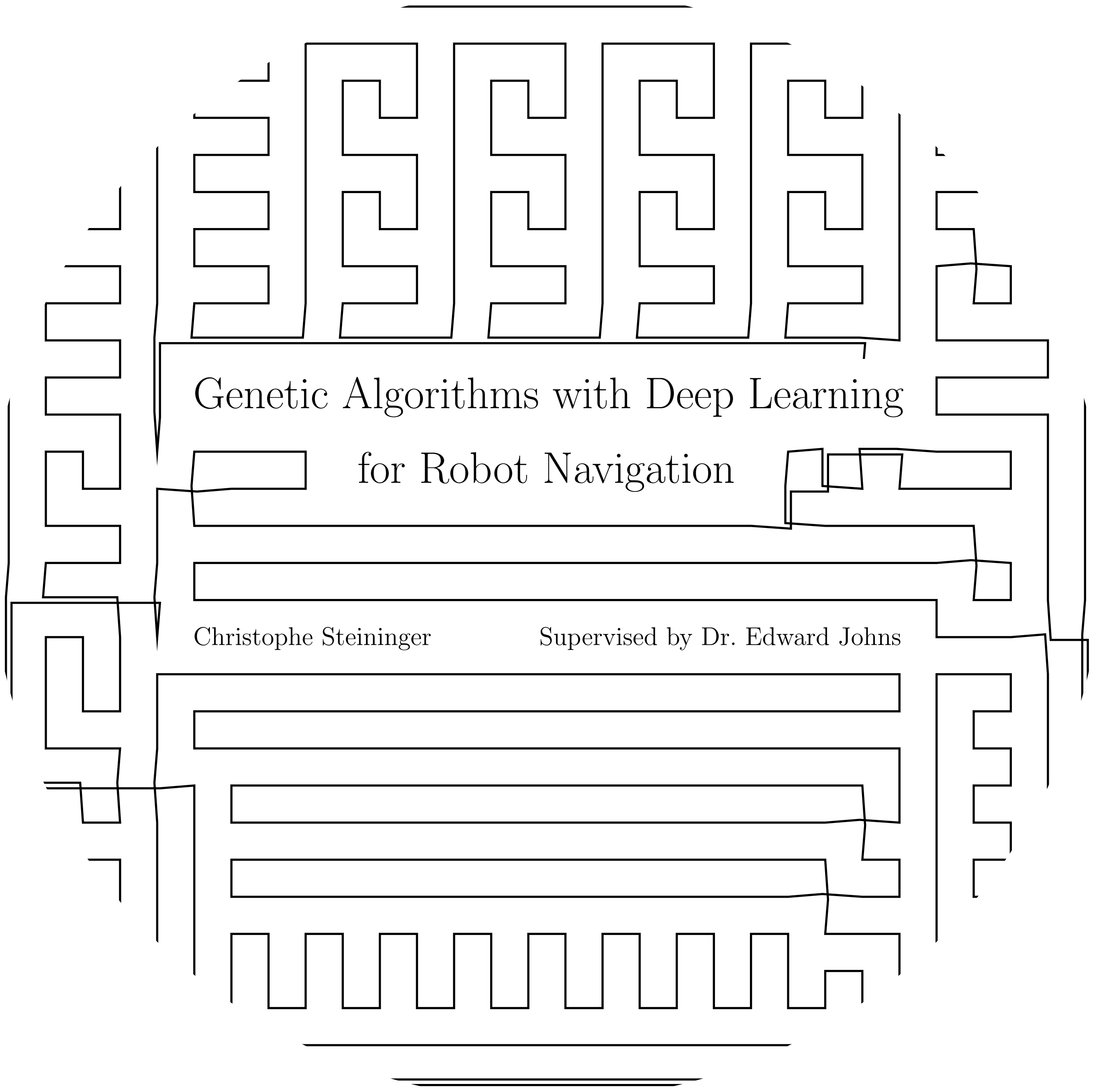


Genetic Algorithms with Deep Learning for Robot Navigation

Christophe Steininger

Supervised by Dr. Edward Johns

June 2016

I would like to thank my supervisor Ed for his guidance and insight, and especially for his advice for the future, and my family for always supporting me.

Abstract

Recently deep learning has been successfully shown to solve very complex problems, however this has largely been limited to supervised learning problems such as classification and regression. We want to investigate a more open ended problem where labelled training data is not available but the quality of the solution is. This thesis aims to address one solution where genetic algorithms are used to train a neural network. The example we use is a mobile vacuum cleaner. The network must learn to clean the entire room without bumping into obstacles. This requires a lot of training so we simulate the room and robots to focus on improving the training method. We show how the basic genetic algorithm and neural network system can be improved from poor performance a single small room to much higher performance in thousands of larger and unknown rooms. Our final method is able to quickly train robots which can consistently perform well in small rooms and large empty rooms with limited information, however the performance degrades in large rooms with many obstacles.

Contents

Chapter 1

Introduction

Deep learning has revolutionised the use of neural networks, in 2011 a deep learning system became the first visual pattern recognition system to achieve superhuman performance (half the error rate of humans) [1, 2] and state of the art solutions to other difficult machine learning challenges such as image captioning [3] and speech recognition [4] also use deep learning systems. In these problems there is a correct output for each input (the caption of an image or the transcript of a sound clip) and large amount of hand labelled training data is required. In real world problems, trianing data is often limited and the value of an output is not immediately known. An example of this is chess where the player is rewarded or punished only at the end of the game. Despite these challenges deep learning has recently been applied to problems such as playing video games using raw pixel values as input [5].

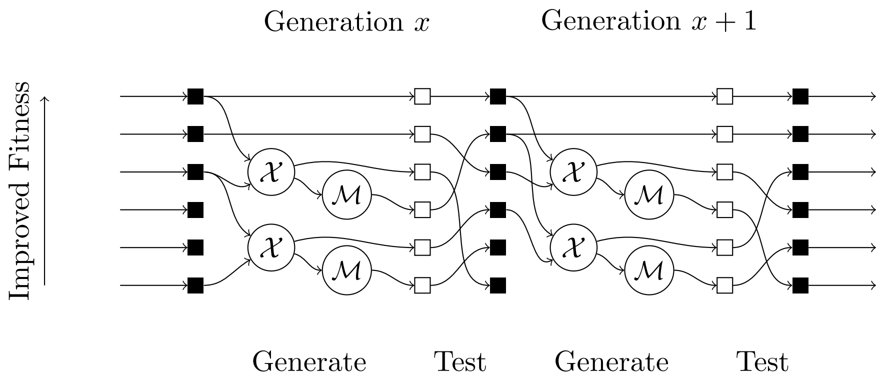


Figure 1.1: An example of the generate and test loop of a genetic algorithm. Six individuals for each generation (□) are created with two genetic operators (crossbreeding, $\mathcal{X}$ and mutation, $\mathcal{M}$) then assigned a fitness and sorted (■). The best two individuals are copied into each new generation without modification to preserve good solutions. Here each individual is a neural network.

In this thesis we will investigate the effectiveness of an alternative approach, namely training a neural network with a genetic algorithm. A genetic algorithm is a search heuristic which can be easily applied to a wide range of optimisation problems as the only requirements are a fitness function, which given a candidate solution outputs a number used to compare to other candidate solutions and a method to encode solutions (e.g. as a list of numbers or

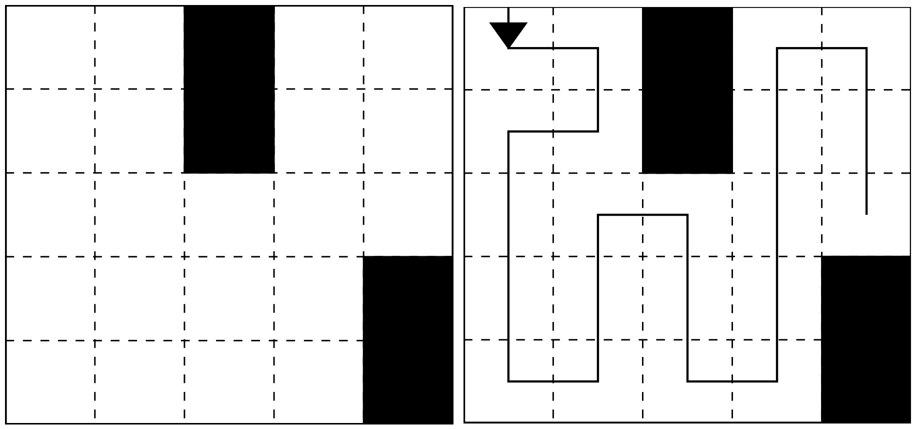


Figure 1.2: The static 5×5 room used for training robots during the first phase (left) and a possible path through the room (right). The black squares are obstacles and the robot must cover every white square beginning from the top left cell.

nodes in a tree). Specifically, the optimisation problem may be highly non-linear, stochastic or non-differentiable and therefore ill suited for standard optimisation algorithms [6]. A genetic algorithm combines pairs of high fitness candidate solutions of one generation using genetic operators to create the next generation, a technique inspired by natural selection. An example of this process is shown in Figure 1.1.

A wide range of real world problems in areas such as robot control and finance can be solved with a genetic algorithm and neural network so this system has broad applications. Here, we will consider one such application, controlling a robot vacuum cleaner in a simulated environment. Robot vacuum cleaners are one the first examples of a widely used domestic robot which can perform a common and complex chore. However all robot vacuum cleaners sold today are controlled by hand coded rules in a wide range of sophistication, from picking a random direction in the hope of cleaning new areas to navigating using a map built from spinning sensor mounted on the robot.

In order to focus on improving the training we will simulate the rooms during the project as a grid containing several obstacles. The robot will be controlled by a trained neural network which must reach every empty cell in the room while avoiding obstacles by using information about the room and the state of the robot to chose appropriate actions. The rooms used will become more difficult as the genetic algorithm is able to find better neural networks. The complexity of the rooms split the project into two phases:

1. Initially the robots will the tasked to cover all empty cells of a small static room such as the room in Figure 1.2.

2. Next, robots will be trained in different static rooms. We will investigate how well the robot has learnt to generalise its path finding behaviour by testing the robot in rooms not present during training.

1.1 Contributions

- We give an overview of the implementation of the simulation, genetic algorithm and neural network system (Chapter 3) and highlight several key optimisations.
- We show how the basic training system which cleans on average 46.6% of the room shown in Figure 1.2 after 100 generations of training can be improved to clean 98.6% of the room. The probability of a single instance of the genetic algorithm finding a solution with complete coverage has increased from 0% after 1,000 generations to 77% after 50 generations (Chapter 4).
- We extend the training method to handle multiple rooms (Chapter 5). The robot is trained on several rooms then tested in a different set of rooms. We show how we improved the performance of the system built in the previous chapter from 10% to 79.7% in thousands of 8×8 testing rooms not present during training.
- We compare robots trained with our solution to a variety of other methods including a heuristic travelling salesman solver to approximate the optimum path. We find that our solution gives reasonable performance in small or mostly empty rooms. However, in large rooms with many obstacles our solution performs poorly and is outperformed by other methods (Chapter 6).

1.2 Other Approaches

We can find optimal paths by converting a room to an instance of a travelling salesman problem (TSP) and using an existing TSP solver. However the purpose of this application is only to test the suitability of applying genetic algorithms to neural networks and not to improve upon the TSP solution.

A robot vacuum cleaner currently on the market is the iRobot Roomba which picks a random direction or performs wall following when reaching an obstacle (Figure 1.3) to clean a room. This is effective because obstacles can be introduced or removed without affecting the performance of the robot and this behaviour is very simple to implement. However the robot may clean some areas much more thoroughly than others and spends much more time than a human. Other robots such as Neato and Evolution Robots Mint use more elaborate sensors to clean the room uniformly in a pattern similar to a human.

The Mint is shipped with a separate device which projects an infrared image onto the ceiling; the robot uses a infrared sensor pointed upward to read the image and determine its location. The Mint detects obstacles with a forward facing proximity sensor and builds a map of the room to find a good cleaning path similar to the Neato.

The Dyson 360 Eye uses a vision sensor mounted on top of the robot to scan the room, the sensor output allows the robot to build a map of the room which is used to approximate the optimal cleaning path. Unlike the Roomba, the Neato does not bump into obstacles and can continue cleaning from the same location after automatically returning to the dock to recharge its batteries.

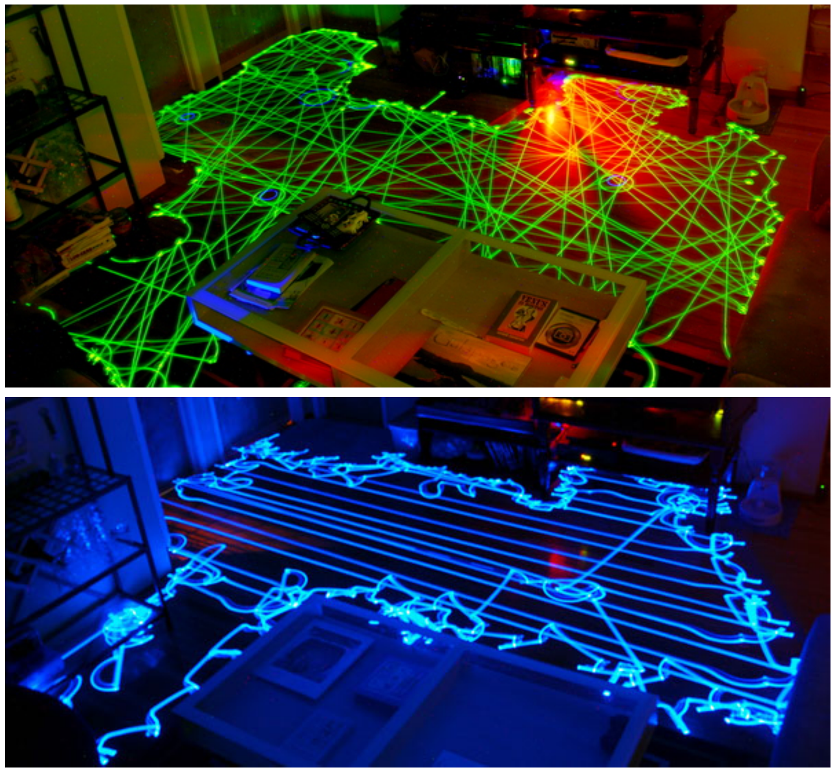


Figure 1.3: Long exposure shots of a light mounted on a Roomba (top) and a Mint (bottom). The Roomba spent 45 minutes cleaning the room while the Mint spent 15 minutes.

