Cooperation versus competition in multi-agent evolutionary systems

Honours Project Report

Alexander Borysov

Supervised by Dr. Geoff Nitschke

October, 2014

Category	**Min**	**Max**	**Chosen**
Requirement Analysis and Design	0	20	0
Theoretical Analysis	0	25	0
Experiment Design and Execution	0	20	15
System Development and Implementation	0	15	10
Results, Findings and Conclusion	10	20	15
Aim Formulation and Background Work	10	15	10
Quality of Report Writing and Presentation	10		10
Adherence to Project Proposal and Quality of Deliverables	10		10
Overall General Project Evaluation	0	10	10
Total Marks	**80**		**80**

Abstract

This report investigates the minimal sensor requirements in terms of type and number that are necessary for a multi-agent system, working on solving a modified foraging task, to evolve cooperative behaviour. We test 8 different sensor morphologies, and show that depending on the types of sensors used, cooperative behaviour can be evolved using only a small number of sensors.

Acknowledgements

I would like to thank my supervisor, Dr. Geoff Nitschke for giving me access to a cluster for performing the simulation runs, as well as all the feedback and advice he provided over the course of this project. I would also like to thank my other two team members, Jamie Hewland and Mary Hsu, for sharing with me the formidable task of writing the simulator.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

Multi-agent system (MAS) research is frequently justified by drawing parallels to real-life swarm behaviours in biological creatures such as ants or bees, claiming that if we were able to reproduce such behaviour in a robotic setting, it could lead to complex, cooperative task-solving capabilities using only relatively simple and cheap individual agents. However, the aspect of agent design for such a cooperative system, in terms of the hardware installed on the physical robot, is not discussed in much depth in typical current literature (Bryant and Miikkulainen 2003)(Waibel, Keller, and Floreano 2009). Often, a particular agent design is used without much justification, presumably after some preliminary testing or research to come up with a reasonable design.

This project thus aims to explore the space of agent design by testing how well various possible designs perform in a cooperative multi-agent task, specifically focusing on the the agents' sensor morphology (the type and number of sensors on the agent). Because the reduction of hardware complexity is an oft-cited basis for MAS research (Panait and Luke 2005)(Nolfi and Floreano 2001), we will focus on minimising the hardware requirements while retaining the ability to evolve cooperation. The research question thus becomes, what is the minimal morphological complexity that can still lead to the evolutionary emergence of cooperative behaviour in a multi-agent system?

Our approach to this question is creating a hand-crafted list of potential agent morphologies of increasing complexity, ranging from, for instance, 1 small sensor in the front, to arrays of sensors covering 360° around the agent. The sensors are selected from a small predefined set of reasonably realistic sensors; these sensor morphologies are then tested out on a series of tasks that aim to measure the levels of cooperation and task fitness these morphologies are capable of.

For this project, the classical foraging task was chosen; this task does not usually require any cooperation at all for successful completion, but it has been modified to require varying levels of cooperation, by introducing three different sizes of resources, with the larger resources requiring more agents pushing in order to move it. Our aim in task selection was to select a task that allows both cooperation and competition to be viable, and yet can be adjusted to favour one or the other to different extents. We have accomplished this by setting up three variants on the classical foraging task; each one has different ratios of smaller to larger boxes, which varies the amount of cooperation required for success. By measuring the fitness differences between less and more cooperatively-demanding tasks, we can determine how effectively a given morphology can cooperate.

An important consideration in the design of the tasks and experiments is the notion of cooperation versus competition. In order to be able to draw a meaningful distinction, the agents need to have a choice between cooperating and competing. We have done this by allowing a resource size that can be carried by a single agent in two of the task setups. This means that it is possible for selfish agents to evolve; agents that prioritise

foraging the small resources over helping other agents carry large ones. However, in the bigger scheme of things, this is not as efficient as cooperation, especially in the more difficult task setups. Our research question will then uncover which morphologies are capable of evolving cooperation.

The agent controllers have been evolved using Genetic Programming (GP). Compared to the more typical neural network based methods of controller evolution, GP is is a very flexible and powerful method of evolving robotic controllers, with the disadvantage of frequently having an untenably large state-space to search through, with most solutions being nonsensical, or at least extremely bad. In this project, a variant of GP called Strongly Typed Genetic Programming is used, which improves on standard GP by imposing types on every input, output and intermediate value used in a GP tree. This drastically reduces the number of valid function combinations, and thus, the state space.

2 Background

Biologically inspired design, such as Evolutionary Algorithms, is one approach to constructing multi-agent systems. Autonomous agents in such systems must be capable of solving complex cooperative tasks, called collective behaviour tasks. MAS is very well suited to such tasks (Arkin and Balch 1998), and as such, collective behaviour tasks are widely studied in this research field. The inspiration for biological approaches to MAS is drawn from real-life creatures that exhibit complex swarm behaviour, such as ants or bees. These manage to perform very complex tasks without any sort of central knowledge or management, using very simple agent design that results in emergent system-wide behaviour. Being able to replicate this in the field of artificial intelligence could allow the creation of simple solutions to complex or distributed problems (Arkin and Balch 1998).
We will briefly cover the general field of MAS, with a more detailed overview of machine learning, and more specifically, evolutionary approaches.
We will first review approaches to single-agent autonomous systems, then, machine learning (ML) approaches, specifically focusing on Genetic Programming (GP). We proceed to narrow down to MAS specifically, going through the added difficulties presented by the involvement of multiple agents. Finally, we cover some of the tasks that MAS is commonly applied to, and discuss the choice between testing on real life robots versus testing on simulations.

2.1 Autonomous agent design

One of the more interesting parts of an autonomous agent is the controller: the logic driving the motor outputs of the agent based on its sensor inputs. It is based on the complexity of designing such a controller by hand, especially if one desires emergent behaviour in groups of simple robots, that machine learning approaches to controller design have been studied (Reeve and Hölldobler 2007).

There are three broad categories of Machine Learning algorithms: supervised, unsupervised, and reinforcement learning (Panait and Luke 2005).

2.1.1 Supervised learning

Supervised learning relies on supplying the learner with a set of known correct input-output pairs, with the intention that it is then able to extrapolate when faced with new data, and react to unseen inputs in a similar way to the examples.
For MASs, it proves very difficult to supply a machine learner with correct example outputs, partially due to the very large space of possible actions, and partially due to the many different ways of obtaining the same goal. Due to these difficulties, very little literature has explored this machine learning approach in MAS research (Panait and Luke 2005).

2.1.2 Unsupervised learning

In unsupervised learning, no feedback is provided about the agent's performance at all. Common tasks in this category include clustering and dimensionality reduction, for all of which you do not need labeled data. This is not a suitable paradigm for the kinds of tasks a MAS is applied for, as solutions will likely be inadequate for MAS tasks, and as such will not be able to attain good performance without any kind of feedback. Additionally, MAS tasks can usually can give agents feedback fairly easily, which in turn permits more intelligent and complex solution searching via reward-based learning.

2.1.3 Reinforcement Learning

Reward-based approaches involve the system giving reward to an agent, either at every iteration, or after a complete agent lifetime. We review two particular reward-based approaches, reinforcement learning and evolutionary computation. Reinforcement Learning, in particular, relies on providing realtime feedback to the agents as to their performance, thus allowing the agents to build up a model of which actions lead to desirable outcomes. The two most commonly used Reinforcement Learning methods are Q-Learning and Temporal-Difference Learning (Busoniu, Babuska, and De Schutter 2008).

