configuring NEAT.

The basic parameters for the NEAT configuration are listed in table 6.

Population size	100
Truncation selection	Top 30% of population
Elitism	30%
Speciation constants	$c_1 = 1.0$, $c_2 = 1.0$, $c_3 = 0.4$, $\delta_t = 3.0$
Initial connection density	50%
Link weight range	$[-1.0, 1.0]$

Table 6: The parameters used to configure the NEAT algorithm.

The truncation selection, elitism, and speciation constants were the default values in Encog (Heaton, 2014). The same values were found in the NEAT C++ source code (Stanley, 2001).

The default value for the initial connection density in Encog is 10% but this was found to be too low. The networks being used always had two outputs (to control each of the robot's two wheels) but often had many more inputs. With a low connection density, very few networks in the initial population had any potential for moving the robot agents in a useful manner. The initial connection density was set to 50%.

The activation function is the logistic function as seen in equation 2. The value for the slope, μ, was 4.924273 as recommended by Stanley (2001) as this causes the activation function to be as close to a linear ascent as possible when the weighted sum is between -0.5 and 0.5. This activation function can be seen in figure 13.

In the Encog NEAT implementation, the recombination operator (crossover operator) is applied in the same manner as a mutation operator. The crossover operation chooses two parent genomes and creates one new genome that does not undergo any further mutations. The other operators choose one parent genome, create a copy of the parent, and then mutate the copy. The mutation operators for NEAT were applied as shown in

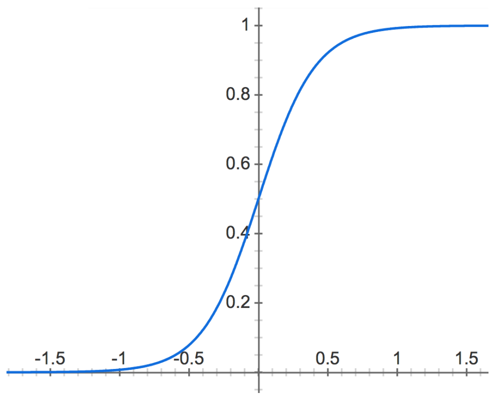


Figure 13: The curve of the activation function with slope $\mu = 4.924273$.

table 7. These probabilities are similar to the default values in Encog (Heaton, 2014) but the probabilities of the add node, add link, and remove link operators which were increased somewhat. This was because the population size was relatively small, causing these operators to only very rarely be used. For the link weight mutation operator, different operators are applied to different selections of links as shown in table 8.

Mutation operator	Probability
Crossover	50%
Link weight	49.3%
Add node	0.1%
Add link	0.5%
Remove link	0.1%

Table 7: The probability of each NEAT mutation operator being chosen.

4.2.3 NEAT-M Implementation

NEAT-M was implemented by extending the NEAT software based on Encog. New mutation operators were implemented using the framework for operators that exists in Encog. NEAT-M was also configured in a very similar manner to NEAT. The parameters remained the same but the new sensor mutation operators needed to be configured as well as the new speciation equation (equation 4). These extra parameters are listed in table 9.

The new sensor mutation operators meant that the probabilities of the other mutation operators had to be changed. The probability of each operator being chosen is shown in table 10. The probabilities for the link weight mutations remained unchanged and are as described in table 8.

Adding proximity sensors was considered to be “cheaper” than adding ultrasonic sensors and had a higher chance (60%) of being added. Real-life proximity sensors are cheaper and simpler than ultrasonic sensors. A limit of 10 sensors for a single robot was enforced. The probability of a given sensor position mutation occurring is shown in table 11.

Mutation operator	Selection method	Probability
Perturb weight ($\sigma = 0.004$)	Select 1 link	11.25%
	Select 2 links	11.25%
	Select 3 links	11.25%
	Select 2% of links	11.25%
Perturb weight ($\sigma = 0.2$)	Select 1 link	11.25%
	Select 2 links	11.25%
	Select 3 links	11.25%
	Select 2% of links	11.25%
Reset weight	Select 1 link	3%
	Select 2 links	3%
	Select 3 links	3%
	Select 2% of links	1%

Table 8: The probability of each NEAT link weight mutation operator being chosen if a link weight mutation is being performed.

New sensor connection density	0.1
Speciation constants	$c_4 = 0.4$, $c_5 = 0.1$

Table 9: The NEAT-M configuration parameters chosen.

4.2.4 Khepera Morphology

A morphology modelled after the Khepera III robot was used as a control case. This morphology was tested with a controller evolved using NEAT and the results were compared to the NEAT-M scheme. The Khepera III in its standard configuration features a total of 11 IR proximity sensors and 5 ultrasonic sensors (Lambercy and Tharin, 2013). Using this number of sensors would drastically slow down our simulations as doing so greatly increases the number of objects in the scene. Thus, some of the sensors were disabled in the interest of being able to run more tests.

The sensors were placed around the circumference of the robot at bearings measured from diagrams in the user manual. The IR proximity sensors' positions were measured using figure 14a and estimated to be at bearings $\pm 10°$, $\pm 40°$, $\pm 75°$, $\pm 140°$, and $180°$. Of these sensors, the sensors at $\pm 40°$ and $180°$ were enabled. The bottom-facing proximity sensors were modelled as a single target area sensor as described in section.