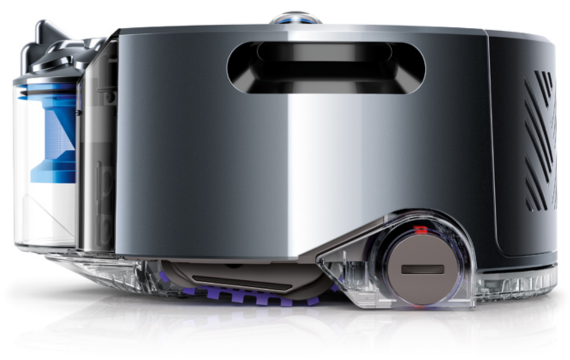


Figure 1.4: The Dyson 360 Eye. The vision sensor is visible at the top of the robot, a mirror which reflects the room into an upward facing camera.

Chapter 2

Background

2.1 Travelling Salesman Problem

The solution to a travelling salesman problem (TSP) is the shortest path which starts and finishes at given city and visits every other city in the problem exactly once. The problem can be formally represented by a complete weighted graph where each vertex corresponds to a city and edge weights are some measure of the cost of travelling between cities, such as time, cost or distance.

Our robot navigation problem can be presented as a TSP instance by creating a complete graph with a vertex for each empty cell in the room. The weight of each edge in the graph is set to the number of cells traversed by the robot in the shortest path between the vertices. For example, the weight of an edge between any orthogonally adjacent vertices is set to one. We set the weight of all edges leading to the start node to zero to search for a path instead of a cycle, this creates an asymmetric TSP. We discuss this conversion in more detail in Section 6.2. The solution to the TSP is the optimum order in which to visit each cell.

An exact solution to a TSP can be found with a dynamic programming algorithm in $O(n^2 2^n)$ time [7]. However, effective heuristics such as the Lin-Kernighan heuristic [8] are known which can solve an asymmetric TSP with several hundred vertices within several seconds on modern desktop [9] and a run time complexity of $O(n^{2.2})$ [10].

2.1.1 Evaluation

An exact solution to asymmetric TSP problems of our size is not tractable, we need a solution which will work for large 15×15 rooms. Heuristic solutions produce optimum or near optimum solutions far faster. Since we are not required to produce the best solution for a room, a heuristic algorithm is a better choice than an exact solver.

A heuristic algorithm will find a solution close to the optimal which we do not attempt to match, instead we use the solutions to evaluate the quality of solutions found by our training method. However, the heuristic TSP approach does have disadvantages, namely we must know the entire room and the room can not change during operation. A neural network is more flexible and can learn to approximate the optimal decision from incomplete information.

2.2 Artificial Neural Networks

2.2.1 Artificial neurons

An artificial neuron is the basic unit of neural networks. Each neuron takes one or more inputs and produces a single output. Each input to a neuron is given a synaptic weight which is multiplied by the input value. An activation function is applied to the sum of the weighted inputs to give the output of the neuron which may be used by other neurons in the network.

The activation function can ensure that the output of the neuron is kept within a certain range. A common choice for this task is the sigmoid function (Equation 2.1) [11]. Recently, the rectifier function has also become widely used (Equation 2.2) [12].

$$S(x) = \frac{1}{1+e^{-x}} \tag{2.1}$$

$$f(x) = \max(x, 0) \tag{2.2}$$

2.2.2 Feed forward networks

An artificial neural network is created by connecting the inputs and outputs of neurons together. The directed graph formed by these neurons and connections is known as the network topology or architecture and is an important design consideration.

A feed forward network is a neural network organized into two or more layers of neurons (Figure 2.1). Each neuron in a layer is an input to every neuron in the next layer. There are no other connections. The first layer consists of input neurons: special neurons which copy an input of the network directly onto their outputs. Input neurons have no synaptic weights or activation function. Neurons in the last layer are known as output neurons and the output of these neurons form the output of the network. These neurons usually have no activation function. The remaining layers and neurons are referred to as the hidden layers and hidden neurons.

While the number of input and output neurons is usually fixed by the problem, the number and size of each hidden layer can vary. Increasing the number of hidden layers and neurons increases the complexity of the task which the network can perform as each layer can extract more complex patterns from the previous layer. A neural network is trained to solve a given problem usually by fixing the topology and searching through the space of all possible synaptic weights [11].

The output of a layer with m neurons and n inputs in a feed forward network in matrix notation is:

$$\mathbf{y} = \varphi(\mathbf{W}\mathbf{x}) \tag{2.3}$$

where $\mathbf{W}$ is $m \times n$ matrix containing the synaptic weights, $\mathbf{x}$ is a column matrix containing the n inputs, $\mathbf{y}$ is the m outputs of the layer and φ is the activation function of the layer applied to each element in a column matrix.

2.3 Simulating Neural Networks

This project requires simulation of many neural networks so it is important to reduce the time spent running a neural network.

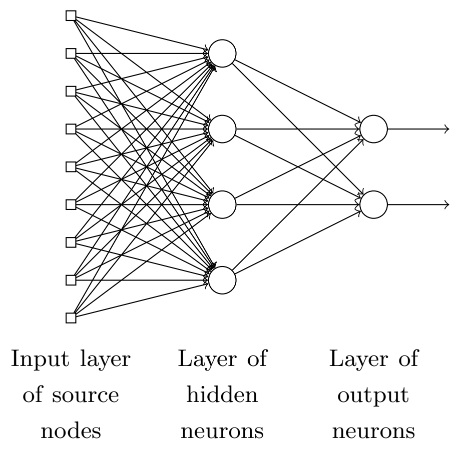


Figure 2.1: A feed forward network with nine inputs, two outputs and a single hidden layer of four neurons.

2.3.1 TensorFlow

TensorFlow is an open source machine learning library by Google released in November 2015. The user defines a graph of operations which may include various types of neural networks, trainers, arithmetic operations and file readers. The graph can be executed any number of times with different inputs. A key feature of TensorFlow is that a graph will run on GPUs and CPUs with no code change.

2.4 Genetic Algorithms

A genetic algorithm is a search heuristic introduced by John Holland in the 1970's inspired by natural selection [13]. Each candidate solution is encoded as a chromosome which is sequence of genes. Depending on the problem a single gene may be a binary bit, a number or an action which an agent can take. The algorithm maintains a population of chromosomes which is evolved by first evaluating the performance of each chromosome then combining and mutating the best performing individuals to create the next generation of chromosomes. This process is repeated for a set time or until the desired fitness is achieved. See Figure 2.2 for pseudocode.

2.4.1 Fitness function

This function evaluates a chromosome by applying the solution encoded in the chromosome to the problem and returning a number which indicates the quality of the solution. The fitness is usually between 0 and 1, and higher numbers usually mean better solutions. For example, if we use a genetic algorithm to design a race car, the fitness function will build a car from the given chromosome then simulate the car racing around a track. The time to complete the course is the fitness of the chromosome.

```
Randomly generate the initial population.
repeat
  Call the fitness function on each chromosome in the population.
  Add fittest chromosomes to the new population (elitism).
  repeat
    Choose two parents using the fitnesses and a selection function.
    Create two children from the parents using a crossover operator.
    Apply the mutation operator to each child.
    Add the children to the new population.
  until new population size equals current population size
until termination condition holds
```

Figure 2.2: Pseudocode of a genetic algorithm.

2.4.2 Elitism

The chromosomes with the highest fitness of a generation are known as elites. These chromosomes are copied without modification to the next generation, ensuring that the best solutions are never destroyed. The number of elites copied is a hyperparameter to the genetic algorithm.

2.4.3 Selection functions

These functions select one parent from the population using the results of the fitness function. Chromosomes with a high fitness are more likely to be selected. Selected chromosomes are paired up and bred in an attempt to create children with higher fitness.

Roulette selection

This function can be imagined as assigning each chromosome a number of pockets on a roulette wheel based on its fitness. The wheel is spun to choose a chromosome. More formally:

1. Create a list of normalized fitnesses by dividing each fitness by the total fitness.
2. Add all previous fitnesses to each fitness in the normalized list. This creates a list of cumulative fitnesses.
3. Chose a random number between 0 and 1.
4. Return the chromosome corresponding to the first element in the cumulative fitness list which is greater than the random number.

Tournament selection

Tournament selection requires two additional hyperparameters: k, the size of each tournament and p, a probability.

1. Create a set of k randomly chosen chromosomes from the population.
2. Sort the set by fitness.

3. The probability of choosing chromosome $i = 0 \dots k-1$ in the sorted set is $p(1-p)^i$. Chromosome 0 is the chromosome with the highest fitness of the set.

2.4.4 Crossover operators

A crossover operator combines two parent chromosomes to create a new child chromosome for the next generation. Each gene in the child's chromosome is a copy of the corresponding gene of one parent. Crossover operators differ in how the parent for each gene is selected. A second child can be created by choosing the opposite parent for each gene, regardless of the crossover function used.

Typically, genetic algorithms include a crossover rate parameter (the probability that a crossover is performed on two parents) otherwise both children are copies of the parents.

Single point crossover

One point in the parents' chromosomes is randomly chosen. All genes before this point are copied from the first parent to the child. The remaining genes come from the second parent.

Two point crossover

All genes between two randomly chosen points are copied from the first parent. Genes which are not between these two points come from the second parent.

Uniform crossover

In uniform crossover, each gene in the child chromosome is copied randomly with equal probability from either parent.

2.4.5 Mutation operator

The mutation operator will randomly modify genes in the chromosome. The probability of a gene mutating is the mutation rate. The mutation of a gene depends on the encoding; if a gene is a binary bit then mutating this gene will flip the bit and if a gene is a number then mutation will add a small random number. This preserves diversity in the population and helps to prevent early convergence.

2.4.6 Evaluation

Genetic algorithms can be easily applied to a wide variety of problems as only the encoding and fitness function are problem specific. The selection functions and genetic operators are general and can be applied to any problem which is encoded as a string of values. Different genetic operators must be used if genes are more complex such as a nodes of a tree or arbitrary objects.

However, all genetic operators and most functions described above introduce a new hyperparameter in addition to population size and the termination condition, resulting in a large number of hyperparameters which must be set. Typical ranges for each hyperparameter have been determined empirically (e.g. less than 1% for the mutation rate) but optimum values must be found for each new problem.

Genetic algorithms generate many chromosomes each of which must be evaluated using the fitness function. This is computationally expensive in some applications e.g. designing an engine or protein folding require complex simulations and decisions. Furthermore, designing the fitness function is often challenging as a chromosome's performance may be difficult to quantify (e.g. creative tasks, generating music and art) or to express as a single number.

2.5 Neuroevolution

Neuroevolution is the process of using an evolutionary algorithm to train an artificial neural network. The fitness of a genotype (a chromosome) is found by mapping to a neural network (the phenotype) which is used to control an agent, and measuring the performance. Neuroevolution is effective in areas such as reinforcement learning and previous studies have shown neuroevolution to outperform backpropagation at classification problems [14, 15].

2.5.1 Conventional neuroevolution

The most direct form of neuroevolution, the chromosome is simply the list of all weights in the network in a predetermined order. A genetic algorithm as described above is used to find a good chromosome.

This approach is simple to implement but does not evolve the network topology, so the size of each hidden layer must be found first using another method. However, this is not a significant disadvantage for the project since the networks are small enough that we can try all combinations of layer sizes.

2.5.2 Co-evolution

More sophisticated methods maintain multiple populations, or species of chromosomes (called subgenotypes in neuroevolution). Each subgenotype of a species maps to a small part of the network [16, 17]. Species are evolved separately but evaluation uses a member from all species. This approach breaks down the complex problem of finding all synaptic weights for a network into smaller problems such as finding the weights for a single neuron.

Co-evolution encourages species to find different evolutionary niches i.e. all species are encouraged to perform a different useful task. This causes subgenotypes in a co-evolution algorithms to be more similar than chromosomes in a genetic algorithm. High similarity in a population reduces the chance of a poor crossover caused by selecting two parents which both achieve a high fitness in far apart areas of the search space (Figure 2.3) [17].

Co-evolution algorithm

An important implementation detail of the algorithm is the granularity of a species. At the finest level each subgenotype is a single number corresponding to a weight in the network, but a subgenotype could also represent all weights of a layer in a feed forward network. We evaluate a subgenotype by choosing one representative subgenotype from all other species to create a full geneotype from which the network to control the agent can be built (Figure 2.4). Representatives are usually the best performing individuals of each species or are chosen with a probability increasing with fitness. Each subgenotype is typically evaluated in several trials

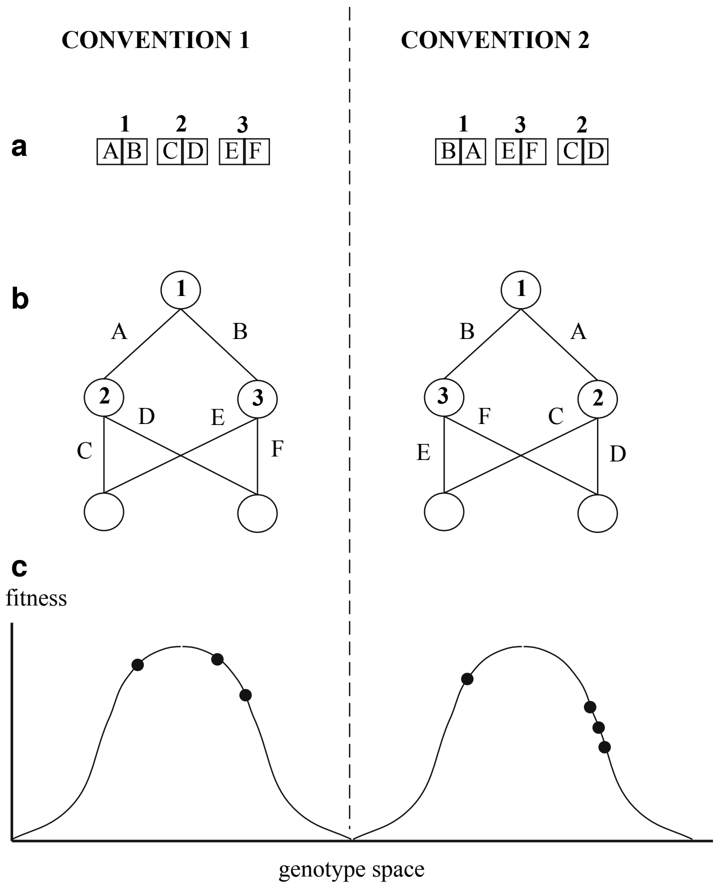


Figure 2.3: Six high fitness genotypes in two different conventions. Crossover between individuals of different conventions is unlikely to produce viable genotypes. [18].