Q-Learning: Q-Learning relies on maintaining a table mapping from state-action pairs to expected future reward. This can be used at every state to select the action with the highest expected future reward. Then, upon receiving the actual reward, the estimate is adjusted (Busoniu, Babuska, and De Schutter 2008). Q-Learning provably converges on an optimal state-action table under assumption that the problem is a finite Markov Decision Process (MDP).
This approach relies on having a relatively small, discrete state and action space, and on your problem being an MDP. Neither of these facts hold in a MAS; the state and action spaces are usually very large and continuous, and a typical MAS is very poorly

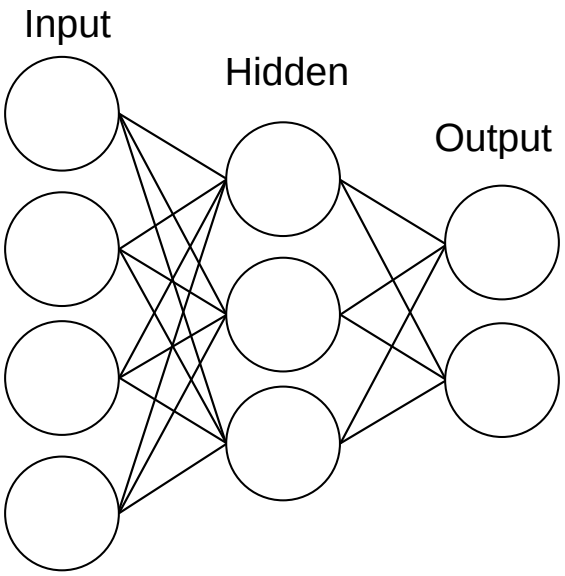


Figure 1: Example ANN; 4 inputs, 3 hidden nodes, and 2 outputs (weights omitted)

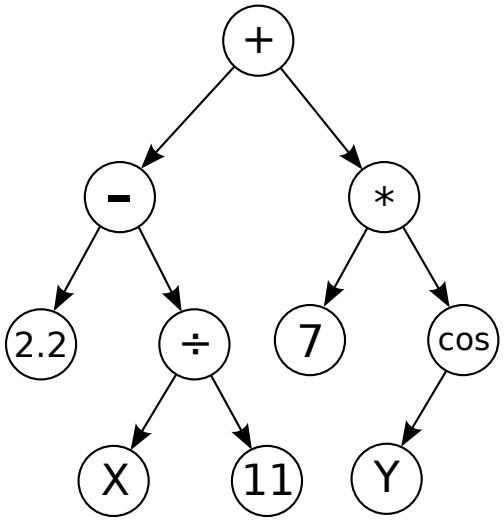
described as an MDP (Tuyls, Verbeeck, and Lenaerts 2003). This makes Q-Learning a very poor fit for MAS, and it has been studied accordingly little in current literature.

Temporal-Difference Learning: Temporal-Difference learning is very similar to Q-Learning, but it additionally uses current reward to update past predictions of the current reward using the discount factor (Sutton 1988). It remains unsuitable for MAS for the reasons outlined above.

2.1.4 Evolutionary Computation

Evolutionary computation is a reward-based approach (Panait and Luke 2005) that involves using the concept of evolution to manipulate a pool of candidate solutions to progressively improve them according to some measure of the solution's effectiveness at solving the problem – called a fitness function (Eiben and Smith 2003). In the case of autonomous agents, the solutions are the controllers. There are two major approaches to representing and evolving these controllers in the context of Evolutionary Robotics.

Neuroevolution: One way of representing controllers is Artificial Neural Networks (ANNs) (Floreano, Mondada, et al. 1994). These simulate neurons as they are posited to work in a biological brain: several levels of artificial neurons, represented by abstract nodes, are fully connected to each neighbouring layer (figure 1). Each edge has an associated weight; these are the values that change as the ANN learns. At each step, the values of the input nodes are multiplied by the edge weight, summed together, and passed through an 'activation function', a function that modifies the sum in some way before passing it onto the next neuron; usually, this takes the form of a smoothed step



$$(2.2 - (\frac{X}{11})) + (7 * \cos(Y))$$

Figure 2: An Example GP candidate. This tree could have been produced from a syntax consisting of various arithmetic functions, the literal values 2.2, 11 and 7, and the two inputs X and Y.

function.
ANNs have proven to be fairly robust and effective at solving a wide range of noisy and complex problems (Hunt et al. 1992). They have also proven to work as controllers for robotic agents (Floreano, Mondada, et al. 1994).
Neuroevolution uses evolutionary algorithms to evolve the network: the weights, and sometimes the topologies of the networks are evolved to optimise for fitness (Floreano, Mondada, et al. 1994).

Genetic Programming: GP is an evolutionary computation technique that operates on a candidate pool of computer programs (John R Koza 1990). Specifically, in the case of GP, the programs are represented using a tree of functions. This fits the needs of autonomous agents, as a computer program is certainly capable of acting as a controller for an agent. GP represents a function from the inputs to the outputs as a tree of smaller composed functions and terminals (input nodes or literal values, these will form the leaves of the tree), selected from some predetermined set, called the syntax (John R Koza 1990); see figure 2. A population of such trees is then evolved using standard genetic techniques, with special consideration made for the tree structure when crossing over and mutating (John R Koza 1990).
This is a general technique for building flexible controllers for autonomous agents, with the disadvantage of potentially infeasible convergence times depending on symbol choice and task complexity (Matarić and Cliff 1996; Haynes et al. 1995).

Haynes et al. (1995) present an enhancement on GP where they use strong typing on the nodes of the tree to drastically reduce the size of the state space, causing a significant increase in the learning rate, outperforming standard GP.
Matarić and Cliff (1996) show that GP has potential for performance improvements if a high-level set of symbols and terminals is chosen. However, they warn against introducing too much domain knowledge through biased choice of symbols and terminals.

2.2 Multi-agent systems

We now consider multi-agent systems proper; there are many additional considerations brought in by the introduction of multiple agents.
It is worth noting that we will only deal with evolutionary approaches to MAS; in fact, many of the following taxonomical divisions only make sense with an evolutionary approach.

2.2.1 Styles of cooperation

One of the big goals of MAS is to reproduce the emergent behaviour we have observed in biological systems. Doran et al. (1997) propose a classification of styles of cooperation in multi-agent systems (figure 3).
Independent systems are systems where no explicit cooperation occurs. However, agents can still unintentionally cooperate (dubbed emergent cooperation), if for instance, every agent is assigned a strict subtask that, when completed, plays its part in the whole. The agent has no knowledge or intention of cooperation, and yet to the casual observer, the system appears to be cooperative.
Cooperative systems are divided into communicative versus non-communicative systems. This will be described further in section 2.2.3, but in brief, a communicative system is one where agents explicitly send and receive messages to one another; non-communicative systems rely on simply observing other agents and their behaviours in the environment to infer their goals and predict their future actions (Doran et al. 1997).

2.2.2 Learning approaches

Waibel, Keller, and Floreano (2009) separate learning approaches by two axes: the genetic team composition and the level of selection (see figure 4).
There are two approaches to the genetic composition of an agent team:

Homogeneous: The most common approach in research, a homogeneous team composition is one where the entire team is composed of identical copies of the same genotype. Note that they still can, and most likely will, act differently because of different starting conditions and experiences. This approach is commonly favoured, because it greatly reduces the state space on a team level compared to heterogeneous teams Panait and Luke 2005.