The ultrasonic sensors' positions were measured using figure 14b and were placed at bearings of $0°$, $\pm 40°$, and $\pm 90°$. Of these sensors, the sensors at $0°$ and $\pm 90°$ were enabled.

This configuration made for a total of 7 sensors. The configuration was chosen to provide an even spread of sensor fields in every direction. The final morphology can be seen in figure 14c.

4.2.5 Fitness Function

Resources are assigned a value according to their size. A medium-sized resource is worth twice as much as a small-sized resource and a large-sized resource three times as much.

Mutation operator	Probability
Crossover	40%
Link weight	46.5%
Add node	1%
Add link	1%
Remove link	0.5%
Sensor position	10%
Add sensor	1%

Table 10: The probability of each NEAT-M mutation operator being chosen.

Mutation operator	Selection method	Probability
Perturb position ($\sigma_B = 0.005$, $\sigma_O = 0.01$)	Select 1 sensor	22.5%
	Select 2 sensors	22.5%
Perturb position ($\sigma_B = 0.1$, $\sigma_O = 0.05$)	Select 1 sensor	22.5%
	Select 2 sensors	22.5%
Reset position	Select 1 sensor	5%
	Select 2 sensors	5%

Table 11: The probability of each NEAT-M sensor mutation operator being chosen if a sensor mutation is being performed.

Each simulation runs for either a fixed number of steps or until all the resources have been collected. A time bonus is awarded for agents that collect all the resources before the end of the simulation.

A fitness value is then calculated at the end of a simulation run using the equation,

$$f = 100 \times \frac{v_c}{v_r} + 20 \times (1.0 - \frac{s_e}{s_t}) \tag{5}$$

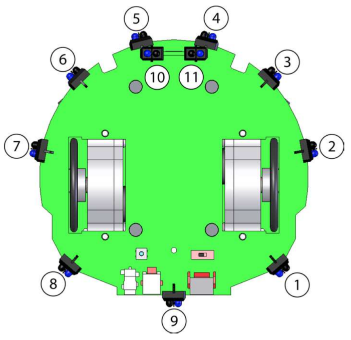
where v_c is the total value of the collected resources, v_t is the total value of all the resources, s_e is the number of elapsed simulation steps, and s_t is the total number of simulation steps allowed.

Fitness was averaged over multiple simulation runs so as to produce an accurate value for agent performance.

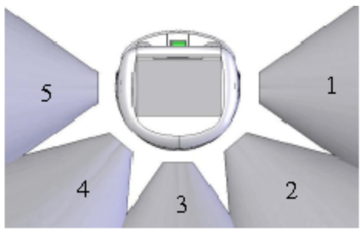
4.2.6 Simulation Configuration

The simulation was set up with the parameters in the table below. Each simulation ran for at most 10,000 steps which equates to 16 minutes and 40 seconds of simulated time. Each agent was tested 5 times before assigning a fitness which was calculated by averaging the scores of the 5 runs. Agents were evolved for a maximum of 250 generations or until an agent's fitness reached a score of 110 or higher. These parameters are summarised in table 12.

Agents and resources are initially placed in the environment in random positions outside of the target area with random orientations.



(a) Bottom view of the robot with numbered IR proximity sensors. The cylindrical silver motors and black tyres can be seen in the centre of the robot's body.



(b) Top view of the robot with numbered fields of the ultrasonic sensors.

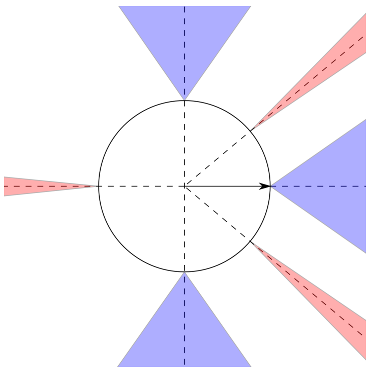

(c) The morphology used to represent the Khepera robot. IR proximity sensors are coloured red while ultrasonic sensors are blue.

Figure 14: The design of the Khepera III morphology. 14a and 14b are taken from (Lambercy and Tharin, 2013).

Simulation steps	10,000
Epochs/generations	250
Number of simulation runs	5
Convergence score	110

Table 12: The simulation configuration for all three experiments.

4.3 Results

A total of 10 tests were run for every test case and an average score was calculated. This number of tests may seem low but it involves a large number of computationally expensive simulations. Each agent was tested 5 times. For a population of 100, as was used in this testing, this results in up to 500 simulations per generation. Thus, over all 250 generations, a maximum of 125,000 simulation runs are performed for each test. As described in section 4.1.6, each simulation involves 10,000 time steps of a tenth of a second each. Each test run then simulates a maximum of nearly 35,000 hours of the scene.

Unlike most ML benchmarks, we are not concerned with the number of computational steps needed to produce a certain level of performance, since the task can always be made harder by reconfiguring the scene. We are instead concerned with absolute task performance with a particular configuration of the task.

4.3.1 Testing

Agents were evolved using the proposed ML method, NEAT-M. The performance of these agents was compared to a control case: robots with a morphology modelled after a real-life robot, the Khepera III, and with controllers evolved using standard NEAT.