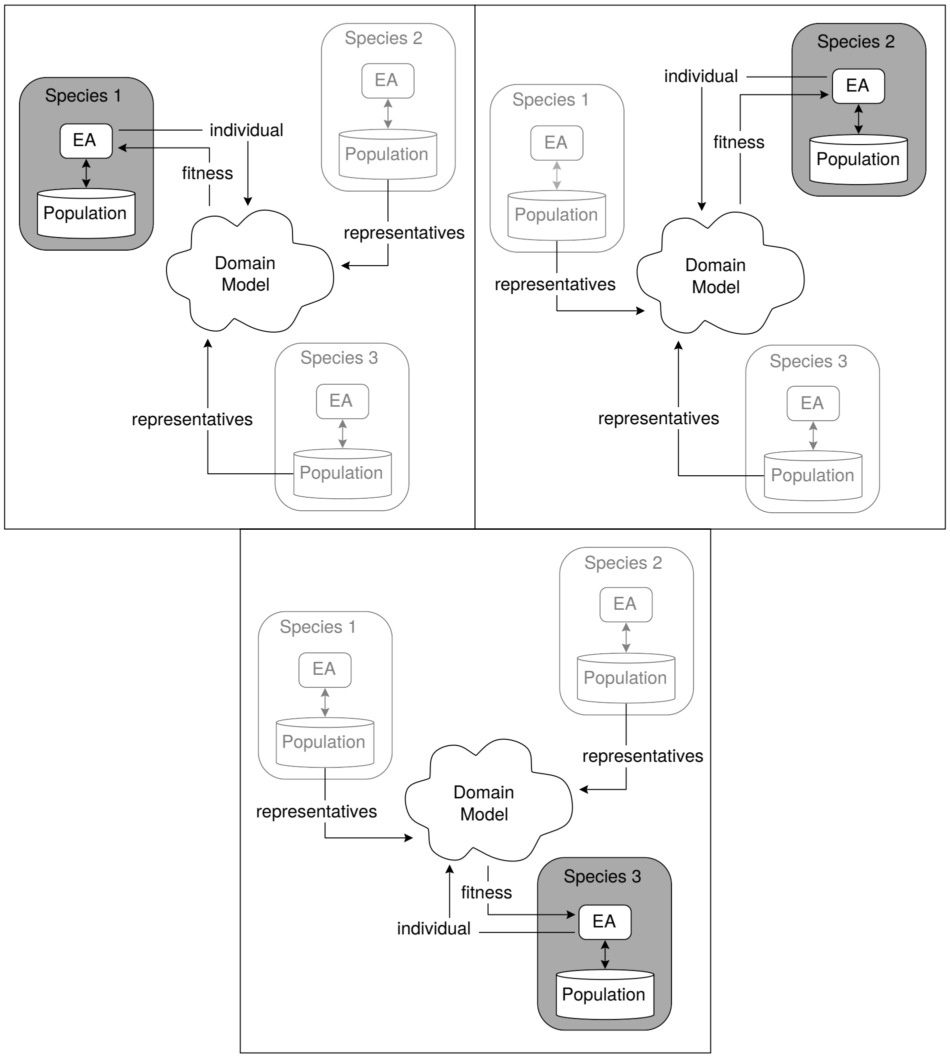


Figure 2.4: Coevolution with three species [17]

using different representatives in order to estimate the contribution of the individual instead of the fitness of the combined genotype. This can be done by taking the maximum or average fitness over all trials. Once evaluation is complete, a new generation is created for each species in isolation as in a normal genetic algorithm, i.e. there is no interbreeding between species.

Co-evolution has been shown to give significantly better results than classical reinforcement and neuroevolution solutions [16].

2.5.3 Evolving topology with synaptic weights

Network topology can have a significant effect on the quality of the network, but choosing the right architecture for a problem is difficult; instead, some methods begin with a simple network and gradually increase the complexity of the network while evolving weights for the network.

For simple tasks, the right topology can be found with an exhaustive search so the extra implementation effort is likely to result in faster search times but not better results.

2.6 Reinforcement Learning

Reinforcement learning is an area of machine learning where an agent (such as our robot vacuum cleaner) interacts with an environment to maximise a reward signal, without an explicit teacher as in supervised learning. The agent must learn how its actions affect the environment and its own decision making policy to maximise the reward [19].

Reinforcement learning algorithms estimate the value of a given state or the value of taking an action at a given state. After training, the agent will greedily choose actions according the value function to maximise the total reward. There are many methods for constructing these estimates such as propagating the final reward of a trace backward along all previous states, or updating the value of the current state on each state transition.

This is very different to a genetic algorithms training a neural network, a genetic algorithm searches for a set of parameters which solve the problem instead of searching through the states of the problem. In the context of our navigation problem, the result of reinforcement learning is a policy which returns an direction to take given the current state, this policy is far easier to understand than the synaptic weights returned by the genetic algorithm. The genetic algorithm is therefore difficult to tune during execution as the function of layers and neurons is difficult to detect.

Chapter 3

Implementation

In this chapter we give an overview of our training method and some of the implementation challenges faced during this project.

3.1 Training Method

Before training begins we decide on the network topology used to control the robot (i.e. the number of input and output neurons and the number and size of each hidden layer). Each synaptic weight in the network is a parameter to be found by the genetic algorithm. The size of the genotype is therefore equal to the number of synapses in the chosen topology.

Next, we begin the generate and test loop of the genetic algorithm to find the weights of the network. In each iteration, the networks are created from the population of genotypes by inserting each gene into the network topology. The fitness of a genotype is the fraction of empty cells reached by the robot. The next generation of genotypes is created using crossover and mutation of the high fitness individuals. This continues until the room is solved (all empty cells have been reached) or the maximum number of generations, usually 100, have been created.

3.1.1 Simulation Method

Throughout the project we must determine how effective a given robot is in a given room. This occurs inside the fitness function of the genetic algorithm and when we evaluate how well a robot can generalise to new rooms. We call our measure of a robot's effectiveness in a room the *coverage*, this is the proportion of the empty cells in the room which were visited at least once by the robot. Coverage is calculated by running the following simulation:

1. Place the robot in the top left corner of the room.

2. Assign a direction (north, east, south or west) to each of the four output neurons in the network.

3. Create a set of previous positions, initially empty.

4. Go to (5) if the termination condition (discussed below) is met, otherwise:

   4.1. Add the robot's position to the previous positions set if this is the first visit to this cell.

   4.2. Create a vector of inputs using the location of obstacles in the given room, the position of the robot and previous position set.

   4.3. Calculate the output of the given network using the input vector.

   4.4. Find the output neuron with the highest value. The direction assigned to this neuron in (2) is the next direction.

   4.5. Update the robot's position.

   4.6. Go back to (4).

5. Return the fraction of empty cells visited at least once by the robot.

3.1.2 Simulation termination condition

The simulation is ended if (1) all cells have been visited, (2) the robot ran into an obstacle or into the edge of the room or (3) the robot is caught in a loop. The first two conditions are straightforward but (3) is more complex. An example of a loop is shown in Figure 3.1 where the robot has reached the top left cell for the third time. The inputs to the network on the third visit to this cell are identical to the second visit as the previous position set has not changed, therefore the robot will make the same choice and go east. In the next step, the inputs are again identical to the inputs during the second visit to the top middle cell and the robot will move to the top right cell. This continues causing the robot to travel around the edge of the room forever and crucially no new cells will ever be covered. Therefore, the simulation should end when the robot reaches the top left cell for the third time as the result of the simulation will not change.

The obvious solution, which we initially used, is to set a maximum number of steps which is much larger than the number of empty cells in the room and only check conditions (1) and (2). Any robot stuck in a loop will use the maximum number of steps but the simulation will still return the correct coverage. This is simple to implement but inefficient. Furthermore, choosing a maximum number of steps may led to incorrect results when a simulation is terminated before achieving the true coverage. To ensure that this does not happen we set the maximum number of steps to ten times the number of empty cells. This is much higher than necessary but we did not want to risk any incorrect results.

A better solution, which we now use, is to trigger condition (3) when the robot reaches a previously visited cell without visiting any new cells in the steps in-between. This detects a cycle before the robot begins the loop and removes the need for the maximum step limit. Verifying that this produces the correct result is simple, we ran both versions with the same random seed and saved and all coverage results over several trials. We found no differences in the results. The new solution is on average 12.2 times faster without any doubt about early simulation terminations. Even with a better estimate of the maximum number of cells this new solution will outperform the maximum steps method as a simulation is ended as soon as the robot is locked in a loop.

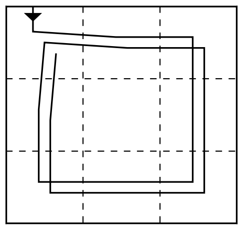


Figure 3.1: Path of robot which is now stuck in a loop.

3.1.3 Input vector

At each step in the simulation, the inputs to the network are formed by creating a vector with three elements for each cell in the room. These elements are set as follows:

- Room input: set to 1 if the cell contains an obstacle, 0 otherwise.
- Position input: set to 1 if the robot is in the cell, 0 otherwise.
- History input: set to 1 if the robot has already visited the cell, 0 otherwise.

In some experiments we ignored one of these inputs and used two input neurons per cell.

3.2 Room Generation

For training and testing the robots created in Chapter 5 we need tens of thousands of unique rooms of a given size. We cannot design each room manually due to the quantity required. Instead we generate this room set randomly before beginning the training with the genetic algorithm. We generate a room by creating a grid with the given width and height then randomly placing obstacles on the grid, if this grid is a valid room we add this to the room set otherwise we throw away the room. We repeat until the room set has the required size. A grid is a valid room if all of the following conditions are met:

1. The top left cell (the starting cell) is empty.
2. All empty cells can be reached from the top left cell.
3. The room is different from all previously generated rooms.

We check the second condition by running a depth first search through empty cells beginning at the top left cell. If the number of empty cells reached by this search is equal to the total number of empty cells in the grid, the condition is met.

Examples of rooms of various sizes generated with this method are shown in Figure 3.2.

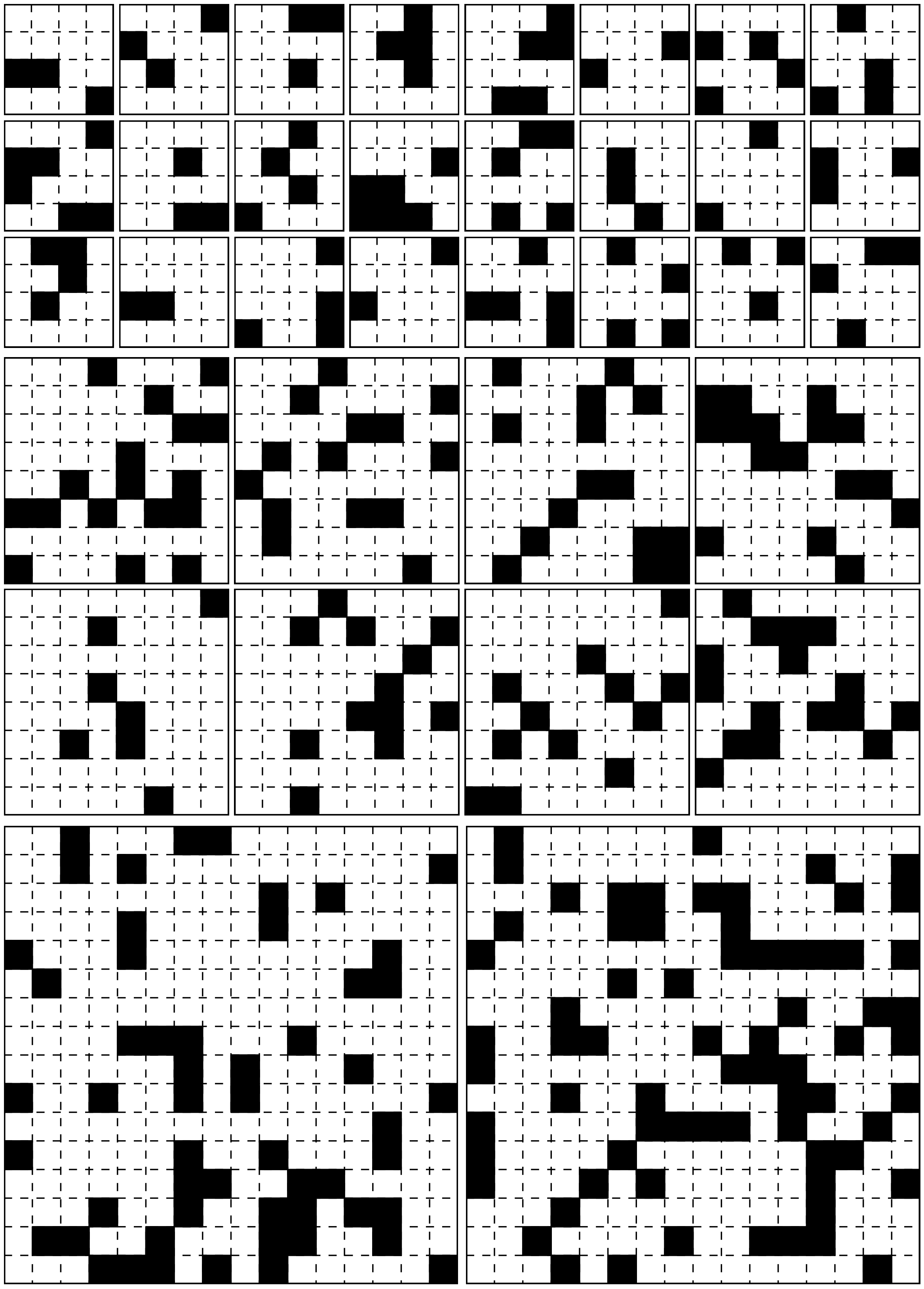


Figure 3.2: Examples of rooms created with our generator. We used several different room sizes during the project. Black cells represent obstacles and the robot must pass through all white cells.

3.2.1 Complete room set generation

When using small room sizes such as 4×4, we can use the set of all valid 4×4 rooms as this is a relatively small set (5,293 rooms) and simulating a robot in each of these rooms can be done quickly and in parallel. To generate the set of all rooms of a given size (a complete room set) the method described above is inefficient as we rely on chance to find new rooms. For example, if we have generated 5,290 out of 5,293 valid 4×4 rooms, each new room we generate has a 0.057% chance of being unique.

We initially solved this problem by finding the power set of the set of all cells in the room. In a 1×2 room with the two cells indexed by 1 and 2 the power set is $\{\emptyset, \{1\}, \{2\}, \{1, 2\}\}$. For each set S in this power set we create a candidate room by placing obstacles on the cells in S then check if this room is valid. This reduced the complete 4×4 room generation from 35 seconds to 0.6 seconds.

Later in the project we became interested in the complete 5×5 room set but the method described above requires too much memory so we could not generate this set. There is no need to store the entire power set then run the valid room check, we can check each candidate room immediately after generation. To implement this we generate candidate rooms from numbers and check if these are valid. We convert a number to a room by interpreting the number as a binary string and assigning each bit to a cell in the room. A zero means the cell is empty and a one means the cell contains an obstacle. For example the number 32 creates a room with an obstacle in the fifth cell (when generating 4×4 rooms this cell is immediately south of the starting cell). By counting from 0 to 2^A where A is the room area we can efficiently generate a complete room set and we can now generate the complete 5×5 set (1,054,065 rooms) in about five minutes.