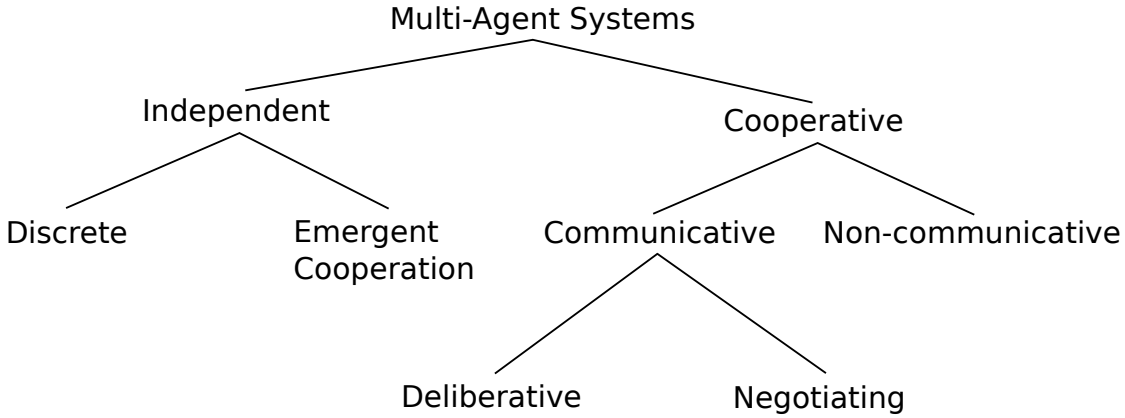


Figure 3: Doran et al.'s (1997) cooperation classification

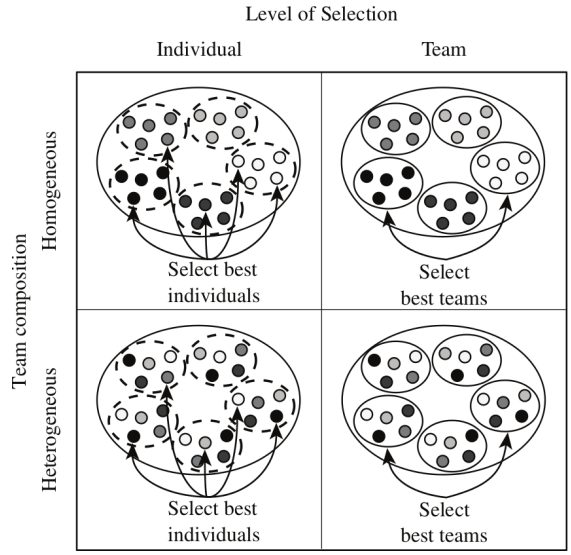


Figure 4: Waibel, Keller, and Floreano's (2009) taxonomy of learning and selection approaches

Heterogeneous: The alternative is a heterogeneous team, where every individual genotype in the team is permitted to be different. This obviously has the advantage over homogenous teams that teams can evolve to have 'specialists', that is, individuals with different behaviour from everybody else, that fulfil some problem-specific niche in a way a homogenous team could not (Luke and Spector 1996). However, it has been shown by Waibel, Keller, and Floreano (2009) that this comes at the heavy cost of learning rate, and in fact, if the problem domain does not reward specialisation sufficiently, it can also result in a lower final fitness. For this reason most of the multi-agent literature focuses on homogeneous teams (Waibel, Keller, and Floreano 2009).
There are two means of agent selection in a heterogeneous task setup:

Individual selection: Individual selection assigns each individual in a team its own fitness value, and performs selection on an individual level.

Team selection: Team selection does not distinguish between individuals in a team; it assigns a single fitness to the entire team based on how it performed. This serves as encouragement for agents to cooperate; however, Waibel, Keller, and Floreano (2009) show that team selection, if combined with heterogeneous teams, leads to inefficient selection and very poor solutions. The selection type did not make a big difference for homogeneous teams.

Credit assignment: If using individual selection, in order to keep cooperation incentivised, a modified credit assignment scheme is sometimes needed. Much like the choice between team and individual selection, the problem of credit assignment is an issue of balancing between assigning all reward to the team as a whole (global reward), or assigning all reward precisely to the agent who earned it (local reward). While global reward encourages cooperation, it also fails to punish laziness (Panait and Luke 2005), and in more complex tasks, does not give sufficient feedback about a particular agent's specific actions (Wolpert and Tumer 2001).
Moreover, certain tasks are better suited to certain credit assignment schemes. Balch (1999) shows that global reward is much more effective in soccer (a strongly cooperative team activity), whereas his results showed foraging actually performing better with local reward, foraging being an activity that can often be accomplished to a great degree of success by a single agent.

2.2.3 Communication

In MAS, an important consideration is the degree of communication allowed between the agents. Unrestricted global communication is not interesting, because it effectively reduces the multi-agent problem to a single-agent problem. Every agent can simply communicate its sensor readings to every other agent, effectively making the multi-agent system a single omniscient master with multiple physically separated hardware appendages, which can be trained in much the same way a single agent would (Panait

and Luke 2005).
Literature in the field studies means of communication that are bound by one or more parameters, such as throughput, latency or locality (Panait and Luke 2005). This communication is either hard-coded into the agent beforehand, or learnt as part of the evolution. The latter approach has been shown to have issues with a greatly enlarged action and state space, which then significantly slows the learning process. For this reason, many researchers have opted to disregard communication if it is not critical to solving the problem (Panait and Luke 2005).
While it may at first seem that communication is central to cooperation between different agents, a lot of research gets good results just using indirect communication (that is, relying on detecting the other agents' positions, and the changes they make to the environment, without worrying about explicit communication) (Waibel, Keller, and Floreano 2009; Panait and Luke 2005).

2.3 Collective Behaviour Tasks

Different tasks inherently promote differing levels of competition versus cooperation. There is great variation in performance of different approaches depending on task (Panait and Luke 2005). Different tasks are biased for cooperation to varying extents. Some tasks benefit greatly from specialisation, and thus allow heterogeneous approaches to outperform homogeneous approaches (Panait and Luke 2005). There are several common tasks used in MAS literature (Panait and Luke 2005):

- **Pursuit-Evasion**: Groups of predators cooperate to capture prey by moving on a 2D world in such a way as to trap the prey (Stone and Veloso 2000).
- **Foraging**: A team of agents navigate a map containing interspersed items, and potentially some obstacles, with the goal of foraging these items, bringing them back to a designated base area (Stone and Veloso 2000). Complications such as increased item weight can be introduced in order to encourage agents to cooperate, since the task in its basic formulation does not lend itself to cooperation.
- **Collective Construction**: A team of agents needs to push boxes found in the environment, potentially among obstacles, into a pre-specified pattern at the target location (Panait and Luke 2005). This task requires a lot more cooperation between agents to be successful, because the boxes need to be positioned correctly, so a greedy, selfish approach will not work.
- **RoboCup Soccer**: Two teams of agents playing a simplified game of soccer in a virtual environment (Asada and Kitano 1999). Most current literature on this task focuses on small teams of 2-3 agents performing various small subtasks of the overall game, as the overall task itself has proven too large and complex for current approaches to solve.

2.4 Testing Platforms

Seeing how the ultimate purpose of MAS research is deployment in multi-robot systems, it would be logical to develop and test the algorithms derived from these methodologies on such a set of robots. Unfortunately, especially with the field of evolutionary MAS, it takes many iterations to obtain a useable result. This makes it incredibly time-consuming to perform the evolutionary process on the robots themselves. However, Miglino, Lund, and Nolfi (1995) show that it is possible to transfer evolved solutions from simulated environments onto real world robots with minimal loss of performance, provided the simulation was sufficiently realistic in its assumptions, and that the evolved solution be allowed to evolve further for a short duration on the real robot. This is a reassuring result, because it means that EC is a viable choice for evolving controllers for real robots, thanks to the ability to rapidly evaluate evolutionary generations on modern computers.

3 Methods

3.1 Robots

The robots have a radius of 15cm, and a mass of 700 g. These values are loosely based on the real-life Khepera III robot (*Khepera III* 2014). They are have two simulated wheels, and zero or more sensors. These sensors are mounted on the circumference of the robot, face outwards, and usually detect objects within a cone determined by the sensor's field of vision (FOV) and range.
The robots experience lesser friction in the axis of the wheel direction, to simulate the reduced friction from wheel rolling.