The statistics of the fitnesses of the best genomes in each run can be seen in table

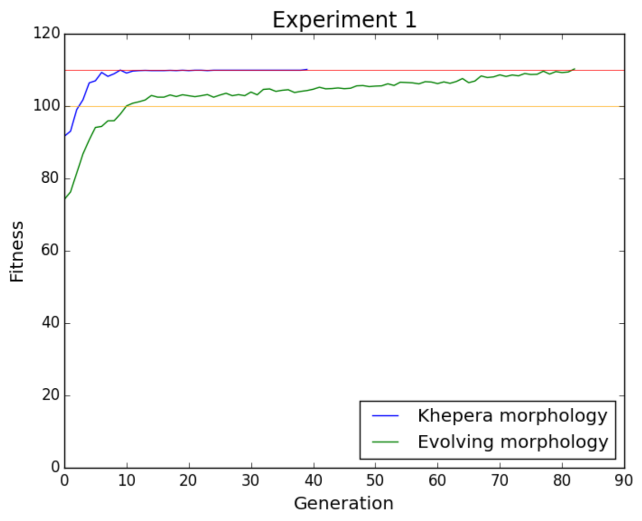


Figure 15: Graph showing task fitness versus EA generation in experiment 1

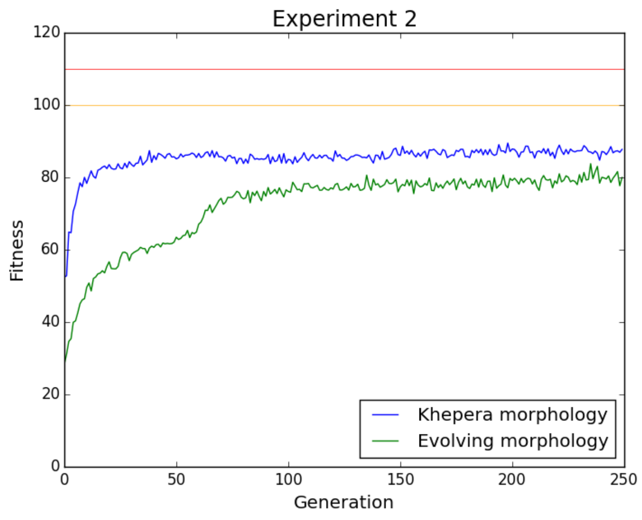


Figure 16: Graph showing task fitness versus EA generation in experiment 2 for the evolving morphology.

13. The normality test used was the Shapiro-Wilk test (Shapiro and Wilk, 1965) as it is suited to smaller datasets (Razali and Wah, 2011). All the data sets had p-values greater than the significance value, $\alpha = 0.05$, and so the null hypothesis that the data is normal was *not* rejected.

The data sets were then tested using a Student's t-test with the null hypothesis that the samples have identical average values. The results of these tests can be seen in table 14.

4.3.2 Further testing: The evolved morphologies

Two of the evolved robots were chosen for further examination. The highest performing evolved networks and morphologies for experiment 2 and 3 are shown in figure 18. In the

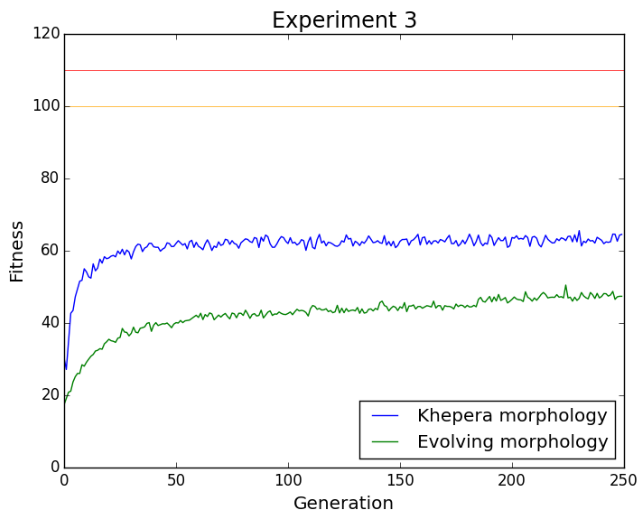


Figure 17: Graph showing task fitness versus EA generation in experiment 3 for the evolving morphology.

figure, the graphs of the networks have the link weight values listed next to the edges. The edges with dashed lines show disabled links. In the diagram of the robot morphology, the red-coloured areas show the fields of the IR proximity sensors while the ultrasonic sensors' fields are blue-coloured.

An evolved morphology was not taken from the tests in experiment 1 as these genomes underwent relatively little evolution before reaching a solution collected all the resources. Thus, it was determined that any solutions would be of relatively little complexity.

These evolved morphologies were tested using standard NEAT to evolve their controllers while keeping their morphologies fixed. The results of these experiments can be seen in figure 19. The morphology in figure 18a is labeled *morphology 1* while the morphology seen in figure 18b is *morphology 2*.

The statistics of the fitnesses of the best genomes in each run can be seen in table 15. Again, the p-values for the normality test were calculated and it was *not* possible to reject the null hypothesis that the data sets conformed to a normal distribution. A three-way comparison of the morphologies using the Student's t-test was conducted and the results are listed in table 16.