3.3 Path Plots

The result of a simulation is a list of (x, y) coordinates representing the location of the robot at each simulation step. We found it useful to create visualisations of these paths, for example Figure 3.3. We colour code cells so we can easily identify which areas of a room where covered perfectly (green), which areas contained inefficiencies (yellow) and which areas were missed (white). In large rooms with complex paths it can be difficult to spot where the path ends after a collision or a loop so we colour these cells red.

Initially, we drew the path through the center of each cell which creates ambiguous paths plots such as Figure 3.4a. In these plots it is not always possible to tell which direction the robot took and the exact number of visits to each cell. To address this, we calculate $\mathbf{c}$, the coordinate of each point which the path line must pass through using:

$$\mathbf{c} = (x + o, y + o) \tag{3.1}$$

$$\text{where } o = \frac{1}{2} + 0.15\left(c - \frac{v-1}{2}\right) \tag{3.2}$$

where x and y is the robot location from the path list, c is the number of times which this cell has been visited before, v is the total total visits to this cell over the entire simulation and 0.15 is a scaling constant which we found gave the most visually appealing plots. This equation spaces the path points out along the leading diagonal of each cell which allows us

```
[(0, 0), (0, 1), (0, 0), (1, 0), (1,
1), (1, 2), (0, 2), (0, 3), (1, 3),
(2, 3), (2, 4), (3, 4), (3, 3), (3,
2), (4, 2), (4, 1), (3, 1), (2, 1)]
```

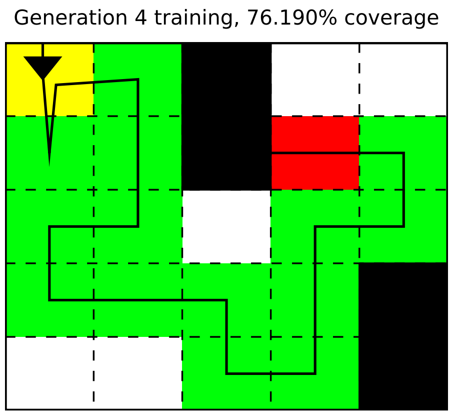


Figure 3.3: An example of a path (left) through the default training room found by an early version of our training method, and the visualisation (right). The robot begins in the top left corner and travels in the direction indicated by the arrow. Any cells visited exactly once are coloured green and cells visited more than once are coloured yellow. Cells coloured in red are the location where the robot made a mistake causing the end of the simulation.

to follow the robot's path (Figure 3.4b). These lines appear cluttered when the robot makes many visits to a single cell but we found this preferable to the ambiguity of the old plots.

3.4 Architecture

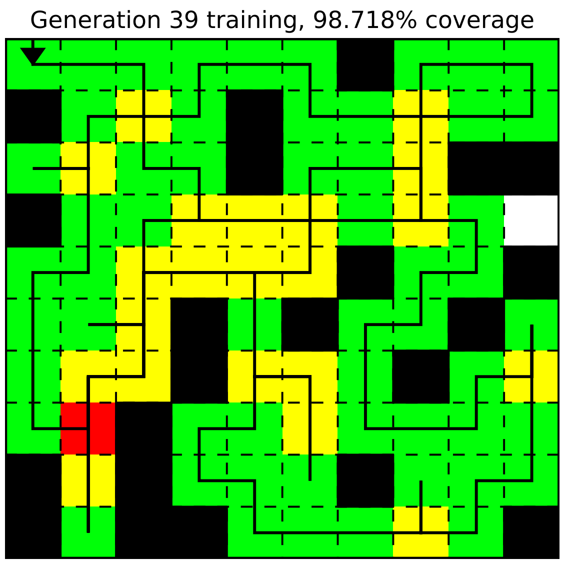
An overview of the main objects in our program can be seen in Figure 3.5. Below we describe the functionality of each:

- **GA Base**: This object is extended by an implementation of a genetic algorithm to find the weights of the network. We extended this object by a variant of the genetic algorithm (discussed in Chapter 4) and our implementation of the canonical genetic algorithm. Both implementations are given a dictionary of hyperparameters (e.g. maximum number of generations, population size, mutation rate) a fitness function and a testing function. Both versions of the genetic algorithm are implemented in general terms, there is no knowledge of simulations, neural networks or robots in these objects. This removes duplication between the genetic algorithm implementations.

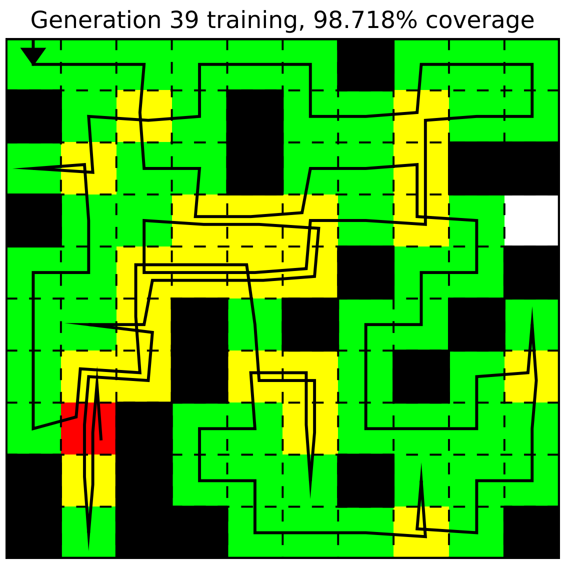
- **Parameters**: Contains the hyperparameters for the genetic algorithm and information to generate the training and testing room sets. These parameters can be overridden by the user at the command line, for example to see the effect of a higher mutation rate or to test with more rooms than usual.

- **Genetic Operators**: A library of all genetic operators (crossover, mutation and selection functions) used throughout the project. These functions can be used in any implementation of a genetic algorithm, so this library reduces duplication.

- **Generational Statistics**: Further reduces duplication between implementations of the genetic algorithm by implementing common functionality which is run after each



(a) An example of our initial path plots.



(b) An example of our final path plots.

Figure 3.4: Effect of the path offset equation. On the left the path line always passes through the center of a cell, on the right the path line is offset by a small amount along the leading diagonal of the cell with each visit.

generation and at the end of the genetic algorithm. This includes projecting genotypes into two dimensions with t-SNE and saving testing paths to disk.

- **Room**: An immutable room storing the dimensions and obstacles of a single room.
- **Room Generator**: Implements the room generation procedures described above to randomly generate sets of rooms.
- **Simulation**: Stores the state of a single robot in a single room. The simulation is advanced by repeatedly passing the robot's next move. A simulation measures coverage and checks if the robot ran into an obstacle or is stuck in a loop. The inputs to a network are also calculated by the simulation object.
- **Phenotype**: A base class for objects which can convert a list of numbers (the genotype) found by the genetic algorithm to a neural network (the phenotype). We implemented a feed forward and convolutional phenotype as well as a phenotype which can grow and take extra inputs each time the genetic algorithm converges to a local maximum.
- **Graph**: For each phenotype we implemented a matching graph object which calculates the output of a network given the parameters of the network and the inputs. We implemented graphs which run a neural network using TensorFlow and graphs using natively implemented functions to find the fastest function.
- **Fitness Single**: This object calculates the fitness (i.e. the coverage) of all robots in a generation using a single room. Instances of this object are given a list of genotypes, for each genotype we convert to the phenotype using a phenotype object and create a

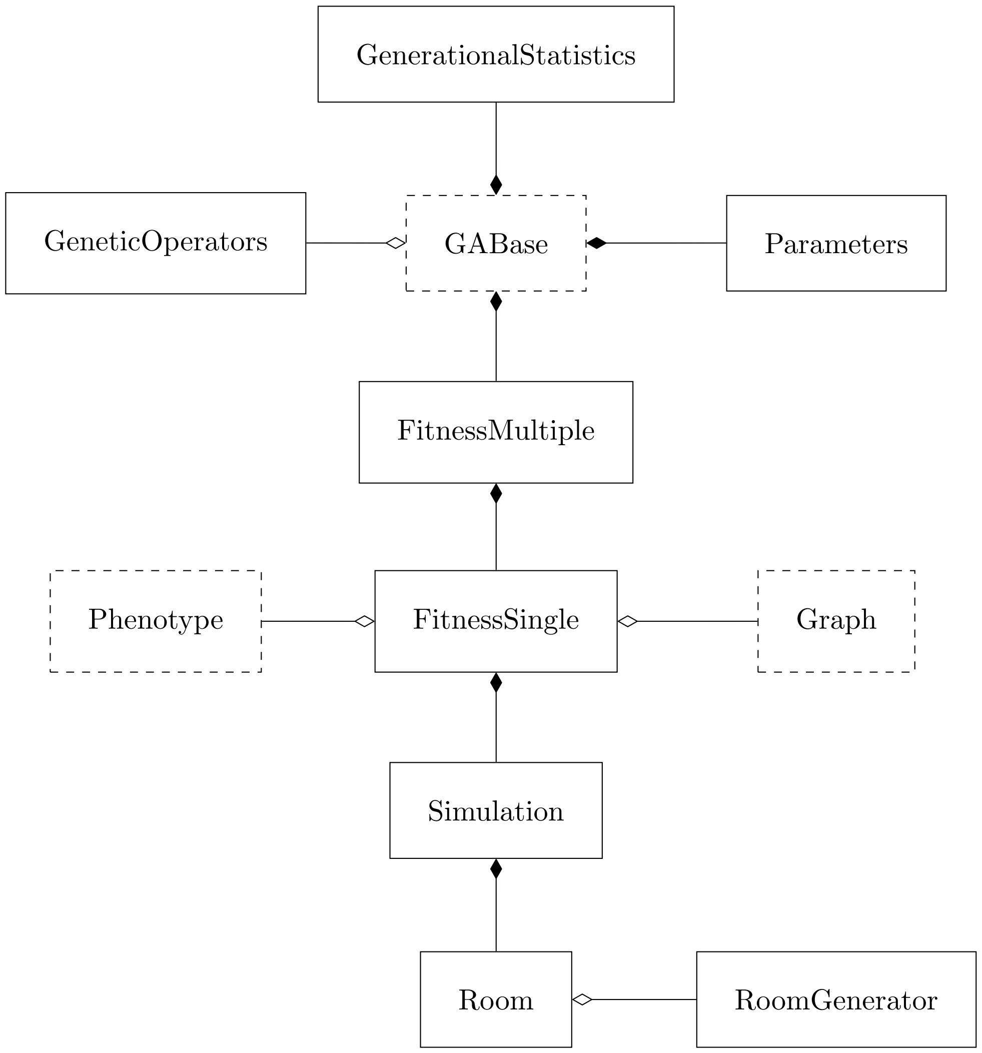


Figure 3.5: Object composition and aggregation diagram of our implementation showing the objects involved in an instance of the genetic algorithm. Objects drawn with a dotted border are super classes.

new simulation object. The simulation and graph objects are used to move the robot around the room to determine the coverage of the robot.

- **Fitness Multiple**: This object calculates the average coverage over all given rooms for all robots in a generation. This is done by using the fitness single object to calculate the coverage per room per robot then averaging across robots to find the final fitness for each robot.

Additionally, we have the following two convenience classes:

- **Output**: Handles the logic to save results to disk such as creating a new directory for each trial and zipping and compressing directories to avoid the file count quota.
- **Builder**: Uses command line options to construct the objects listed above and contains the default values for each hyperparameter.

3.4.1 Program outputs & plotting

All scripts and objects created for this project are for either data generation and running experiments, such as the phenotype classes, or data visualisation such as the path plots. Each experiment uses the data generation half of the project to generate results which are saved to a new directory. These results, or older results can be read by scripts in the data visualisation half to produce the path plots and graphs in this report.

The split between generation and visualisation causes more complexity in the code than plotting results immediately after generation from the machine's memory. However, this approach allows us to plot the effect of improvements to our training method by using data saved at the start of the project.

3.5 Optimisation

The aim of this project is not to produce a highly optimised training system, however early versions of our implementation were prohibitively slow: the hyperparameter sweep experiments carried out during the project (Section 4.6) would have taken 70 hours to complete, we reduced this to 90 seconds.

Our simplest execution of the genetic algorithm using one room and 100 generations with a population size of 100 took 65 seconds to complete. Hyperparameter sweeps require over 4,000 of these executions to try a range of values for each hyperparameter (e.g. 0% to 100% in steps of 1%) and multiple trials for each value (usually 40) to reduce the effect of randomness of genetic algorithms. For example the random initial population of the genetic algorithm could contain an individual very close to a global optimum. We could reduce the runtime by using larger step sizes or fewer trials, however we chose to implement basic optimisations instead.

3.5.1 NumPy v. Tensorflow

We reduced the 65 second runtime of a single run to 45 seconds by reducing the data sent to TensorFlow and by calculating more network outputs per graph execution. This is still much slower than we expected; the inefficiency seems to be caused a high overhead when switching between Python and Tensorflow. For example, when a new generation is created the synaptic weight matrices for each network were sent to TensorFlow one by one. This took about 9 seconds for 100 generations, so instead we now send all matrices at once which is 8 times faster on average.

TensorFlow is designed to execute a complex graph such as a neural network with a back propagation trainer several times, whereas we need to execute a relatively simple graph hundreds of thousands of times. We therefore implemented neural network evaluation using the NumPy library. This reduced the runtime of a single trial from 45 seconds to 3 seconds.

3.5.2 Parallelization

We can run a separate instances of the genetic algorithm on each CPU core, and furthermore we can run instances on multiple machines. We use Condor, a high throughput batch processing system provided by the department to run our program on idle machines. Each instance of the genetic algorithm saves its results to disk, which are combined when all instances have finished to be plotted. With many idle machines we can quickly perform the hyperparameter sweeps or other experiments which involve many executions of the genetic algorithm.

Chapter 4

Single Room Case

In this chapter we present the improvements we built to train a neural network to cover a single given room in chronological order. We decided to begin with the single room case, usually the room in Figure 4.1, in order to build a basic working system which could be extended in the future to handle more complex cases.

4.1 Starting Point

Our initial implementation did not include the history inputs and used two hidden layers of 10 neurons each to solve the room in Figure 4.1. We used a 2% mutation rate drawn from a uniform distribution, 80% one point crossover rate and kept 10 elites from a population of 100. This performed poorly, in 1,000 trials of this implementation with 100 generations an average of 9.79 or 46.6% of empty cells were visited at least once. None of these trials succeeded in covering the entire room. Increasing the number of generations does not improve the performance significantly as 300 generations increases the average coverage to 52.57%, which shows us that the genetic algorithm typically converges to a local maximum. An example of one of these paths is shown in Figure 4.2.

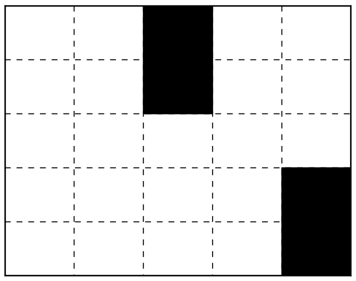


Figure 4.1: The 5×5 room used for most experiments.