3.2 GP Method

The robot controller is evolved using Genetic Programming; more specifically, Strongly Typed Genetic Programming[2]. GP is used because of its great flexibility and power. Since we are testing minimal sensor morphologies, it is important to be able to extract as much utility out of a sensor morphology as possible, so a powerful evolutionary technique is necessary. GP is a powerful technique that ought to be more than capable of handling the task.
Here we discuss the various options exposed by GP, both evolutionary and syntactical, and the motivations for the choices made. For a detailed table of parameter values, refer to table 1 in section 4.

3.2.1 Parameters

Selection: Three methods of selection were considered: fitness proportionate selection, linear rank selection, and tournament selection. From the testing done, fitness proportionate selection and linear rank selection severely underperformed, struggling to

[2]A Java library called EpochX (Castle 2012) was used for the GP implementation

```
Data: population
initialize population;
while haven't reached generation limit do
    Data: parent pool
    while parent pool's size is less than the population size do
        choose a tournament from population;
        evaluate the fitness of the tournament participants;
        place the winner into parent pool;
    end
    empty out population;
    while population's size is less than its full size do
        if crossover probability succeeds then
            cross over parents and place into population;
        else
            if mutation probability succeeds then
                mutate parents and place into population;
        end
    end
end
```

Algorithm 1: A GP Run

consistently improve the average fitness of the population; on the other hand, tournament selection worked well. Additionally, choosing tournament selection gave an easy way to tune the selection pressure by varying the tournament size. It was tweaked to find a balance between convergence time and population diversity.

Elitism: During testing without elitism, the population lost track of good solutions very easily, and improved at a very slow rate. This result is corroborated by current GP research, which recommends elitism as a means of not losing good solutions in a large state space (Poli, McPhee, and Vanneschi 2008). Some testing showed that a value of 25% produced good results.

Population size: Past literature shows that extremely large population sizes are greatly beneficial to GP (John R. Koza 1992)(Piszcz and Soule 2006); however, using STGP with a doman-specific, high-level syntax, seems to have reduced this requirement down to a manageable level. Testing showed populations as low as 200 being viable, if somewhat prone to unreliability due to the randomness involved. Improvement in task performance was observed all the way up to 1000, however due to time constraints for finishing enough experiment runs, the actual population size used was 700.

[4]See the RunManager and GenerationManager classes in EpochX (Castle 2012)

```
Input: max depth
Input: return type
Output: tree
while haven't reached max depth do
    Data: candidate nodes
    fill with candidate functions or terminals that return needed return type;
    Data: chosen node
    randomly pick a candidate node;
    for i = input of chosen node do
        recursively generate tree that gives necessary return type for this input;
        add tree as appropriate child to chosen node;
    end
    return chosen node as root of tree;
end
```

Algorithm 2: Generating an n-depth tree (or subtree) in STGP. This procedure takes as parameters the required return type and the maximum depth[4].

Max Tree Depth: This is a limit imposed by GP on the tree depth during mutation and crossover operations. Initialisation was further limited to a depth of 5.

Mutation: Two classic mutation operators for GP were considered: Subtree mutation, which picks a point in the candidate, removes the old contents, and generates a new subtree from that point, and Point mutation, which picks a single point, and simply replaces it with another compatible terminal node. Subtree mutation was chosen as it is the more powerful means of mutation, and one more widely used in GP literature (Luke, Hohn, et al. 1998).

Crossover: Two crossover operators were considered: Subtree Crossover, and One Point Crossover (as introduced by Poli and Langdon (1998)). Subtree crossover was chosen because it is the more established crossover method Poli and Langdon (1998), and performed favourably compared to one point crossover during testing.

3.2.2 Types

The approach taken with the syntax is designing custom types whenever necessary to prevent the possibility of semantically nonsensical trees.

WheelDrive: This is a class that encapsulates a pair of floats clamped to -1 to 1. This type is the ultimate return value of the tree.

ProximityReading: This is a class that stores a sensor reading along with some extra information about the sensor. This allows a ProximityReading object to be converted into a bearing (direction of the reading), or distance (knowing the range of the sensor).

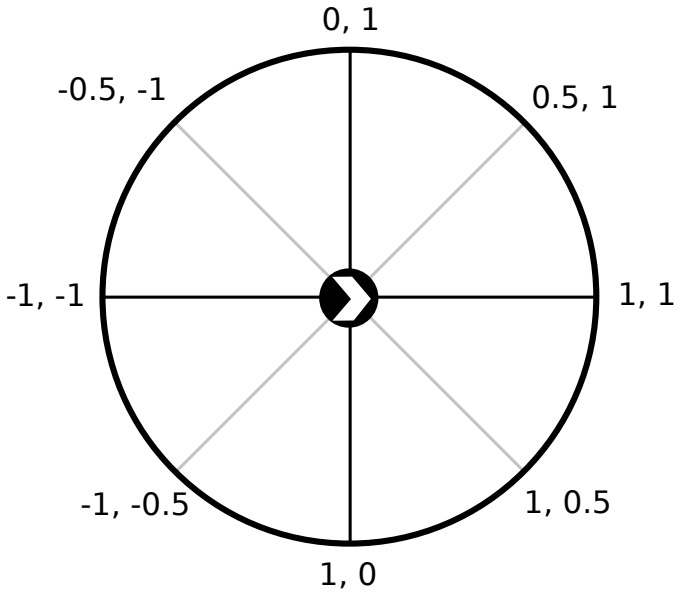


Figure 5: The algorithm for driving an agent to a given bearing; the diagram maps each direction to a pair of wheel drive values. Notice that if the target is behind the agent, it will opt to drive at it backwards instead of turning around to face it first. This is due to the symmetrical design of the wheels and engine: it is no less efficient to reverse than to drive forward.

TypedProximityReading: This is a class much like ProximityReading, except it additionally stores the type of object the reading is about.

Bearing: A simple wrapper around a float value. Represents a bearing from 0 to 2π. This type's existence prevents arbitrary floats from being treated as bearings.

GPFloat, GPFloatLiteral, GPFloatVariable: GPFloat and its subclasses represent a simple floating point value. The GPFloatLiteral subclass is used for the terminal literal float values added to the syntax, whereas the GPFloatVariable subclass is used for every other instance where the value is variable (e.g. a sensor input). This setup allows functions to declare that at least one of their inputs must be non-literal.

DetectedObject.Type: A Java enum listing the types of objects that a typed sensor can detect: RESOURCE, ROBOT, WALL

ScaleFactor: A wrapper around a float, specially made for the ScaleWheelDrive function; see below. All the instances used were literals: 0.25, 0.5 and 0.75. The scaling was intended to mitigate the tendency of the turning algorithm (see figure 5) to drive at full power at all times.

3.2.3 Functions

WheelDriveFromFloats Creates a WheelDrive from a pair of input GPFloats.

WheelDriveFromBearing Returns a WheelDrive directing the agent in the direction of the given Bearing, using the mapping detailed in figure 5

ReadingToFloat Returns the literal reading from the input ProximityReading as a GPFloat.

ReadingToDistance Uses the sensor range knowledge to compute the approximate distance to the detected object, and return it as a GPFloat.

ReadingPresent Returns a Boolean of whether the ProximityReading is non-0.

ReadingIsOfType Given a TypedProximityReading and a Type, returns a Boolean specifying whether they are the same.

ScaleWheelDrive Given a ScaleFactor and a WheelDrive, scales both WheelDrive components by the factor and returns the new WheelDrive.

GT This is a customised greater than function; takes in two GPFloats, ensuring that both are not GPFloatLiteral, and returns a Boolean.