Case	Score	NEAT-M	Khepera & NEAT
Experiment 1	Maximum	112.6328	113.5896
	Minimum	110.01	110.31
	Mean	110.93	111.57
	Standard deviation	0.94	1.03
	Normality test p-value	0.06	0.59
Experiment 2	Maximum	100.41	107.54
	Minimum	64.00	78.67
	Mean	89.13	94.75
	Standard deviation	11.03	11.27
	Normality test p-value	0.10	0.87
Experiment 3	Maximum	69.00	93.22
	Minimum	41.00	40.00
	Mean	53.85	71.93
	Standard deviation	9.02	15.64
	Normality test p-value	0.94	0.75

Table 13: Summary of scores of agents evolved with NEAT-M compared to agents evolved with NEAT and the Khepera morphology.

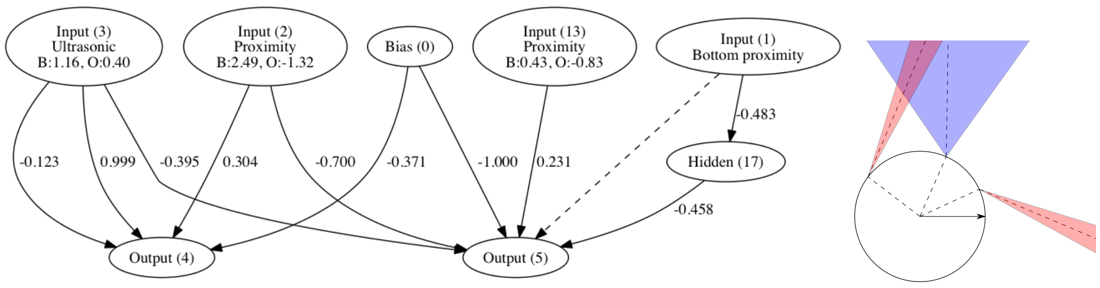
	t-test p-value
Experiment 1	0.17
Experiment 2	0.27
Experiment 3	0.005

Table 14: The p-values from the Student's t-test with the null hypothesis that the samples have identical expected values.

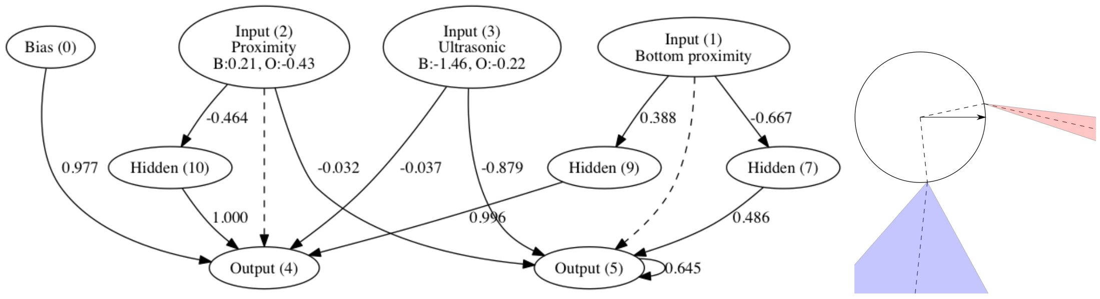
5 Discussion

In experiment 1, both NEAT and NEAT-M quickly evolve to solutions that solve the task and reach the convergence criteria. Both methods perform similarly but on average NEAT with the Khepera morphology converges faster. This task shows that without cooperation this configuration of the foraging task is not a hard problem. Both methods had runs in which they converged after only 3 generations.

Experiment 2 proves much more challenging for both methods and neither NEAT nor NEAT-M manage to converge to a solution as can be seen in figure 15. Both methods did feature at least one run in which all the resources were collected within the time limit, though. When comparing the best scores from each run (table 13), the Khepera and NEAT solution produces higher performance. Still, the results are comparable as the t-test values (table 14) show that we cannot reject the null hypothesis that each result set could be produced from normal distributions with the same mean value. While both NEAT-M and NEAT with the Khepera morphology produce comparable genomes after 250 generations, the NEAT and Khepera solution reaches high task performance in fewer generations. Overall, task performance is similar, if slightly decreased, with NEAT-M. Thus, it is difficult to argue that NEAT-M is either beneficial or detrimental to cooperation in this particular task.



(a) This network and morphology scored 100.41 at generation 199 for experiment 2.



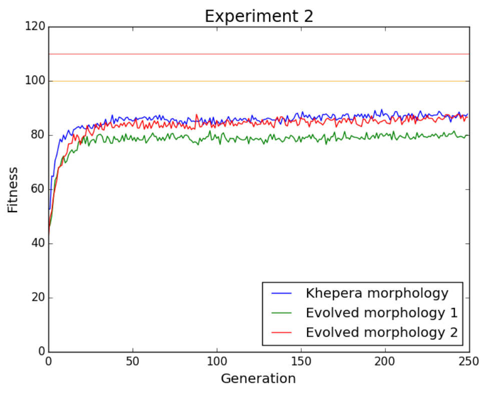
(b) This network and morphology scored 69 at generation 225 for experiment 3.

Figure 18: The best solutions evolved after 250 generations in experiment 2 and 3 using NEAT-M. The network graphs are shown on the left while the morphologies are shown on the right.