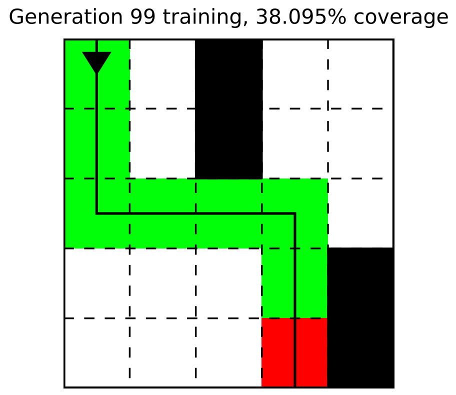


Figure 4.2: A typical path found by our initial training method. Any cells visited exactly once are coloured green. Cells coloured in red are the location where the robot made a mistake causing the end of the simulation.

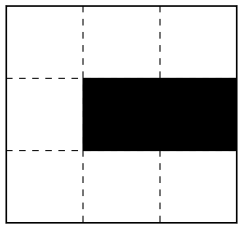


Figure 4.3: A 3×3 room which requires backtracking (visiting a cell more than once) to solve if starting from the top left.

4.2 Backtracking Problem

The major issue with this implementation is that a robot able to retrace its steps can not be trained because the input vector only depends on the position and room state. Therefore the robot will be caught in a loop if a position is visited for the second time. This reduces the performance as the robot must find a path with visits every cell exactly once instead of at least once, a harder problem. Additionally, this restriction leaves some rooms unsolvable. The room used for training (Figure 4.1) can be solved by visiting each exactly cell once but the room in Figure 4.3 cannot be solved without backtracking. We found two solutions to this problem which we will describe next.

4.2.1 Simulation memory & valid move checks

Our first solution was to extend the simulation to help the network by ignoring outputs of the network which would cause the simulation to end. This prevented the simulation termination conditions from triggering. The challenge for the network was to solve the room in as few moves as possible.

Method

We modified the simulation to keep track of the actions made at each cell and ignored any output neuron corresponding to an action previously taken in the same cell or an action which would lead the robot into an obstacle. If all four actions have already been taken then the memory for the cell is cleared and the valid action with the highest confidence is taken instead. This guarantees that any action taken has not been tried before in the same cell, if such an action exists. We also limited the number of moves which the robot can take in a simulation as robots will run until the room has been solved. To avoid the backtracking problem in Figure 4.3, the network must learn to ensure that the second highest output neuron will cause the robot to retrace its steps correctly.

Results

Using the same settings as in Section 4.1 we achieve much higher fitness. On average, 18.84 or 89.7% of empty cells are visited at least once after 100 generations of training. Additionally 17.5% of trials ended with the algorithm solving the room.

4.2.2 History inputs

Our second solution was to add simulation state inputs to the network and leave the simulation unmodified.

Method

We added the history inputs described in Section 3.1.3 to the network. This solves the backtracking problem of the room in Figure 4.3 by giving the network a different input on the second pass over a cell, allowing the possibility to take a different action.

Results

We performed an independent samples t-test to compare the best fitness found for the room in Figure 4.1 with and without the history inputs. There was a significant difference in coverage when including the history input (mean = 12.22, SD = 2.5) and when excluding the history input (mean = 9.79, SD = 2.16); unequal variances t(390) = -10.0, $p < 0.05$ when comparing cells covered over 200 trials. The room in Figure 4.1 can be solved without backtracking and therefore without the history inputs, the test shows that including the history inputs can improve performance as well as solving the backtracking problem. The expected fitness is lower when excluding the history inputs because the robot will not visit any new cells after visiting a cell for the second time, therefore the robot must find a path which visits each cell once. A path found using history inputs is shown in Figure 4.5.

4.2.3 Evaluation

The simulation memory with valid move checking performs far better than the history inputs, however we decided to move forward with the history input solution and remove the memory and the move checks for the remainder of the project.

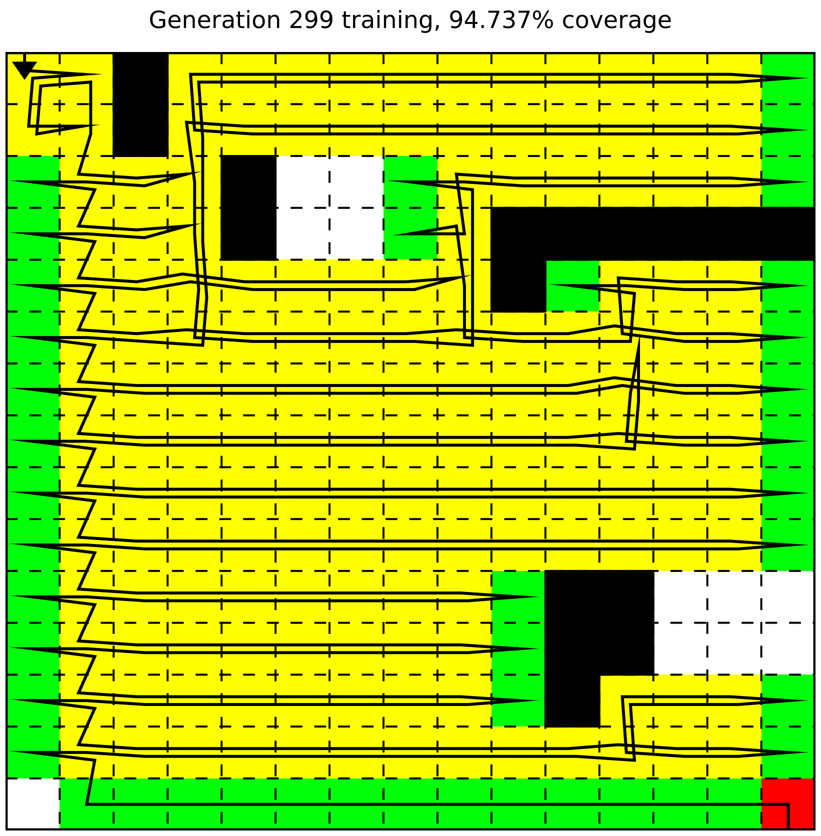


Figure 4.4: A solution to a 15×15 room found by the genetic algorithm using simulation memory and valid move checks. The robot begins in the top left corner and travels in the direction indicated by the arrow. Any cells visited exactly once are coloured green and cells visited more than once are coloured yellow. Cells coloured in red are the location where the robot made a mistake causing the end of the simulation.

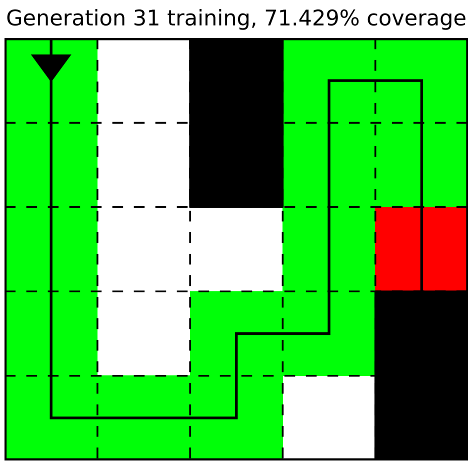


Figure 4.5: A typical path using history inputs. The robot begins in the top left corner and travels in the direction indicated by the arrow. Any cells visited exactly once are coloured green and cells visited more than once are coloured yellow. Cells coloured in red are the location where the robot made a mistake causing the end of the simulation.

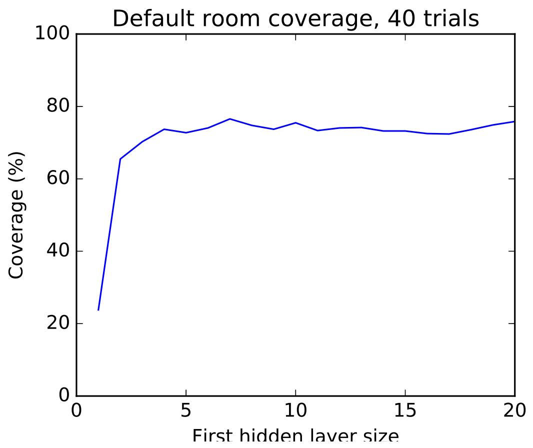
When using the memory and move checks, the networks consistently evolved the behaviour seen in Figure 4.4, where the robot spent much of the time retracing its path instead of moving on to new cells. For almost all cells in the bottom left quadrant of Figure 4.4 the order of the network's outputs from highest to lowest correspond to east, west then south. The simulation memory causes the robot to choose these actions in turn. Choosing the same order of actions in so many cells simplifies the problem significantly and shows that the networks rely heavily on the simulation memory. This behaviour is very effective in covering the room but requires too many steps, and ideally the networks should not need help from an extra system.

4.3 Optimising Network Topology

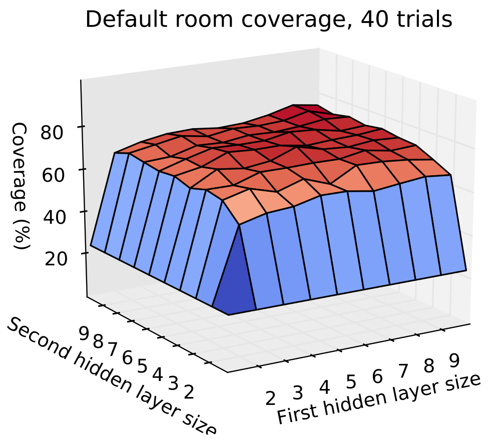
4.3.1 Method

Next we searched for the optimal feed forward topology. Our fitness function is fast enough that we can find the optimal topology by a brute force search, as each trial runs for less than 10 seconds on a single CPU core. We will find the optimum number of hidden neurons for a network with one and two hidden layers. Then we can compare the performance of the optimal network with one hidden layer, two hidden layers and no hidden layers to find our final topology.

Genetic algorithms are randomised so to ensure that any difference in performance is not caused by chance we ran each topology 40 times for 100 generations with different random seeds and recorded the maximum coverage for each trial. We averaged the 40 maximum coverages for each network topology to calculate the final coverage for that topology.



(a) Using a single hidden layer.



(b) Using two hidden layers.

Figure 4.6: A brute force search for the optimal topology by comparing the average coverage after 100 generations. There is no significant difference between one and two hidden layers.

4.3.2 Results

Using no hidden layers we achieved an average coverage of 74.5%. With one hidden layer we find there is no advantage of using more than four hidden neurons, which gives an average coverage of 72.5% (Figure 4.6a). With two hidden layers (Figure 4.6b) the maximum average coverage is 72.6% by using five hidden neurons in each layer, more hidden layers does not improve the coverage.

4.3.3 Evaluation

There is no significant difference between no hidden layers, the optimum one layer and two hidden layer topologies. We chose the simplest topology with no hidden layers. We could have continued and tried a network with three hidden layers but this would have been time consuming with so many configurations and there is no evidence that more hidden layers will increase the coverage.

This was a surprising result as we expected some change in coverage when using none, one and two hidden layers because network topology is an important consideration for other applications. After applying the next improvement (Section 4.4) and repeating this experiment, we find that no hidden layers provides a significant increase in coverage compared to other topologies. We discuss this in Section 4.4.3.

4.4 Room Input

At this point in the project, the inputs to the network consisted of the room, position and history inputs. The position and history inputs vary as the robot moves around the room, however the room inputs never change because the room is static. The constant room inputs reduces the effect of the weights between the room inputs and the output neurons to a bias

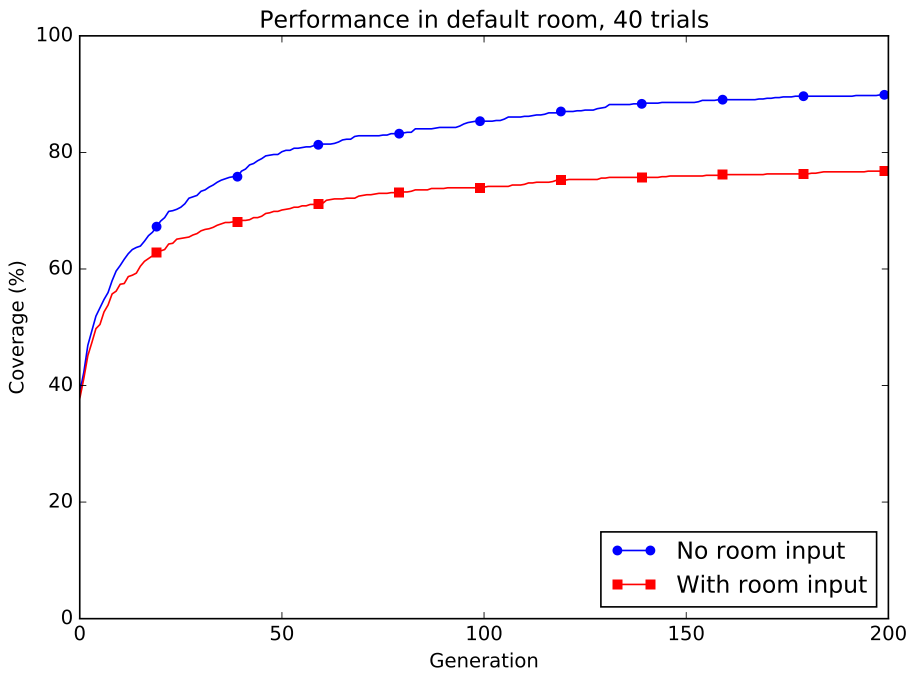


Figure 4.7: Comparison of average coverage per generation with room input enabled and disabled.

on each output neuron, instead of encoding useful behaviour. We tried removing the room inputs to see if these biases provided any value.

4.4.1 Method

We ran our program with and without the room inputs. As before we run 40 trials for each version with different random seeds to reduce any randomness in the result. We record the maximum coverage of each generation for all trials to plot the average maximum coverage per generation with and without room inputs.

4.4.2 Results

Figure 4.7 shows the average maximum coverage per generation with room inputs included and excluded. Excluding the room inputs improves the average performance from 76.7% to 89.4%. We ran all trials for 200 generations instead of 100 to show that the room input performance converges to a lower coverage and does not have a slower start due to the higher number of parameters. An example of a path created without room inputs is shown in Figure 4.8. With this change, for the first time in the project, a small number of trials (approximately 17%) achieve 100% coverage within 200 generations.

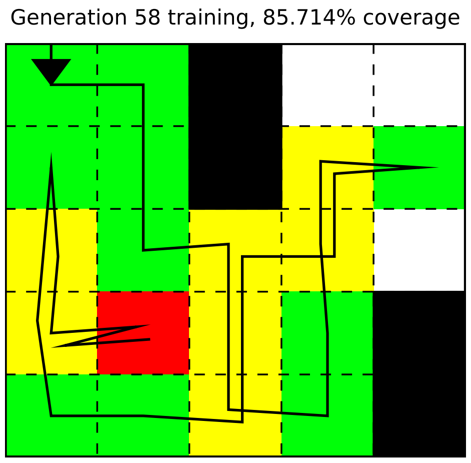


Figure 4.8: A typical path found within 100 generations with room inputs disabled. The robot begins in the top left corner and travels in the direction indicated by the arrow. Any cells visited exactly once are coloured green and cells visited more than once are coloured yellow. Cells coloured in red are the location where the robot made a mistake causing the end of the simulation.