IF Given a Boolean, and two other values (of the same type), returns the first if the boolean is true, otherwise the second.

3.2.4 Terminals

WheelDriveSpotTurnLeft, WheelDriveSpotTurnRight Return WheelDrives of $-1, 1$ and $1, -1$ to perform a turn on the spot left and right, respectively.

RandomBearing This returns a random Bearing on every evaluation.

GPFloatLiteral The values 0.0, 0.5 and 1.0 were added as literals to the syntax.

3.3 Fitness calculation and allocation

Individual fitness: Robot teams are heterogeneous, with individual selection. Individual selection has been shown to be a superior approach to team selection in the case of heterogeneous teams (Waibel, Keller, and Floreano 2009).
In the simulation, every resource has a specific, fixed value (see table 4). For every resource pushed into the target area, the fitness is divided equally between the participants. A robot is deemed to have partricipated in pushing a resource in by a 'blaming'

algorithm; this checks a bounding box around the resource, one of slightly larger size than the resource itself, for agents. If no agents can be found, the box is iteratively increased in size until some are.
A similar process occurs if a resource is pushed out of the target area; the blame algorithm is run, and a fitness to the value of the resource is subtracted among the participants, with the usual fitness sharing rules in place.
This scheme ensures that the total fitness in the system remains constant at all times, and is determined by the quantity and types of resources it was initialised with.

Team fitness: In addition to individual fitness, for every team, a 'team fitness' was calculated, which is simply the sum of all the individual fitnesses, normalised to 100 by the theoretical total of all the resources in the area. Additionally, a 20-point bonus was added in the case that the agents managed to get all of the resources before the simulation time limit was reached. The formula for calculating the bonus is as follows:

$$B = 20 \times (1 - \frac{S_i}{S_t}) \tag{1}$$

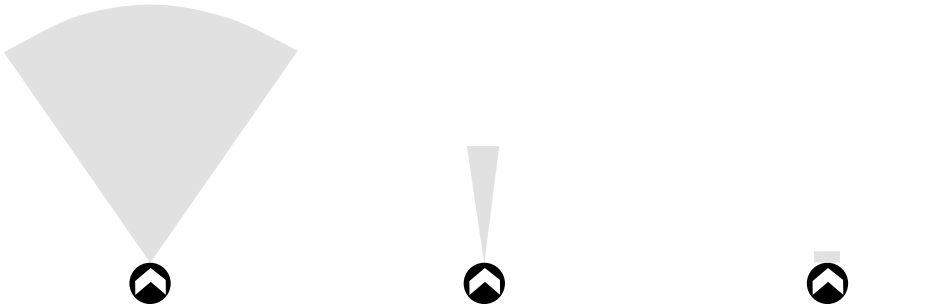
where S_c represents the steps taken by the agents, and S_t is the total number of steps permitted. This means that a team completing the task in 5000 steps would gain a bonus of 10. Reaching the full 20 is impossible, as it requires a team to complete the task in 0 steps.
This 20-point bonus allows us to differentiate between good solutions in the easier test cases, where one would reasonably expect all resources to be collected within the time limit.

The team fitness is not used for selection; only the individual fitnesses are used. Team fitness is merely an indicator of how well the team is cooperating to solve the task. As such, if a team scores well in team fitness, it is an indication that they have evolved cooperation. If they score poorly, this indicates either low individual scores across the team, or the presence of selfish agents; in either case, it is indicative of a lack of cooperation. Thus, team fitness will be the variable that we will investigate in the results section as our primary indicator.

3.3.1 Sensors

Here we describe the basis and purpose of each type of sensor. For detailed operating parameters in the simulation, see section 4.

Ultrasonic Sensor The ultrasonic sensor is based on the real-life Khepera ultrasonic sensor (*Khepera III* 2014). It uses an ultrasound transmitter and receiver to measure the delay between the transmission and receipt of the reflection, thereby determining the distance of the object. The simulation models this using a sensor with a wide FOV and a long range. This sensor returns a reading that linearly scales from 1 if the object is right in front of the sensor, to 0 if the object is past the range of the sensor.



(a) Ultrasonic sensor (b) Typed sensor (c) Pickup sensor

Figure 6: The different sensor types (to scale)

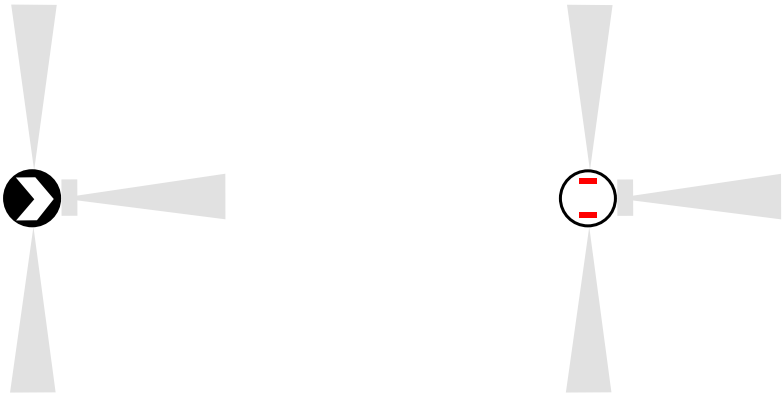
Typed Sensor The 'typed sensor' is a simple narrow-FOV proximity sensor that can detect the type of object it has sensed. Specifically, it can see other agents, resources and walls. The type of the object, and the distance (in the same manner as for the ultrasonic sensor) is reported to the controller. The idea behind this sensor is that STGP can make use of the type information to better inform its decisions, and perhaps even specialise its behaviour based on the type of object it is seeing.
In terms of real-life sensors, such a type-aware sensor can be implemented using a colour or camera sensor that can detect objects that are painted using different colours. Currently, commodity colour sensors have very short ranges, and so are unsuitable for this type of task; however, it is entirely within the realm of engineering to construct such a sensor. With this in mind, it was investigated as part of this report, due to the potential GP has of putting the type information to good use.

Target Area Sensor The target area sensor is a sensor on the bottom of the agent that detects when the robot is in the target area, returning a floating point value depending on how much of the sensor is in the target area. For the purposes of GP, this output is transformed into a boolean by thresholding by 0.5.

Pickup Sensor Additionally, each agent has a mandatory Pickup Sensor. This is a tiny rectangular sensor on the front of the agent that does not provide an output to the controller; instead, whenever it detects a resource, it attaches this resource to the robot, and triggers a heuristic that makes the robot drive towards the target area and drop the resource as soon as it is completely in the target area. This heuristic overrides the controller; that is, the controller is not evaluated at all while the heuristic is active.

3.3.2 Actuators

The robots have only two actuators: the wheels, positioned as indicated in figure 7. These are the only two controller outputs accepted by the robot. These wheel drives are



(a) An agent (b) An agent with the wheel positions marked in red

Figure 7: The wheel positions on the agent

Selection	Tournament selection
Elitism	25% of population
Population Size	700
Generation Limit	70
Max Tree Depth	7
Mutation Operator	Subtree mutation
Mutation Probability	0.1
Crossover Operator	Subtree crossover
Crossover Probability	0.9

Table 1: GP evolution parameters

valued from -1 to 1 for each wheel. This allows robots both to turn, and to reverse by varying the power supplied to each of the wheels.

4 Experiments and Results

For testing our research question, we have selected the foraging task. Furthermore, we set up three variants of the foraging task, of increasing complexity; more specifically, with each one requiring increasingly more cooperation to succeed. Heterogeneous teams of agents are deployed on the three tasks, and those teams whose performance drops off the least, as the tasks become more demanding of cooperation, are deemed the most cooperative. Thus, the relationship between morphological complexity and level of cooperation can be explored, with the ultimate goal of finding an effective minimal morphology for the evolution of cooperative emergent behaviour.
To test the robot teams, a foraging task simulator was developed [5].