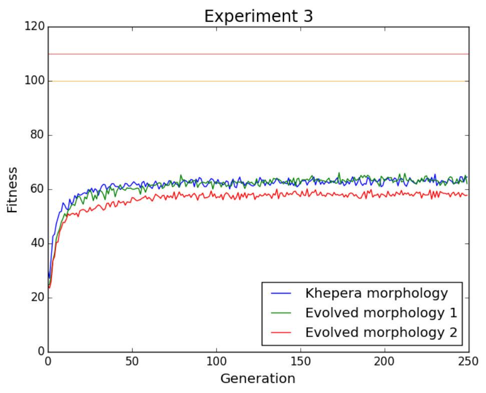
The third experiment configuration is even more challenging and neither method managed to ever collect all the resources within the simulation time. The performance gap between NEAT and NEAT-M widens, and it becomes difficult to argue that NEAT-M's performance is comparable to that of the NEAT and Khepera solution. Unlike in the previous two experiments, the *t*-test shows (table 14) that these data sets are clearly independent as the *p*-value (0.005) is below the significance, $\alpha = 0.05$. The NEAT-M method struggles to evolve good solutions for this problem and based on its performance it could be argued that morphology evolution is *detrimental* to the level of cooperation exhibited by the agents.

The results for the NEAT-M method were disappointing in comparison to the fixed morphology method. The question of whether this was due to poor morphologies being produced or due to some other flaw in the method was examined by evolving controllers for some of the evolved morphologies. The morphologies chosen were taken from the highest performing agents in experiment 2 and 3 that were evolved after all 10 runs of 250 epochs. These morphologies can be seen in figure 18 and are quite different from the Khepera morphology seen in figure 14c. The key difference is that they feature fewer sensors than the Khepera morphology. The evolved morphologies were not tested in experiment 1 in the interests of saving time. Besides, the task takes requires a small number of generations for agents to evolve to a solution that it is unlikely that particularly complex networks are produced.

The agents with fixed morphologies perform much closer to the Khepera morphology. For both morphologies in both experiment 2 and experiment 3, the *t*-test shows (figure 16) that the best solutions created with these morphologies are comparable to the best Khepera solutions as the *p*-values are greater than the significance value, $\alpha = 0.05$. In the case of morphology 1, this performance is achieved with only two sensors and the bottom



(a) Experiment 2.



(b) Experiment 3.

Figure 19: Graphs showing task fitness versus EA generation for the fixed, evolved morphologies.

Case	Score	Khepera	Morphology 1	Morphology 2
Experiment 2	Maximum	107.54	102.74	103.80
	Minimum	78.67	71.67	90.34
	Mean	94.75	90.61	96.22
	Standard deviation	11.27	8.73	4.61
	Normality test p-value	0.18	0.16	0.52
Experiment 3	Maximum	93.22	79.38	75.00
	Minimum	40.00	65.00	52.50
	Mean	71.93	72.61	66.05
	Standard deviation	15.64	5.22	7.17
	Normality test p-value	0.75	0.39	0.42

Table 15: Summary of scores of agents with the evolved morphologies compared to agents evolved with the Khepera morphology.

proximity sensor. These results suggest that the morphologies produced by NEAT-M are useful in that they can match the performance of the Khepera morphology, but with fewer sensors. Still, these morphologies do not outperform the Khepera morphology and it cannot be argued that they result in a higher level of cooperation than can be achieved with the Khepera morphology.

	Khepera vs. Morphology 1 t-test p-value	Khepera vs. Morphology 2	Morphology 1 vs. Morphol
Experiment 2	0.37	0.71	0.09
Experiment 3	0.90	0.29	0.03

Table 16: The p-values from the Student's t-test with the null hypothesis that the samples have identical expected values.

6 Conclusion and Future Work

The results show the challenge of developing a system for concurrent evolution of robot controllers and morphologies in a cooperative co-evolution task. The evolving morphology method, NEAT-M, is comparable to evolving just the controller using the Khepera morphology, but generally produces solutions with lower performance in all three experiments.

The reason for this performance discrepancy is difficult to isolate. Adding the evolution of morphology does increase the search space of the algorithm as there are more parameters to optimise. There are also new parameters for the experimenter to select, such as how much weight to give the sensor mutation operators and how much weight to give sensor position differences in the speciation equation. It is possible that NEAT-M could perform well with the correct parameters but it will be challenging to determine what those parameters are.

It can be seen in figure 18 that NEAT-M produced some novel morphologies that an experimenter would have been unlikely to design. It can also be seen that NEAT-M does not fully explore the potential of these morphologies. The performance of the morphologies is improved when the controller is evolved separately using standard NEAT as can be seen in figure 19. The agents featuring fixed, evolved morphologies did not exceed the performance of the Khepera agent in experiment 2 and 3 but achieved similar performance while making use of fewer sensors.

Using fewer sensors has a number of advantages. Firstly, fewer sensors mean simpler, cheaper robots. Another advantage is that there is less noise created by sensors in the scene. Since both the proximity sensors and the ultrasonic sensors emit energy (light and sound, respectively), in MASs there is likely to be a higher level of noise in the scene if agents feature more sensors. Our simulations did not emulate the effects of inter-robot sensor noise. Yet another advantage is that agents with fewer sensors are less computationally expensive to simulate, allowing experimenters to run a greater number of tests.