4.4.3 Evaluation

The connections between the position inputs and output neurons form a lookup table where the weight of a connection is the confidence that the action of the output neuron should be taken when the robot is in the cell of the position neuron. As an example, all position input neurons are connected to each of the four output neurons and at the start of the simulation all position neurons are set to zero except the neuron of the starting cell which is set to one. Therefore at the start of the simulation the value of each output neuron is equal to the weight of the connection between the starting position neuron and the output neuron.

This creates a fixed path which can be modified only by the history inputs but not by any obstacles in the room. Therefore the robot cannot learn to clean an arbitrary room. If the robot's first move is to go south and the robot is set in a room with an obstacle immediately below the starting position, the robot will run into this obstacle and will only clean the starting cell. However, learning behaviour to solve an arbitrary room will require training on multiple rooms and testing on unseen rooms which was beyond the scope of this part of the project. Instead, we focus on improving single room performance, so we disabled the room inputs.

If we repeat the topology experiments in Section 4.3 we find there is now a significant coverage increase between no hidden layers (mean = 0.85, SD = 0.078) and the optimal single hidden layer topology (mean = 0.8, SD = 0.094); unequal variances t(76.0) = 2.3, p = 0.0235. There is a greater increase between no hidden layers and the optimal two hidden layer topology (mean = 0.77, SD = 0.11); unequal variances t(72.0) = 3.6, p = 0.000662.

4.5 CoSyNE Algorithm

Our next idea was to use a more powerful genetic algorithm which could find better weights on average than our implementation of the canonical genetic algorithm. Cooperative Synapse Neuroevolution (CoSyNE) [16] evolves each synapse in the network as a separate species. This sustains diversity in the population which allows the algorithm to avoid local maximums. The paper introducing CoSyNE shows the algorithm outperforming several other methods to train a neural network, included algorithms which evolve the network topology in parallel with the network weights and the single species approach which we use.

4.5.1 Method

We implemented the CoSyNE algorithm as described in [16]. An implementation of this algorithm exists but we found it easier to modify our genetic algorithm than to work the C++ implementation into our program.

We used the same genetic operators (roulette selection, uniform additive mutation and one point crossover) and the same hyperparameters in our CoSyNE implementation as in the genetic algorithm used in the previous sections (we now call this the canonical genetic algorithm). As before, we ran each algorithm several times to measure the average coverage across all generations.

4.5.2 Results

CoSyNE performed significantly better than the canonical genetic algorithm. Figure 4.9 shows the maximum fitness per generation averaged across all trials. Both algorithms begin from a random population so as expected there is no difference in initial coverage but CoSyNE achieves a higher coverage in generation 50 than the canonical genetic algorithm does at generation 500.

The difference in the algorithms becomes more pronounced when we compare the likelihood of reaching 100% coverage in the room at each generation (Figure 4.10) by calculating the proportion of trials with maximum coverage per generation for each algorithm. We ran both algorithms for more generations than in previous experiments to show the coverage beginning to converge. A path which achieves 100% coverage is shown in Figure 4.11.

4.5.3 Evaluation

CoSyNE gave better results without increasing the runtime of the program or the complexity of the code so we used this algorithm for the remainder of the project.

4.6 Hyperparameter tuning

As discussed in Section 2.4.6, a disadvantage of genetic algorithms is the large number of hyperparameters to set. Throughout the project we performed many hyperparamter sweeps for the CoSyNE algorithm and canonical genetic algorithm to find the optimal values. Here we show the result of a sweep after implementing the improvements presented above.

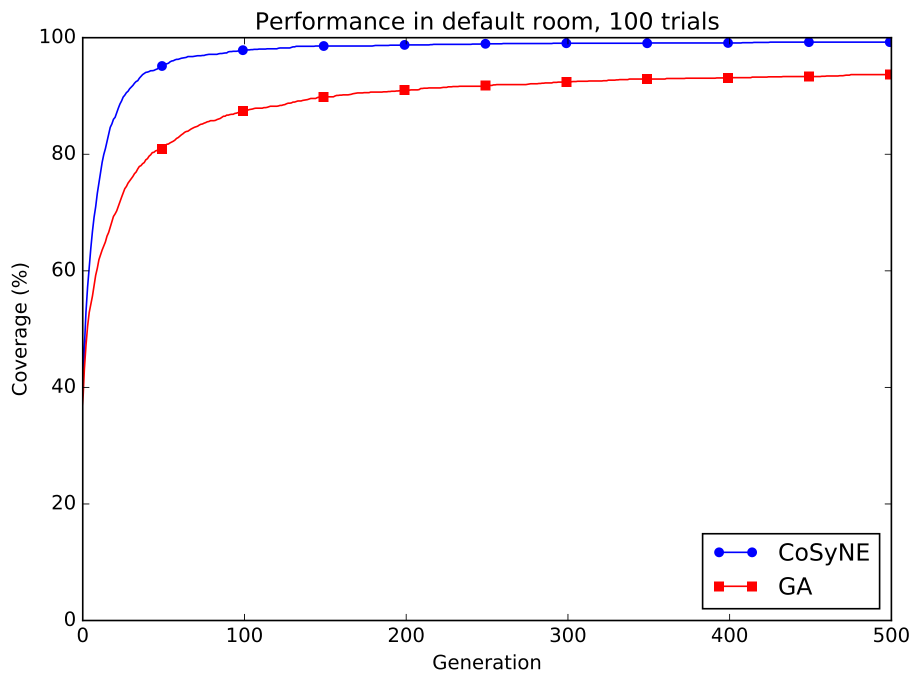


Figure 4.9: Comparison of coverage per generation using the canonical genetic algorithm (GA) and the CoSyNE algorithm.

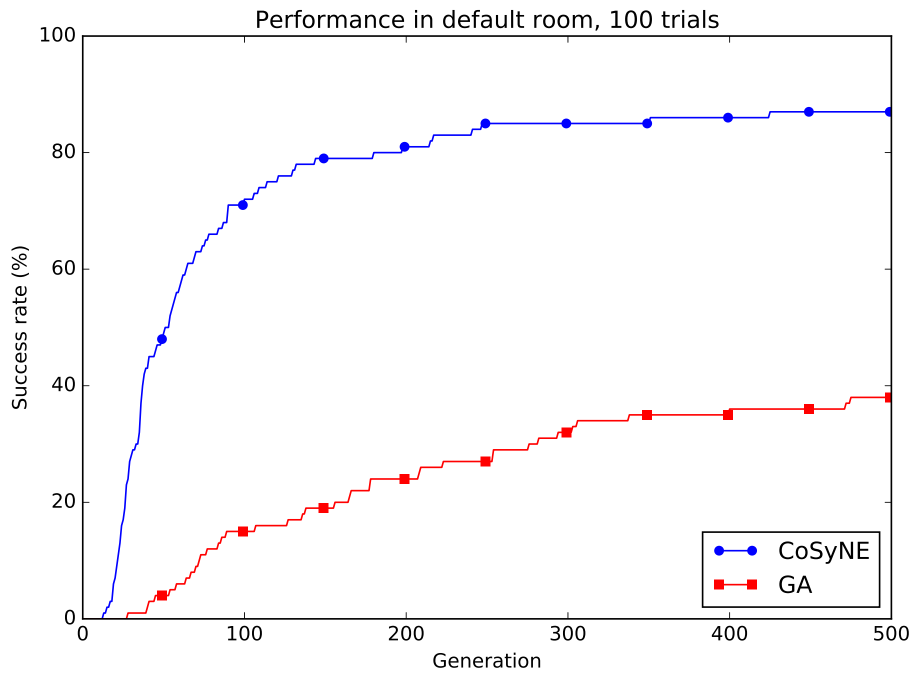


Figure 4.10: Comparison of the proportion of trials which reach 100% coverage (the success rate) per generation for the canonical genetic algorithm (GA) and the CoSyNE algorithm.

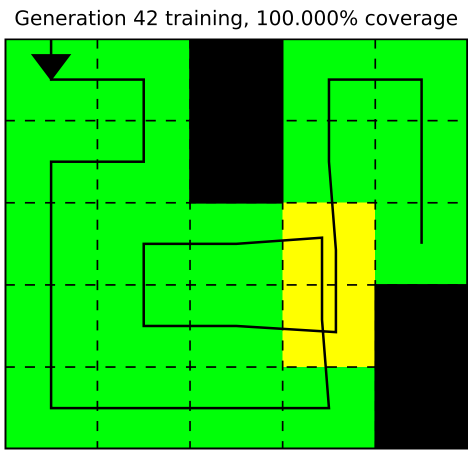


Figure 4.11: A solution found using the CoSyNE algorithm. The robot begins in the top left corner and travels in the direction indicated by the arrow. Any cells visited exactly once are coloured green and cells visited more than once are coloured yellow.

4.6.1 Method

We search for the optimal value for each hyperparameter by sweeping over the range of all possible values as this is the simplest approach and the program is fast enough for this method (all sweeps were performed in under two minutes by running across multiple machines). All tests were performed in the room in Figure 4.1 and measure the maximum coverage achieved within 20 generations averaged over 40 trials. We reran all tests below each time a parameter was changed to check if another parameter has been affected.

4.6.2 Results

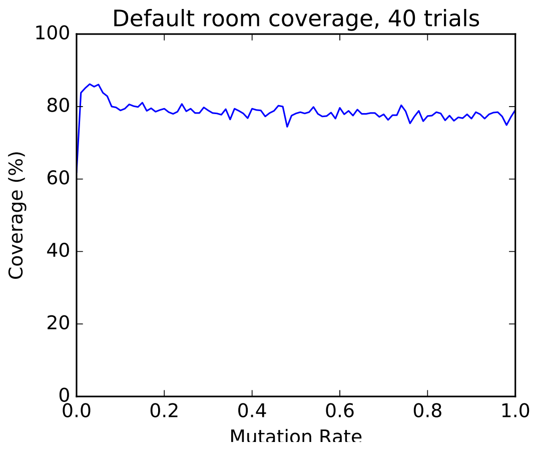
The result of each sweep and the value chosen for each hyperparameter is shown in Figure 4.12.

4.6.3 Evaluation

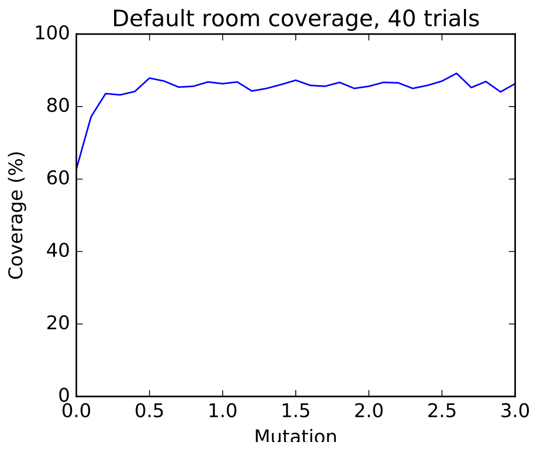
These experiments show that although there more hyperparameters than other optimisation methods, genetic algorithms are not particularly sensitive to their hyperparameters. None of the values found are unexpected so choosing reasonable values for each would have resulted in similar performance.

4.7 Improved Genetic Operators

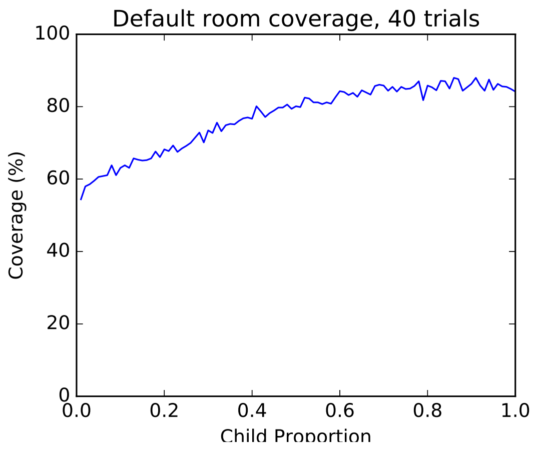
So far we have relied on standard genetic operators namely one point crossover and uniform mutation to create new candidate solutions. By using including knowledge about the application in our genetic operators we can further improve the performance.



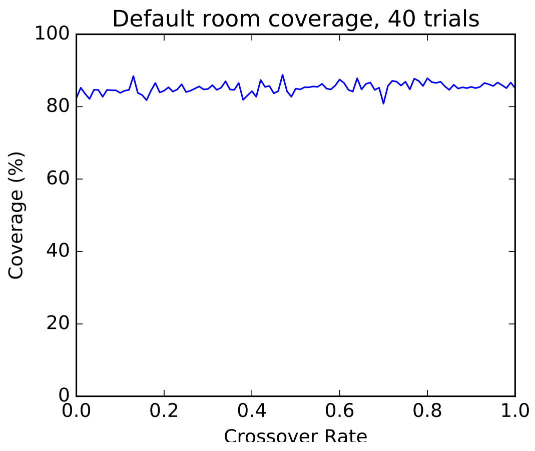
(a) A mutation rate less than 1% gives significantly worse performance. We chose 4% mutation rate.



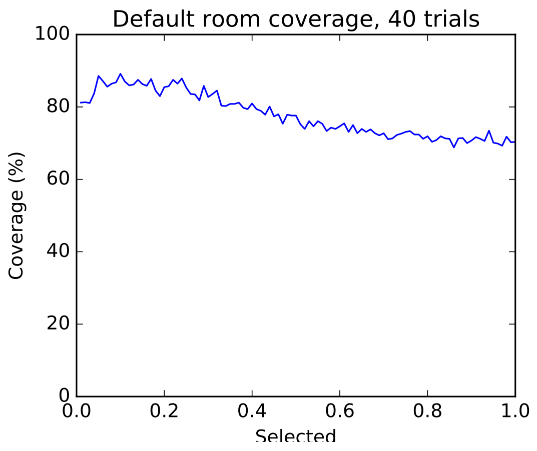
(b) Maximum mutation magnitudes greater than 0.5 does not affect performance. We use 1.0 magnitude.



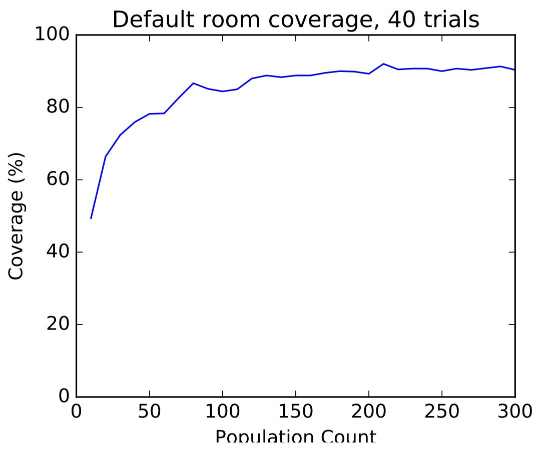
(c) Child proportion is the proportion of genotypes which are replaced by newly generated genotypes. We replace 90% each generation.