[5]Developed in collaboration with Jamie Hewland and Mary Hsu.

Name	FOV	Range
Ultrasonic Sensor	70	4 m
Typed Sensor	15	1 m
Target Area Sensor	N/A	N/A

Table 2: The parameters of the sensor types

	Small Task	Medium Task	Large Task
Small Boxes	15	5	0
Medium Boxes	0	5	5
Large Boxes	0	5	10

Table 3: The resource compositions of the three tasks

4.1 Experiment Design

8 hand-designed agent morphologies of varying complexities were tested; see figure 8 for a detailed list. Each morphology was tested over three task setups, the small, medium and large case. The environment size remained constant at 20×20 m, and the total number of resources remained constant at 15, but the composition of resource sizes was varied. See table 3 for the per-task resource compositions. The simulation was stepped for a fixed 10000 steps, or until all the resources have been collected, whichever one occurrs *earlier*.

4.2 Simulation

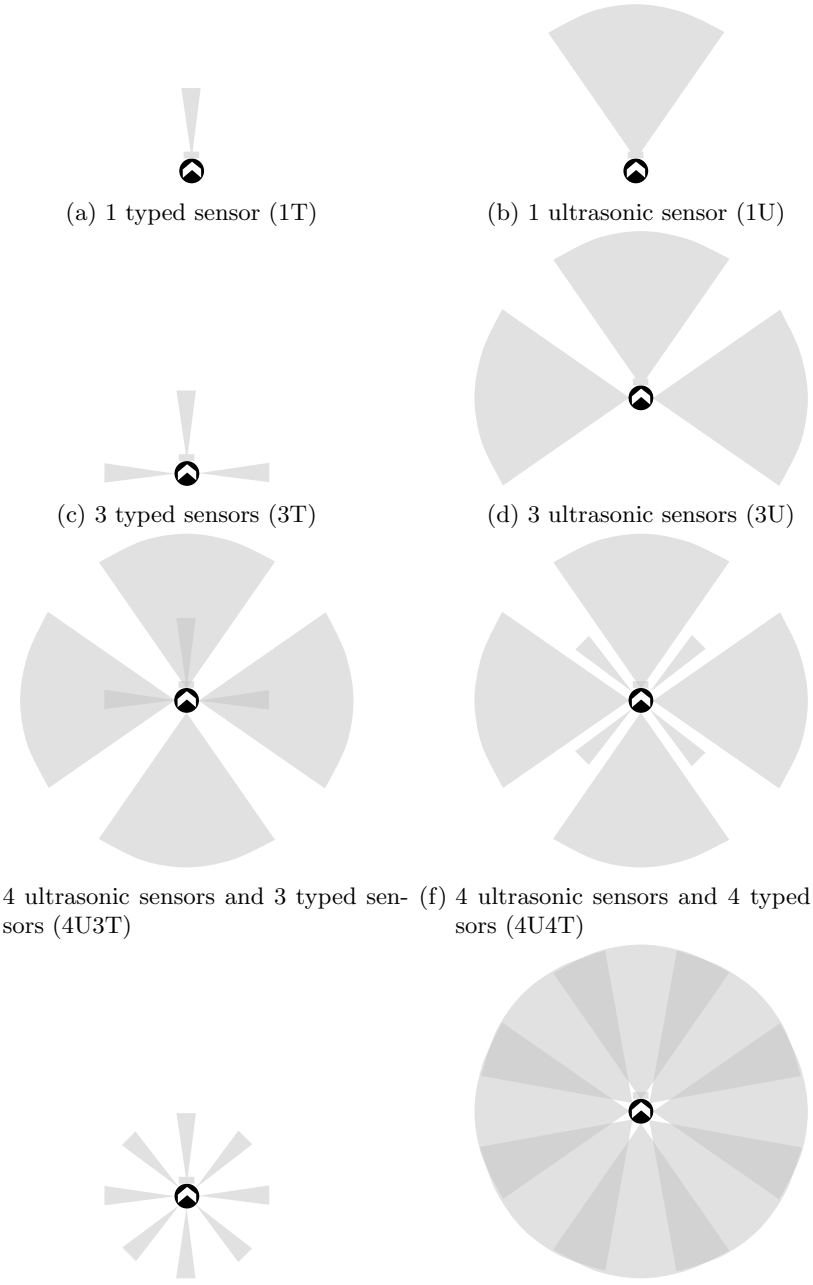
The simulation platform was written in Java. It uses the MASON framework (*MASON Multiagent Simulation Toolkit* 2014) for visualising the simulation. A physics engine called JBox2D (*JBox2D: A Java Physics Engine* 2014) was used for simulating the robot and resource physics.
In general terms, the task setup was as follows:

Environment: The environment is a 20×20 metre area, with walls on all four sides.

Target Area: The target area is a portion of the environment where robots need to drop off the resources. The actual size was set to be the bottom 10% of the environment.

Robots: The robots are circular, and have zero or more sensors attached to them. They are initialised at random positions and with random orientations around the environment. Each robot has two wheels it can use to move itself.

Resources: The resources are square, heavy blocks, intitialised with random positions and orientations. It is the agents' goal to push (or drag) these resources back to the



(a) 1 typed sensor (1T)

(b) 1 ultrasonic sensor (1U)

(c) 3 typed sensors (3T)

(d) 3 ultrasonic sensors (3U)

(e) 4 ultrasonic sensors and 3 typed sensors (4U3T)

(f) 4 ultrasonic sensors and 4 typed sensors (4U4T)

(g) 8 typed sensors (8T)

(h) 8 ultrasonic sensors (8U)

Figure 8: The morphologies tested in this report. In addition to the sensors listed in the descriptions above, every robot had a bottom proximity sensor and a pickup sensor. Listed in brackets is the shorthand for the morphology.

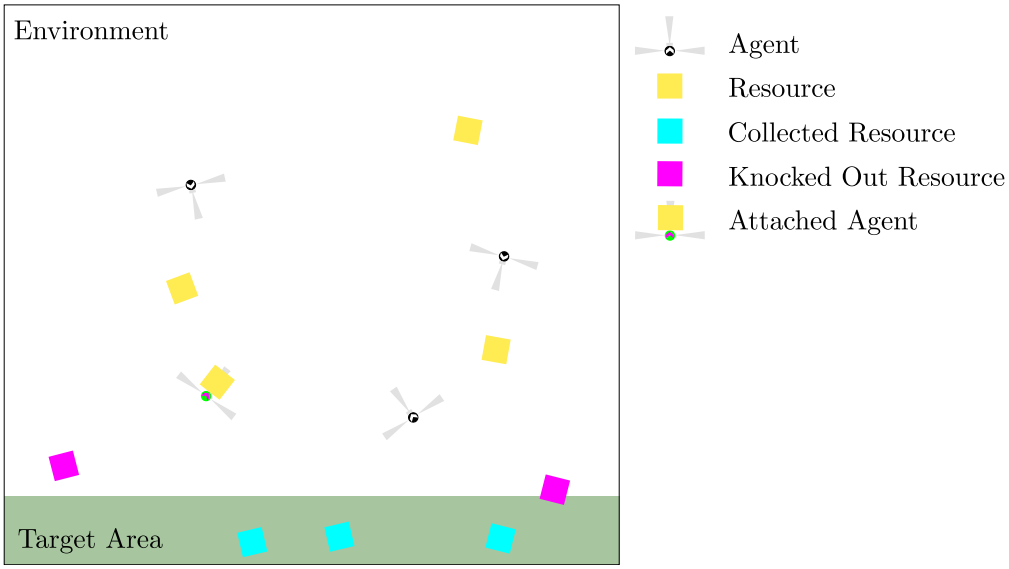


Figure 9: An example simulation environment, populated by agents and resources. Labeled are the environment, the target area, an agent, a resource, a collected resource, a 'knocked out' resource (a resource that used to be collected, but was accidentally pushed out of the target area), and an agent attached to a resource.

target area. There are three resource sizes: small, medium and large, requiring 1, 2 and 3 robots to move respectively.