The sensors used in this experiment were relatively primitive — they did not detect specific types of objects making it difficult to distinguish between robots, resources, and the wall enclosing the environment. On the other hand, simpler sensors are easier to generalise into a model that can be permuted using an EA. There is room for further research into new types of sensors. For example, it may be useful for agents to be able to evolve to detect certain types of objects in the scene. Distinguishing between some objects may be important while distinguishing between others may be less so. Experimenters often design simulations in which agents can distinguish between object based on some special property of the object. Waibel et al. (2009) designed their environment such that a sensor mounted at a greater height on their robotic agents could “look over” resource objects (this robot morphology can be seen in figure 6). These sort of design decisions should be automated so that their validity can be tested.

Despite the low performance of the tested configuration of NEAT-M, a case can still be made for the concurrent evolution of agent controllers and morphologies. Experimenters must still make a large number of decisions when designing a simulation such as the cooperative foraging task — more than can be justified by test results.

NEAT-M builds on the foundation of NEAT but does not propose a completely new methodology to tackle the problem of morphology evolution. NEAT-M does not propose a radically different genome representation, mutation, crossover, or speciation scheme. The system NEAT-M uses to encode sensor information in genomes is at a different level of abstraction to the encoding of the controller. A set of link edges and weights describe little about the agent before they are decoded into a neural network that then exhibits some behaviour. The structure of a neural network is effectively limitless. On the other hand, the sensor type, bearing, and orientation describes every detail about a certain sensor while providing only a few axes along which to optimise. This is understandable given that generally sensors are fixed parts and evolving more properties of the sensors becomes difficult to justify as being realistic. Finding a balance between the evolution of more advanced sensors and what is possible given the available hardware is a key challenge in evolving robot morphologies.

Evolving robot morphologies is an important step in truly automating the design of robotic agents. NEAT-M proves that sensor evolution is possible using the NEAT framework and may be a useful solution if further research is able to refine the method’s parameters and improve its performance.

References

T. Nagao A. Hara. Emergence of cooperative behavior using adg: Automatically defined groups. In *Proceedings of the 1999 Genetic and Evolutionary Computation Conference (GECCO-99)*, 1999.

Thomas Bäck. *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms.* Oxford University Press, Oxford, UK, 1996. ISBN 0-19-509971-0.

Tucker Balch. Learning roles: Behavioral diversity in robot teams. *College of Computing Technical Report GIT-CC-97-12, Georgia Institute of Technology, Atlanta, Georgia*, 73, 1997.

Tucker Balch. The impact of diversity on performance in multi-robot foraging. In *Proceedings of the third annual conference on Autonomous Agents*, pages 92–99. ACM, 1999.

Gianluca Baldassarre, Stefano Nolfi, and Domenico Parisi. Evolving mobile robots able to display collective behaviors. *Artificial life*, 9(3):255–267, 2003.

Andrew G Barto. *Reinforcement learning: An introduction.* MIT press, 1998.

Christopher M. Bishop. *Pattern Recognition and Machine Learning.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006. ISBN 0387310738.

Gunnar Buason and Tom Ziemke. Co-evolving task-dependent visual morphologies in predator-prey experiments. In *Proceedings of the 2003 International Conference on Genetic and Evolutionary Computation: PartI*, GECCO'03, pages 458–469, Berlin, Heidelberg, 2003. Springer-Verlag. ISBN 3-540-40602-6. URL `http://dl.acm.org/citation.cfm?id=1761233.1761302`.

Gunnar Buason, Nicklas Bergfeldt, and Tom Ziemke. Brains, bodies, and beyond: Competitive co-evolution of robot controllers, morphologies and environments. *Genetic Programming and Evolvable Machines*, 6(1):25–51, March 2005. ISSN 1389-2576. doi: 10.1007/s10710-005-7618-x. URL `http://dx.doi.org/10.1007/s10710-005-7618-x`.

Christoph Burgmer. Diagram of an artificial neuron., July 2005. URL `http://commons.wikimedia.org/wiki/File:ArtificialNeuronModel_english.png`.

Erin Catto. Box2d: A 2d physics engine for games, 2014. URL `http://box2d.org`.

Thomas P. Caudell and Charles P. Dolan. Parametric connectivity: Training of constrained networks using genetic algorithms. In *Proceedings of the 3rd International Conference on Genetic Algorithms*, pages 370–374, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc. ISBN 1-55860-066-3.

D. Cliff, P. Husbands, and I. Harvey. Analysis of evolved sensory-motor controllers. In *Proceedings of Second European Conference on Artificial Life (ECAL93)*, pages 24–26, 1992a.

Dave Cliff, Philip Husbands, and Inman Harvey. Evolving visually guided robots. In *Proceedings of SAB92, the Second International Conference on Simulation of Adaptive Behaviour*, 1992b.

K-Team Corporation. Specifications, October 2014. URL `http://www.k-team.com/mobile-robotics-products/khepera-iii/specifications`.

George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

Agoston E. Eiben and J. E. Smith. *Introduction to Evolutionary Computing.* Springer Berlin Heidelberg, 2003. ISBN 3540401849.

Manfred Eigen. *Ingo Rechenberg Evolutionsstrategie Optimierung technischer Systeme nach Prinzipien der biologischen Evolution.* mit einem Nachwort von Manfred Eigen, Friedrich Frommann Verlag, Struttgart-Bad Cannstatt, 1973.