(d) There is no significant difference between crossover rate of 0 and 1, $p = 0.0626$. We use 0% crossover rate.



(e) The top x% of genotypes which may be used to create the next generation. We select from the top 10%.



(f) We use a population size of 100 as a trade off between runtime and performance.

Figure 4.12: Example of a hyperparameter sweep with different values against coverage. We try 100 different values for each hyperparameter.

4.7.1 Method

We create two new genetic operators. Our new mutation operator, input permutation begins by randomly choosing some (or none) of the cells in the room then permuting the weights between the position and history inputs for these cells and all outputs. An example of this is shown below:

$$\text{Weight matrix : } \left(\begin{array}{cccc|cccc} p_{1,1} & \mathbf{p_{1,2}} & p_{1,3} & p_{1,4} & h_{1,1} & \mathbf{h_{1,2}} & h_{1,3} & h_{1,4} \\ p_{2,1} & \mathbf{p_{2,2}} & p_{2,3} & p_{2,4} & h_{2,1} & \mathbf{h_{2,2}} & h_{2,3} & h_{2,4} \\ p_{3,1} & \mathbf{p_{3,2}} & p_{3,3} & p_{3,4} & h_{3,1} & \mathbf{h_{3,2}} & h_{3,3} & h_{3,4} \\ p_{4,1} & \mathbf{p_{4,2}} & p_{4,3} & p_{4,4} & h_{4,1} & \mathbf{h_{4,2}} & h_{4,3} & h_{4,4} \end{array}\right) \quad (4.1)$$

$$\text{After input permutation : } \left(\begin{array}{cccc|cccc} p_{1,1} & \mathbf{p_{3,2}} & p_{1,3} & p_{1,4} & h_{1,1} & \mathbf{h_{3,2}} & h_{1,3} & h_{1,4} \\ p_{2,1} & \mathbf{p_{2,2}} & p_{2,3} & p_{2,4} & h_{2,1} & \mathbf{h_{2,2}} & h_{2,3} & h_{2,4} \\ p_{3,1} & \mathbf{p_{4,2}} & p_{3,3} & p_{3,4} & h_{3,1} & \mathbf{h_{4,2}} & h_{3,3} & h_{3,4} \\ p_{4,1} & \mathbf{p_{1,2}} & p_{4,3} & p_{4,4} & h_{4,1} & \mathbf{h_{1,2}} & h_{4,3} & h_{4,4} \end{array}\right) \quad (4.2)$$

Here input permutation has selected the second cell in a weight matrix for a 2×2 room (4.1), so weights from the position and history inputs of that cell ($p_{x,2}$ and $h_{y,2}$) are permuted to give the matrix in (4.2).

Our new crossover operator works similarly, several cells of the room are picked and all weights associated with these cells are swapped between the two parents to create two children.

As usual, we run several trials to compare the difference in performance. As we are close to optimal performance we compare success rates per generation instead of coverage per generation.

4.7.2 Results

Each operator when used individually increases the performance, the effect of both operators is shown in Figure 4.13. We also use uniform mutation to randomly introduce new genes in the population, as otherwise we would only swap genes created with the initial population and never create new genes in subsequent generations.

These operators work well enough to guarantee a solution with 100% coverage within 300 generations but are likely to find a solution within 100 generations.

4.7.3 Evaluation

One point crossover and uniform mutation are general and can be applied to any application when genotypes are a list of numbers, however in our case these operators do not make any meaningful changes to the neural network. Our new operators build on the observation in Section 4.4.3 that the weights form a table of decisions and swap these entries in the hope of finding a better path.

4.8 Summary

We have shown how single room coverage has been improved from 46% after 100 generations to 98.6%. The probability of a single instance of the genetic algorithm finding a solution with complete coverage has increased from 0% after 1,000 generations to 77% after 50 generations

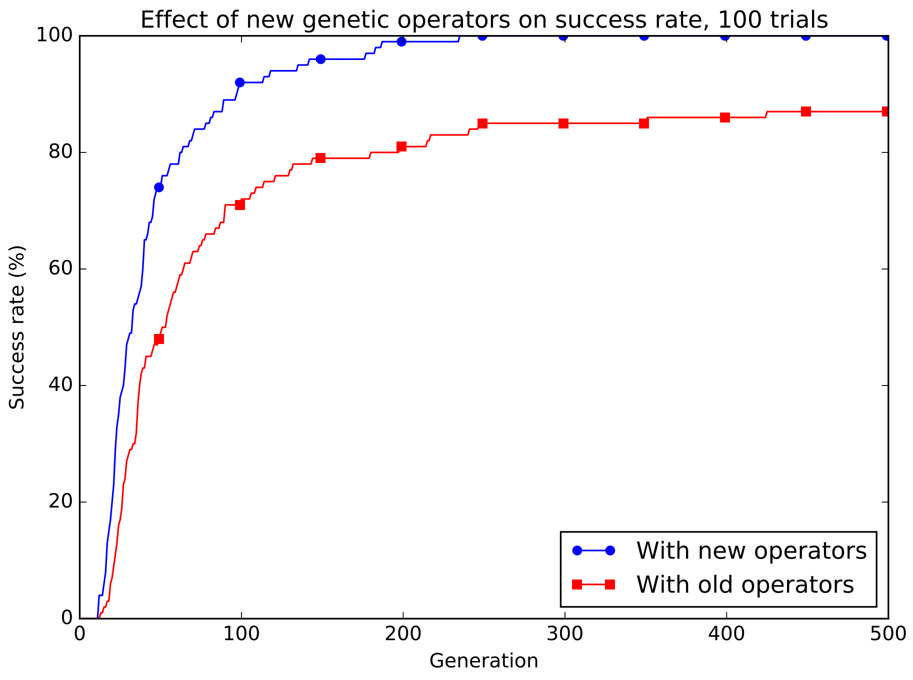


Figure 4.13: Comparison of the proportion of trials which reach 100% coverage (the success rate) per generation when using both new operators.

and 100% probability within 300 generations. We accomplished this by improving our genetic operators, the genetic algorithm and simplifying our model.

It may have been possible to improve the performance in the single room case further, but we could now solve our room reliably within several seconds so we decided to use the remaining time of the project on a more complex problem (Chapter 5).

Chapter 5

Multiple Room Case

During the experiments in Chapter 4 we built a system which could learn a path through a single room, in this chapter we extend this system to learn behaviour which can solve an arbitrary room.

5.1 Single to Multiple Room Modifications

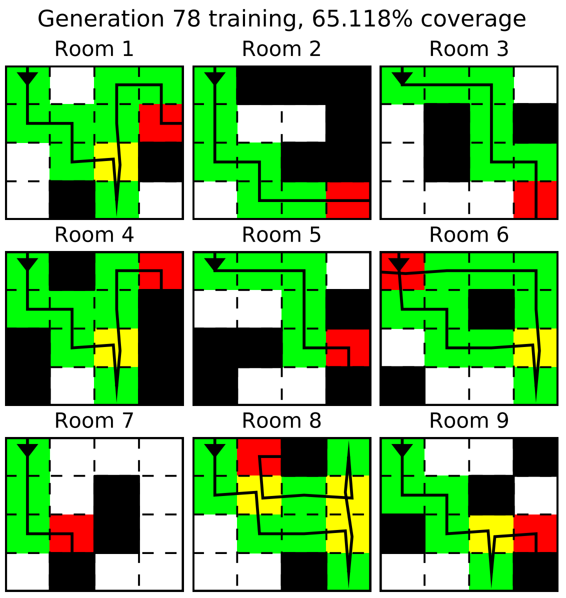
We modified the system to create two disjoint sets of equal sized rooms before the genetic algorithm is run: a small set of training rooms and a much larger set of testing rooms. The fitness of a robot is found by simulating the robot in each training room and taking the average coverage. After training is finished the robot with the highest fitness is run in each testing room. The average coverage in the testing set is a measure of how well the robot's behaviour generalises to unseen rooms. We used the same parameters for the CoSyNE algorithm as before and re-enabled the room inputs.

5.2 Starting Point

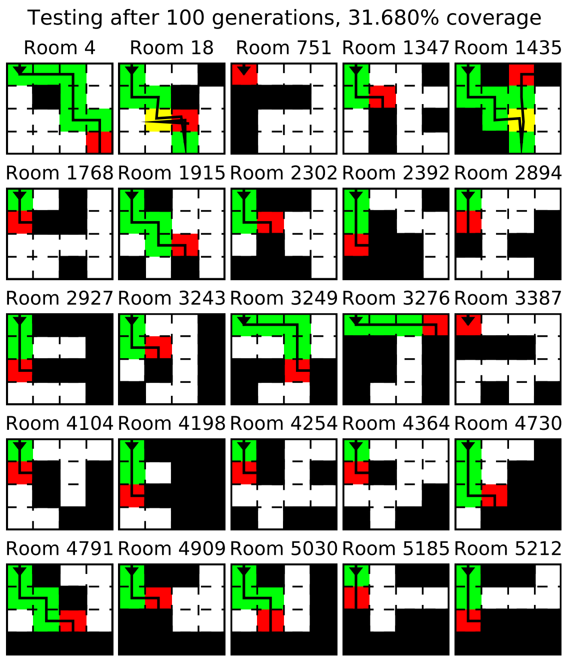
We decided to begin with a few small training rooms and increase the difficulty of the training set as the performance increases. We initially used 9 4×4 training rooms and ran the algorithm 100 times with different training rooms and random seeds. We use the complete set of 4×4 rooms as the testing set.

This first test showed that our system could not learn any intelligent path finding behaviour room from its poor performance. In the testing set the average coverage was 30.0%. On the training set the performance dropped significantly compared to the single room case averaging at 69.3% after 100 generations. The paths taken by one of these robots in shown in Figure 5.1.

We also looked at the effect of varying the number of training rooms (Figure 5.2). As expected, increasing the number of training rooms decreases the training performance while increasing the testing performance. Increasing the number of training rooms past 50 gives diminishing returns in testing performance. We tried using all 5,293 4×4 rooms as the training set to check the maximum testing performance. In the multiple room case, testing performance or average coverage in the testing rooms, is more interesting information than the



(a) Paths taken by the robot through the 9 randomly generated training rooms.



(b) A random sample of 25 of the testing rooms.

Figure 5.1: Paths taken by the best performing robot found within 100 generations. The robot begins in the top left corner and travels in the direction indicated by the arrow. Any cells visited exactly once are coloured green and cells visited more than once are coloured yellow. Cells coloured in red are the location where the robot made a mistake causing the end of the simulation.

training performance as this measures the ability to generalise. We calculate the testing performance per generation by picking the individual with the highest training room coverage and measuring the individual's average coverage in the testing set. Testing performance is not used to calculate fitness, fitness is found using only the training set.

5.3 Convolutional Phenotype

The feed forward phenotype used so far does not use the topology of the inputs to improve the quality of the outputs. For example, inputs from cells close to the robot's position are not treated different to cells far away, however an obstacle next to the robot is much more important than an obstacle on the other side of the room. Convolutional neural networks are used extensively in image processing to address this problem by detecting local features and by exploiting the topology of the inputs. These networks have the additional benefit of reducing the number of weights in the network through the parameter sharing of the convolutional layer.

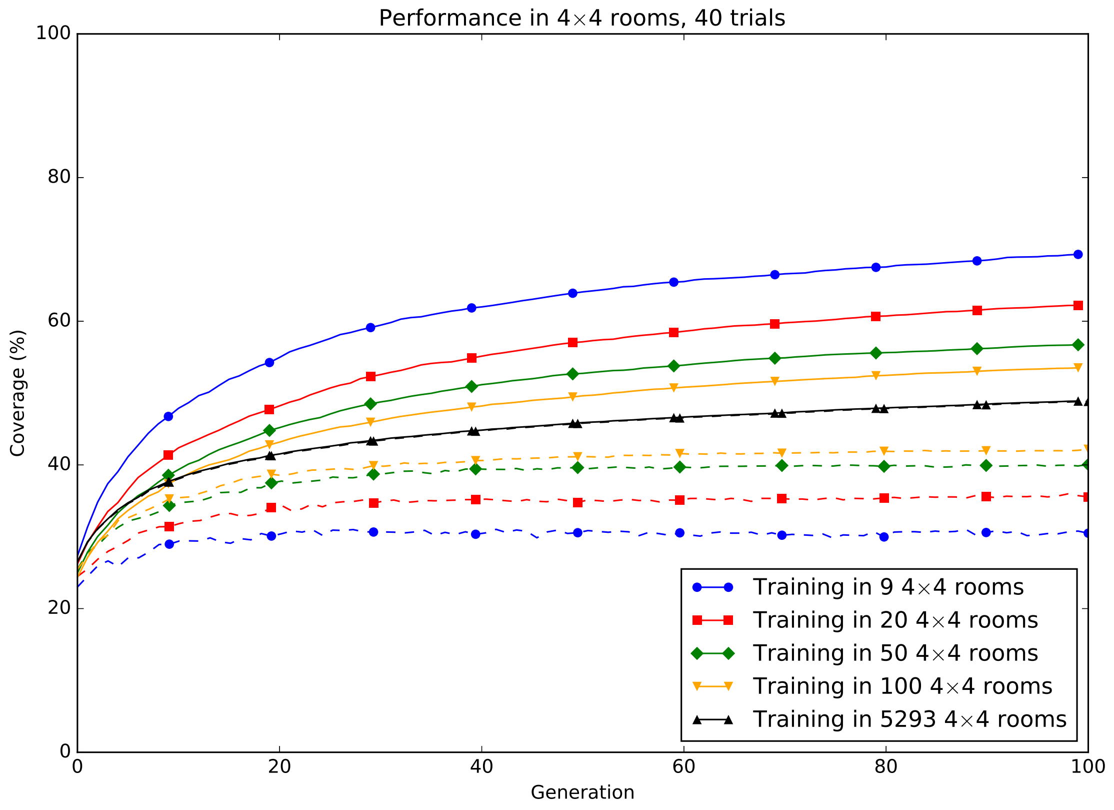


Figure 5.2: Coverage per generation when varying the number of training rooms. Solid lines show coverage in the training set. The dashed lines show the coverage in the testing set (all 4×4 rooms). Training with all 5,293 4×4 rooms took 5.5 hours, training with 9 rooms took 8 minutes.

Filter shape	Stride	Depth			
		1	2	3	4
2×2	1	0.291 ± 0.031	0.311 ± 0.035	0.318 ± 0.033	0.326 ± 0.035
	2	0.283 ± 0.034	0.295 ± 0.037	0.300 ± 0.030	0.314 ± 0.031
3×3	1	0.303 ± 0.034	0.307 ± 0.031	0.304 ± 0.035	0.314 ± 0.032

Table 5.1: Testing performance after training for 100 generations on 9 4×4 rooms using a convolutional phenotype.