Resource Size	**Mass (kg)**	**Side Length (m)**	**Bots Required**	**Value**
Small	1	0.4	1	1
Medium	3	0.6	2	2
Large	6	0.8	3	3

Table 4: The parameters of the different resource types

4.2.1 GP Syntax Calibration

Through extensive testing, many functions and terminals that were originally considered as reasonable additions to the syntax, were omitted for the purposes of keeping the state space down. For instance, no arithmetic functions (add, subtract, multiply or divide) or trigonometric functions (sine, cosine, tangent, and so forth) were present in the final syntax. Testing showed that these functions greatly worsened evolution performance, mostly due to the sporadic generation of dead code; branches that multiply or divide constants, or perform various unnecessary operations on a value before passing it along. The approach taken in this project with regard to syntax choice, was one of preferring having a few high-level transformations to having a large set of low-level functions,

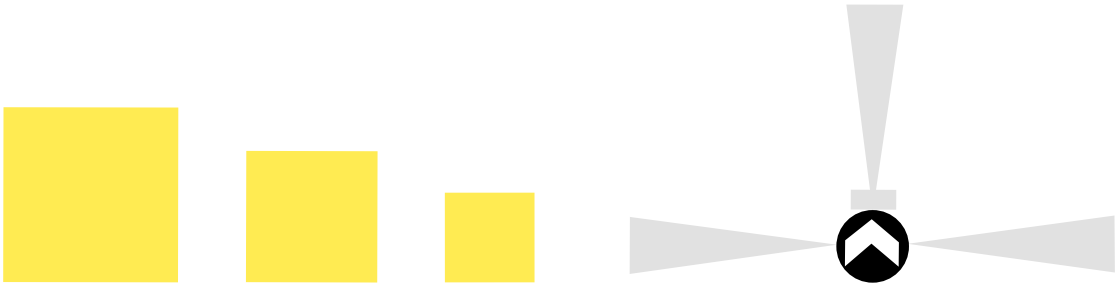


Figure 10: The three resource sizes, with robot for scale

as the latter relies on the evolution being able to compose such low-level functions in an intelligent manner. Creating high-level functions as part of the syntax is a way of encoding domain knowledge, thus reducing the search space.

4.3 Results

Each of the 8 morphologies was evaluated 10 times on each of the 3 task setups, small, medium and large, for a total of 240 simulation runs. See figure 11 for the team fitness results: the maximum team fitness at the end of 70 generations, averaged over 10 runs, sorted by large case performance. The overall result, is all the morphologies performing excellently on the small case, finishing the task in less than half the allocated time (derived from the score allocation outlined in section 3.3). A handful of the best morphologies exhibit similarly excellent behaviour on medium, and all the tested morphologies perform relatively poorly on the large case.
The individual results graph (figure 12) shows somewhat different results: in particular, 8U is a clear outlier in the small task; this is potentially due to its large array of sensors, which it can use to detect resources more quickly than its competitors. Additionally, because the small task requires no cooperation, once an agent has detected a resource, it is a trivial task to collect it. However, as is seen from the team fitness graph, this excellent selfish performance did not amount to any sort of improvement over the other morphologies in terms of team fitness.

5 Discussion, Conclusions and Future Work

5.1 Discussion

In order to answer the research question, it is necessary to look at the morphologies in order of increasing complexity, in order to observe the point at which introducing additional sensors has comparatively little benefit. Our base cases are thus 1T and 1U. Both have underperformed in the medium and large cases; however, upscaling by just two sensors to 3U yields the best score in the entire experiment. Interestingly, adding further sensors to the 3U morphology actually harms performance, as can be seen by

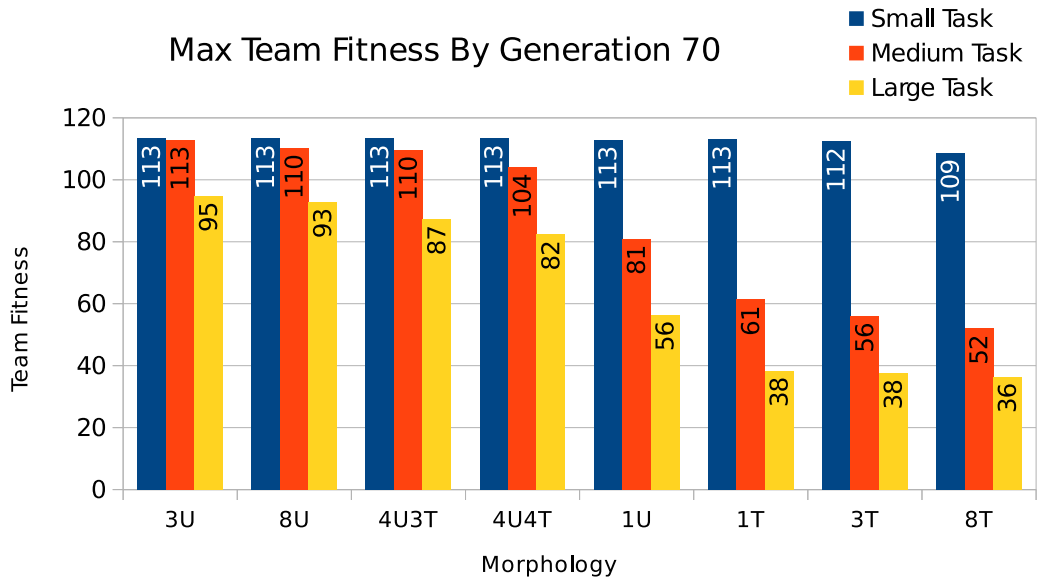


Figure 11: The maximum attained team fitness by generation 70, per morphology, averaged over 10 runs

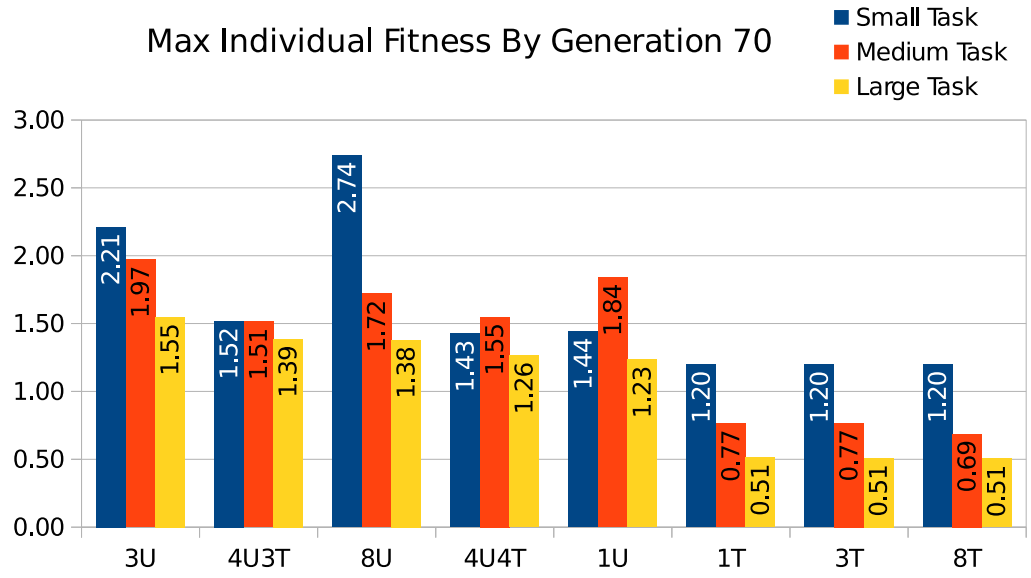


Figure 12: The maximum attained individual fitness by generation 70, per morphology, averaged over 10 runs