Jacques Ferber. *Multi-Agent Systems: an Introduction to Distributed Artificial Intelligence*, volume 1. Addison-Wesley Reading, 1999.

D. B. Fogel, L. J. Fogel, and V. W. Porto. Evolving neural networks. *Biol. Cybern.*, 63(6):487–493, September 1990. ISSN 0340-1200. doi: 10.1007/BF00199581. URL `http://dx.doi.org/10.1007/BF00199581`.

Lawrence J Fogel, Alvin J Owens, and Michael J Walsh. *Artificial Intelligence through Simulated Evolution.* John Wiley & Sons, 1966.

Jason Gauci and Kenneth Stanley. Generating large-scale neural networks through discovering geometric regularities. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 997–1004. ACM, 2007.

Jelena Godjevac and Nigel Steele. Neuro-fuzzy control of a mobile robot. *Neurocomputing*, 28(1):127–143, 1999.

Dani Goldberg and Maja J. Matarić. Robust behavior-based control for distributed multi-robot collection tasks. *Robot Teams: From Diversity to Polymorphism*, 2000.

Faustino Gomez, Jürgen Schmidhuber, and Risto Miikkulainen. Efficient non-linear control through neuroevolution. In *Machine Learning: ECML 2006*, pages 654–662. Springer, 2006.

Frédéric Gruau, Darrell Whitley, and Larry Pyeatt. A comparison between cellular encoding and direct encoding for genetic neural networks. In *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation*, GECCO '96, pages 81–89, Cambridge, MA, USA, 1996. MIT Press. ISBN 0-262-61127-9.

Jeff Heaton. Encog machine learning framework, 2014. URL `http://www.heatonresearch.com/encog`.

John H Holland. *Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence.* U Michigan Press, 1975.

Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4(2):251–257, 1991.

Tomonori Sakamoto Kazuhiro Ohkura, Toshiyuki Yasuda and Yoshiyuki Matsumura. Evolving robot controllers for a homogeneous robotic swarm. In *Proceedings of 2011 IEEE/SICE International Symposium on System Integration*, 2011.

John R Koza. *Genetic programming: on the programming of computers by means of natural selection*, volume 1. MIT press, 1992.

F. Lambercy and J. Tharin. *Khepera III User Manual.* K-Team Corporation, February 2013. Version 3.5.

Martin Lauer and Martin Riedmiller. An algorithm for distributed reinforcement learning in cooperative multi-agent systems. In *In Proceedings of the Seventeenth International Conference on Machine Learning*, pages 535–542, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

Wei-Po Lee, John Hallam, and Henrik H Lund. A hybrid gp/ga approach for co-evolving controllers and robot bodies to achieve fitness-specified tasks. In *Proceedings of IEEE International Conference on Evolutionary Computation, 1996.*, pages 384–389. IEEE, 1996.

Kian Hsiang Low, Wee Kheng Leow, and Marcelo H Ang Jr. A hybrid mobile robot architecture with integrated planning and control. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 1*, pages 219–226. ACM, 2002.

Sean Luke. *Multiagent Simulation and the MASON Library.* Department of Computer Science, George Mason University, August 2014. Version 18.

Sean Luke, Charles Hohn, Jonathan Farris, Gary Jackson, and James Hendler. Co-evolving soccer softbot team coordination with genetic programming. In *RoboCup-97: Robot soccer world cup I*, pages 398–411. Springer, 1998.

Sean Luke, Claudio Cioffi-Revilla, Liviu Panait, Keith Sullivan, and Gabriel Balan. Mason: A multiagent simulation environment. *Simulation*, 81(7):517–527, 2005.

Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, 5(4):115–133, 1943.

Air Ultrasonic Ceramic Transducers. Midas Components Ltd.

Risto Miikkulainen. Neuroevolution. In *Encyclopedia of Machine Learning*, pages 716–720. Springer, 2010.

David J Montana and Lawrence Davis. Training feedforward neural networks using genetic algorithms. In *IJCAI*, volume 89, pages 762–767, 1989.

Daniel Murphy. Jbox2d: A java physics engine, 2014. URL `http://www.jbox2d.org`.

Stefano Nolfi and Dario Floreano. *Evolutionary Robotics: The Biology, Intelligence, and Technology of Self-Organizing Machines.* MIT Press, Cambridge, MA, USA, 2000.

Stefano Nolfi, Dario Floreano, Orazio Miglino, and Francesco Mondada. How to evolve autonomous robots: Different approaches in evolutionary robotics. In *Artificial Life IV*, pages 190–197. Cambridge, MA, 1994.

Esben H Østergaard, Gaurav S Sukhatme, and Maja J Matari. Emergent bucket brigading: a simple mechanism for improving performance in multi-robot constrained-space foraging tasks. In *Proceedings of the fifth international conference on Autonomous agents*, pages 29–30. ACM, 2001.

Liviu Panait and Sean Luke. Cooperative multi-agent learning: The state of the art. *Autonomous Agents and Multi-Agent Systems*, 11(3):387–434, 2005.

Mitchell A Potter and Kenneth A De Jong. A cooperative coevolutionary approach to function optimization. In *Parallel Problem Solving from Nature—PPSN III*, pages 249–257. Springer, 1994.

Mitchell A Potter and Kenneth A De Jong. Cooperative coevolution: An architecture for evolving coadapted subcomponents. *Evolutionary computation*, 8(1):1–29, 2000.