4U3T, 4U4T, and 8U performance results. This can be explained by the additional sensor inputs causing an expansion of the state space, due to the increase in the number of variable terminals that can be used in a GP candidate.
The typed sensors proved to be largely ineffective. As shown by the performance of 1T, 3T and 8T, the three worst performers in the experiment, these sensors alone are not capable of adequate task performance; and as shown by 4U3T and 4U4T, their addition to morphologies that perform well already, namely 3T, does not improve performance. In fact, it seems to decrease performance, likely due to the state space expansion discussed earlier.
The cause for the typed sensor's underperformance is likely to be its comparatively low range and field of vision, when contrasted with the ultrasonic sensor (see table 2). A simple arc area calculation shows the area covered by the typed sensor:

$$\begin{aligned} A &= \frac{1}{2} r^2 \theta \\ A &= \frac{1}{2}(1)^2 \frac{1}{12}\pi \\ &= \frac{1}{24}\pi \\ &\approx 0.13 \end{aligned}$$

Contrasted to the ultrasonic sensor:

$$\begin{aligned} A &= \frac{1}{2} r^2 \theta \\ A &= \frac{1}{2}(4)^2 \frac{7}{18}\pi \\ &= \frac{28}{9}\pi \\ &\approx 9.77 \end{aligned}$$

These two area coverage values are approximately a factor of 75 apart from one another; this is a strong indicator that the likely reason the typed sensor has been underperforming, is its lack of coverage when compared to the wider and more far-reaching ultrasonic sensor.

5.2 Conclusions and Future Work

This project has revealed that the requirements for evolving effective cooperative behaviour are modest; a robot with just 3 sensors [6] is capable of good to excellent performance across a range of foraging task setups.
An additional discovery of the experimental results is the poor performance of the typed sensor. A possible area of future work is properly examining how the performance of such type-aware sensors varies with its parameters such as range and field of vision. A

[6] We disregard the pickup and target area sensor in this count, as these do not aid cooperation, and were not varied in the experiments.

Figure 13: The 3U morphology proved adequate for the three foraging tasks presented in this report

more powerful version of such a sensor could have the potential to match, or out-perform the simpler proximity sensors that have been dominant in this report's experiments.

References

Arkin, Ronald C and Tucker Balch (1998). "Cooperative multiagent robotic systems". In: *Artificial Intelligence and Mobile Robots. MIT/AAAI Press, Cambridge, MA*.

Asada, Minoru and Hiroaki Kitano (1999). *RoboCup-98: Robot Soccer World Cup II*. Vol. 1604. Springer.

Balch, Tucker (1999). "Reward and diversity in multirobot foraging". In: *IJCAI-99 Workshop on Agents Learning About, From and With other Agents*.

Bryant, Bobby D and Risto Miikkulainen (2003). "Neuroevolution for adaptive teams". In: *Proceedings of the 2003 congress on evolutionary computation*. Vol. 3, pp. 2194–2201.

Busoniu, Lucian, Robert Babuska, and Bart De Schutter (2008). "A comprehensive survey of multiagent reinforcement learning". In: *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on* 38.2, pp. 156–172.

Castle, Tom (2012). *EpochX: genetic programming for research*. URL: `http://www.epochx.org`.

Doran, Jim E et al. (1997). "On cooperation in multi-agent systems". In: *The Knowledge Engineering Review* 12.03.

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

Floreano, Dario, Francesco Mondada, et al. (1994). "Automatic creation of an autonomous agent: Genetic evolution of a neural-network driven robot". In: *From animals to animats* 3.

Haynes, Thomas et al. (1995). "Strongly Typed Genetic Programming in Evolving Cooperation Strategies." In: *ICGA*. Citeseer, pp. 271–278.

Hunt, K Jetal et al. (1992). "Neural networks for control systems—a survey". In: *Automatica* 28.6, pp. 1083–1112.

JBox2D: A Java Physics Engine (2014). URL: `http://www.jbox2d.org/`.

Khepera III (2014). URL: http://www.k-team.com/mobile-robotics-products/khepera-iii.

Koza, John R (1990). "Genetically breeding populations of computer programs to solve problems in artificial intelligence". In: *Tools for Artificial Intelligence, 1990., Proceedings of the 2nd International IEEE Conference on*. IEEE, pp. 819–827.

Koza, John R. (1992). *Genetic Programming: on the Programming of Computers by Means of Natural Selection*. MIT Press. ISBN: 9780262111706.

Luke, Sean, Charles Hohn, et al. (1998). "Co-evolving soccer softbot team coordination with genetic programming". In: *RoboCup-97: Robot soccer world cup I*. Springer, pp. 398–411.

Luke, Sean and Lee Spector (1996). "Evolving teamwork and coordination with genetic programming". In: *Proceedings of the first annual conference on genetic programming*. MIT Press, pp. 150–156.

MASON Multiagent Simulation Toolkit (2014). URL: http://cs.gmu.edu/~eclab/projects/mason/.

Matarić, Maja and Dave Cliff (1996). "Challenges in evolving controllers for physical robots". In: *Robotics and autonomous systems* 19.1, pp. 67–83.

Miglino, Orazio, Henrik Hautop Lund, and Stefano Nolfi (1995). "Evolving mobile robots in simulated and real environments". In: *Artificial life* 2.4, pp. 417–434.

Nolfi, Stefano and Dario Floreano (2001). *Evolutionary robotics. The biology, intelligence, and technology of self-organizing machines*. Tech. rep. MIT press.

Panait, Liviu and Sean Luke (2005). "Cooperative multi-agent learning: The state of the art". In: *Autonomous Agents and Multi-Agent Systems* 11.3.

Piszcz, Alan and Terence Soule (2006). "Genetic programming: Optimal population sizes for varying complexity problems". In: *Proceedings of the 8th annual conference on Genetic and evolutionary computation*. ACM, pp. 953–954.

Poli, Riccardo and William B Langdon (1998). "Schema theory for genetic programming with one-point crossover and point mutation". In: *Evolutionary Computation* 6.3, pp. 231–252.

Poli, Riccardo, Nicholas Freitag McPhee, and Leonardo Vanneschi (2008). "Elitism reduces bloat in genetic programming". In: *Proceedings of the 10th annual conference on Genetic and evolutionary computation*. ACM, pp. 1343–1344.

Reeve, H. Kern and Bert Hölldobler (2007). "The emergence of a superorganism through intergroup competition". In: 104.23.

Stone, Peter and Manuela Veloso (2000). "Multiagent systems: A survey from a machine learning perspective". In: *Autonomous Robots* 8.3, pp. 345–383.

Sutton, Richard S (1988). "Learning to predict by the methods of temporal differences". In: *Machine learning* 3.1, pp. 9–44.

Tuyls, Karl, Katja Verbeeck, and Tom Lenaerts (2003). "A selection-mutation model for q-learning in multi-agent systems". In: *Proceedings of the second international joint conference on Autonomous agents and multiagent systems*. ACM, pp. 693–700.

Waibel, Markus, Laurent Keller, and Dario Floreano (2009). "Genetic Team Composition and Level of Selectionin the Evolution of Cooperation". In: 13.3.

Wolpert, David H and Kagan Tumer (2001). "Optimal payoff functions for members of collectives". In: *Advances in Complex Systems* 4.02n03, pp. 265–279.