Matt Quinn. A comparison of approaches to the evolution of homogeneous multi-robot teams. In *Evolutionary Computation, 2001. Proceedings of the 2001 Congress on*, volume 1, pages 128–135. IEEE, 2001.

Matt Quinn, Lincoln Smith, Giles Mayley, and Phil Husbands. Evolving controllers for a homogeneous system of physical robots: Structured cooperation with minimal sensors. *Philosophical Transactions of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences*, 361(1811):2321–2343, 2003.

Nicholas J Radcliffe. Genetic set recombination and its application to neural network topology optimisation. *Neural Computing & Applications*, 1(1):67–90, 1993.

Steven F Railsback, Steven L Lytinen, and Stephen K Jackson. Agent-based simulation platforms: Review and development recommendations. *Simulation*, 82(9):609–623, 2006.

Nornadiah Mohd Razali and Yap Bee Wah. Power comparisons of shapiro-wilk, kolmogorov-smirnov, lilliefors and anderson-darling tests. *Journal of Statistical Modeling and Analytics*, 2(1):21–33, 2011.

Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2 edition, 2003. ISBN 0137903952.

Jacob Schrum and Risto Miikkulainen. Evolving multimodal behavior with modular neural networks in ms. pac-man. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2014)*, pages 325–332, Vancouver, BC, Canada, July 2014. URL `http://www.cs.utexas.edu/users/ai-lab/?schrum:gecco2014`.

Keith Sullivan Sean Luke, Gabriel Catalin Balan and Liviu Panait. Mason multiagent simulation toolkit, 2014. URL `http://cs.gmu.edu/ eclab/projects/mason/`. Version 18.

Samuel Sanford Shapiro and Martin B Wilk. An analysis of variance test for normality (complete samples). *Biometrika*, pages 591–611, 1965.

M Srinivas and LM Patnaik. Learning neural network weights using genetic algorithms-improving performance by search-space reduction. In *Neural Networks, 1991. 1991 IEEE International Joint Conference on*, pages 2331–2336. IEEE, 1991.

Kenneth Stanley. Nnrg software - neat c++, 2001. URL `http://nn.cs.utexas.edu/?neat-c`.

Kenneth Stanley. The neuroevolution of augmenting topologies (neat) users page, 2013. URL `http://www.cs.ucf.edu/ kstanley/neat.html`.

Kenneth O. Stanley. *Efficient Evolution of Neural Networks Through Complexification.* PhD thesis, Department of Computer Sciences, The University of Texas at Austin, 2004. URL `http://nn.cs.utexas.edu/?stanley:phd2004`.

Kenneth O Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127, 2002.

Kenneth O Stanley and Risto Miikkulainen. Competitive coevolution through evolutionary complexification. *J. Artif. Intell. Res.(JAIR)*, 21:63–100, 2004.

Kenneth O Stanley, Bobby D Bryant, and Risto Miikkulainen. Real-time neuroevolution in the nero video game. *Evolutionary Computation, IEEE Transactions on*, 9(6):653–668, 2005.

Kenneth O. Stanley, David B. D'Ambrosio, and Jason Gauci. A hypercube-based encoding for evolving large-scale neural networks. *Artif. Life*, 15(2):185–212, April 2009. ISSN 1064-5462.

Wolfgang Stolzmann. Latent learning in khepera robots with anticipatory classifier systems. In *Proceedings of the 1999 Genetic and Evolutionary Computation Conference Workshop Program*, pages 290–297, 1999.

Ming Tan. Readings in agents. chapter Multi-agent Reinforcement Learning: Independent vs. Cooperative Agents, pages 487–494. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993. ISBN 1-55860-495-2.

Reflective Optical Sensor with Transistor Output. Vishay Intertechnology, Inc., August 2009. Rev. 1.7.

Markus Waibel, Laurent Keller, and Dario Floreano. Genetic team composition and level of selection in the evolution of cooperation. *Evolutionary Computation, IEEE Transactions on*, 13(3):648–660, 2009.

Shimon Whiteson, Kenneth O. Stanley, and Risto Miikkulainen. Automatic feature selection in neuroevolution. In *Genetic and Evolutionary Computation Conference*, pages 1225–1232. ACM Press, 2005.

Darrell Whitley, Timothy Starkweather, and Christopher Bogart. Genetic algorithms and neural networks: Optimizing connections and connectivity. *Parallel computing*, 14 (3):347–361, 1990.

Marco Wiering, Rafał Sałustowicz, and Jürgen Schmidhuber. Reinforcement learning soccer teams with incomplete world models. *Autonomous Robots*, 7(1):77–88, 1999.

Michael Wooldridge. *An Introduction to Multiagent Systems.* John Wiley & Sons, 2009.

Xin Yao. Evolving artificial neural networks. *Proceedings of the IEEE*, 87(9):1423–1447, 1999.

Xin Yao and Yong Liu. A new evolutionary system for evolving artificial neural networks. *Neural Networks, IEEE Transactions on*, 8(3):694–713, 1997.

Byoung-Tak Zhang and Dong-Yeon Cho. Co-evolutionary fitness switching: Learning complex collective behaviors using genetic programming. *Advances in Genetic Programming*, 3:425, 1999.