5.3.1 Method

We implemented a neural network with one convolutional layer followed by a fully connected layer. We varied the stride, filter shape, output depth and padding. We used zero-padding for the history and position inputs as the robot will never be in a cell outside of the room, and one-padding for the room input to mark cells outside of the boundaries of the room as obstacles. All experiments are averaged across 40 trials as before. We initially implemented a convolutional neural network manually to reduce CPU time but we switched to the TensorFlow implementation of convolutional nets to ensure that the results of this experiment were correct.

5.3.2 Results

The results of this experiment are in Table 5.1. The best convolution topology is a network with a 2×2 filter shape, stride of one and output depth of four. The increase in performance is statistically significant compared to the feed forward topology with a mean coverage of 0.304 and a standard deviation of 0.028 at 0.01 level of significance.

5.3.3 Evaluation

By a simple search through possible topologies we can improve the average coverage. With more experience we may have been able to find these, or better, convolutional hyperparameters without such a search. Instead of continuing to work with convolutional networks, we found that a different phenotype (described below) gave much better coverage performance so we continued the project with this new phenotype instead.

5.4 Localised Phenotype

After our experiments with the convolutional phenotype we decided to remove the final fully connected layer and only apply the convolutional kernel to the inputs around the robot's current position. We remove the position inputs as the robot will always be in the center of this kernel and this input would never change. If we use a filter shape of 3×3 then only the room and history inputs of the cell containing the robot and the cells adjacent to the robot will be passed to the network. As in the convolutional phenotype we use zero-padding for the history inputs and one-padding for the room inputs.

Phenotype	Condition		
	Room completed	Hit obstacle	Loop
Localised	3.45%	19.38%	77.17%
Full input	0%	99.3%	0.7%

Table 5.2: Conditions causing the end of a simulation. Averaged over 20 trials of a localised network with a radius of one in 1,000 8×8 testing rooms.

5.4.1 Method

We define the local radius of a network as the maximum distance between a cell included in the inputs and the current position of the robot. For example, a local radius of 1 gives a 3×3 block centred around the robot and local radius of 2 gives a 5×5 centred around the robot. We tried different local radii averaged across multiple trials. We also ran the old feed forward phenotype in the to see the difference in performance.

5.4.2 Results

Localised networks perform better than the feed forward or the convolutional phenotype. In Figure 5.3 we show the results of using the localised phenotype in 8×8 rooms with localised networks covering much more cells than the testing set. We ran all trials for more generations than usual to show the testing performance of a local radius of two network converge to a lower value than with a local radius of one. An example of paths taken with this network is shown in Figure 5.4.

5.4.3 Evaluation

The localised networks need fewer weights in general than full input networks. With a local radius of 2, a network will contain 200 weights while a full input network in an 8×8 room contains 768. The increased performance may be partly due fewer weights creating a simpler problem. However this cannot be the only explanation as a local radius of 5 requires 968 connections, more than the full input network with significantly worse performance.

Instead, the performance gain is likely due to the fact that obstacle avoidance is easier to learn if the inputs are presented in this manner. For example, to avoid hitting an obstacle east of the robot the weight between the room input neuron immediately east of the robot position and the east output neuron must be a large negative number. In full input networks this behaviour is not as easily encoded. We can show this by comparing how simulations end using localised robots (with local radius of one) and the old full input robots in 8×8 testing sets (Table 5.2) to find that robots with a localised network are 5 times less likely to end a simulation by colliding with an obstacle.

The testing performance decreases with increasing local radius. This is a problem because the low local radius prevents the robot from making decisions which will benefit the robot in the long term. The extra information gained from a higher radius does not offset the cost of the quadratically increasing number of weights. We address this problem in the next section.

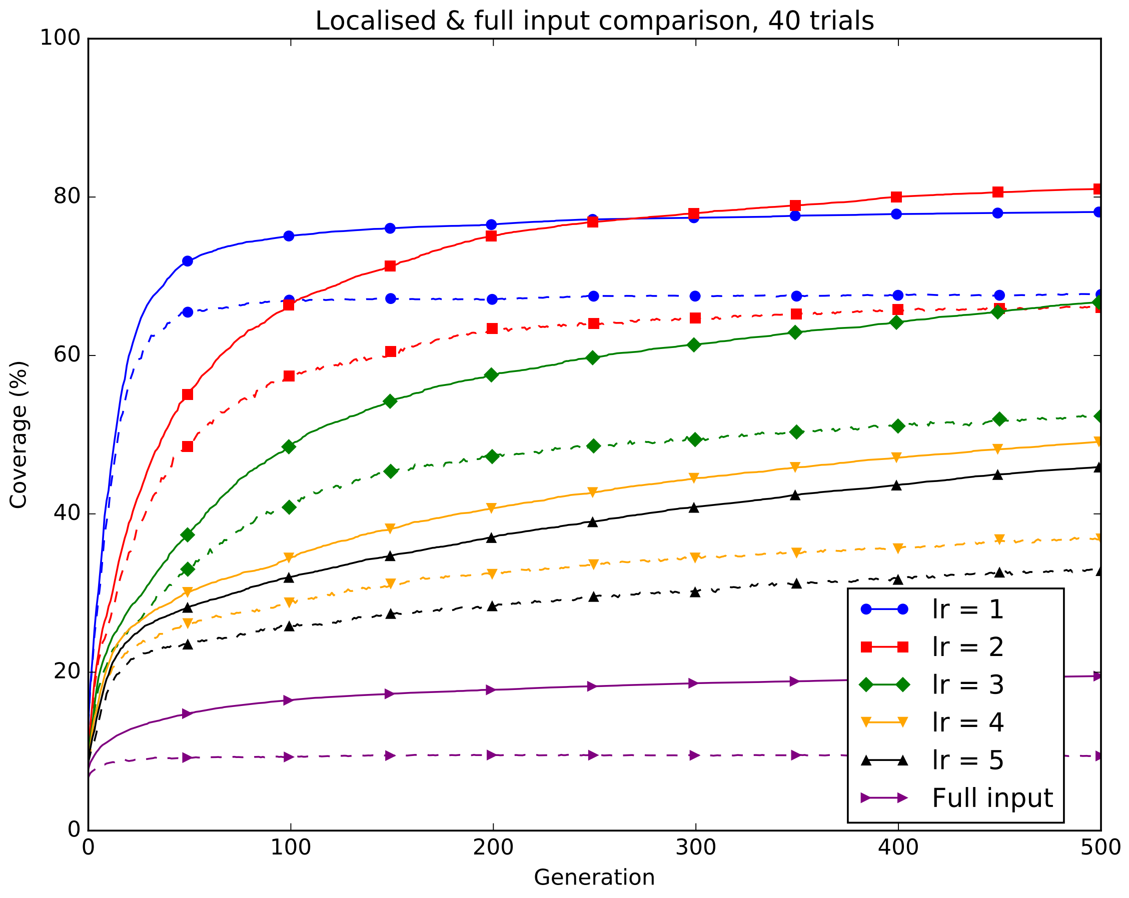
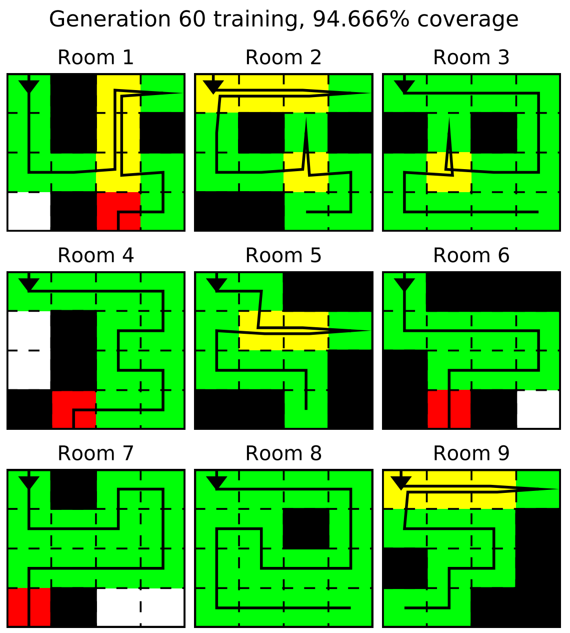
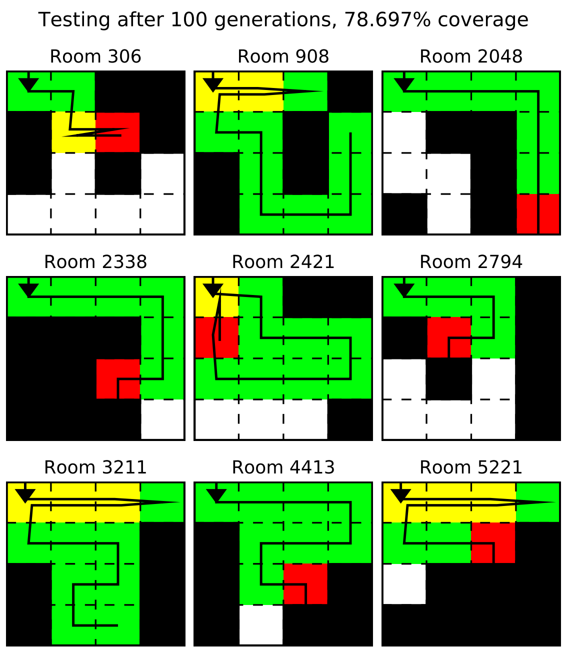


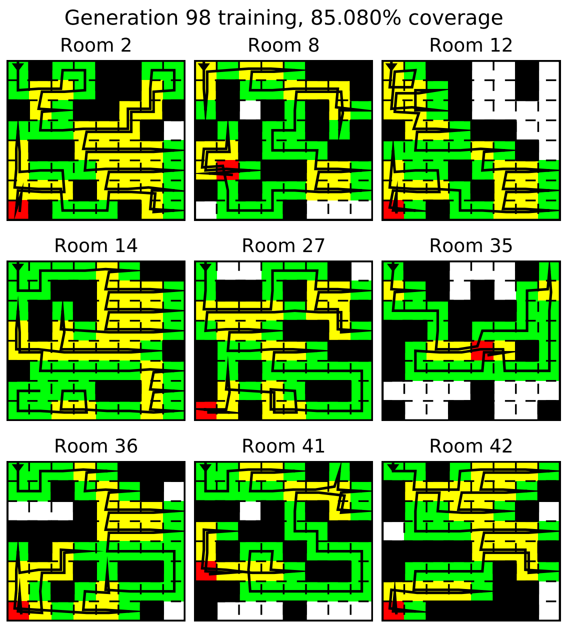
Figure 5.3: Comparison of average coverage per generation using 50 8×8 training rooms and 1,000 8×8 testing rooms. Solid lines are training performance, dashed lines are testing performance. lr is short for local radius.



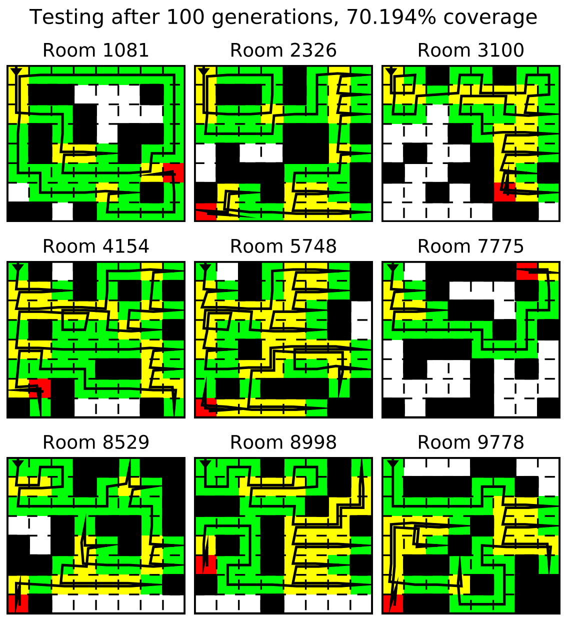
(a) Paths taken by the robot through the 9 randomly generated 4×4 training rooms.



(b) Paths taken through a random sample of 9 of the complete set of 4×4 testing rooms.



(c) Paths taken through a random sample of 9 of the 50 8×8 training rooms.



(d) Paths taken through a random sample of 9 of the 10,000 8×8 testing rooms.

Figure 5.4: Examples of paths taken by the best performing robot found within 100 generations in samples of the training and testing sets. We use the localised phenotype with a radius of one through 4×4 rooms (top) and 8×8 rooms (bottom). The robot begins in the top left corner and travels in the direction indicated by the arrow. Any cells visited exactly once are coloured green and cells visited more than once are coloured yellow. Cells coloured in red are the location where the robot made a mistake causing the end of the simulation.

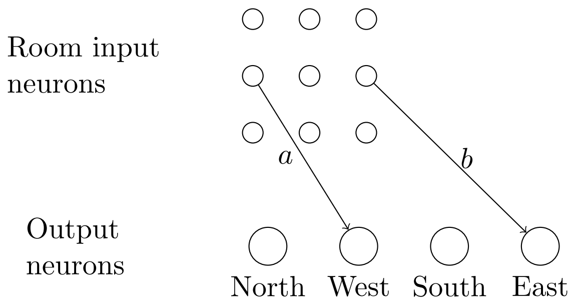


Figure 5.5: An example of a 3×3 localised network, we show only the weights between two input and output neurons (a and b) and the robot is at the centre of the input neurons.

5.5 Reduced Localised Phenotype

In the localised networks there is redundancy in the weights leading to the output neurons. An example of this is in Figure 5.5 where a and b must take a negative value to ensure that the robot will not go east or west if there is an obstacle in these cells as this would end the simulation. Different weights must be learnt for both cases but the logic for each output neuron is the same: do not activate if there is an obstacle in the proposed cell. From this example we realised that we can simplify the network by removing all but one output neuron and rotating the inputs to calculate the value of other three directions (e.g. if we keep the north output then the output of the network after rotating the inputs by 180° is the value of going south).

5.5.1 Method

We implemented this idea by adding constraints to the network such that the weights between a related input/output neuron pair are shared. In Figure 5.5 this means that $a = b$. This removes the need to rotate inputs and allows us to calculate network outputs as before. Applied to all input and output neurons this constraint reduces the size of the genotype by 75%.

5.5.2 Results

We compare the performance of the reduced phenotypes using several different local radii to our best localised phenotypes from the previous section (local radius of 1) in Figure 5.6. As before, we use 8×8 rooms and average across multiple trials.

5.5.3 Evaluation

The testing performance of the reduced localised networks increases as the local radius is increased up to four. A local radius of five or higher performs worse than the localised networks of the previous section. A local radius of four creates a 9×9 kernel which is large enough to take inputs from the majority of cells in an 8×8 room. The increase in performance up to a radius of four shows that the network is able make better informed decisions, instead of the blind obstacle avoidance behaviour of the robots in the previous section with a local radius of one.

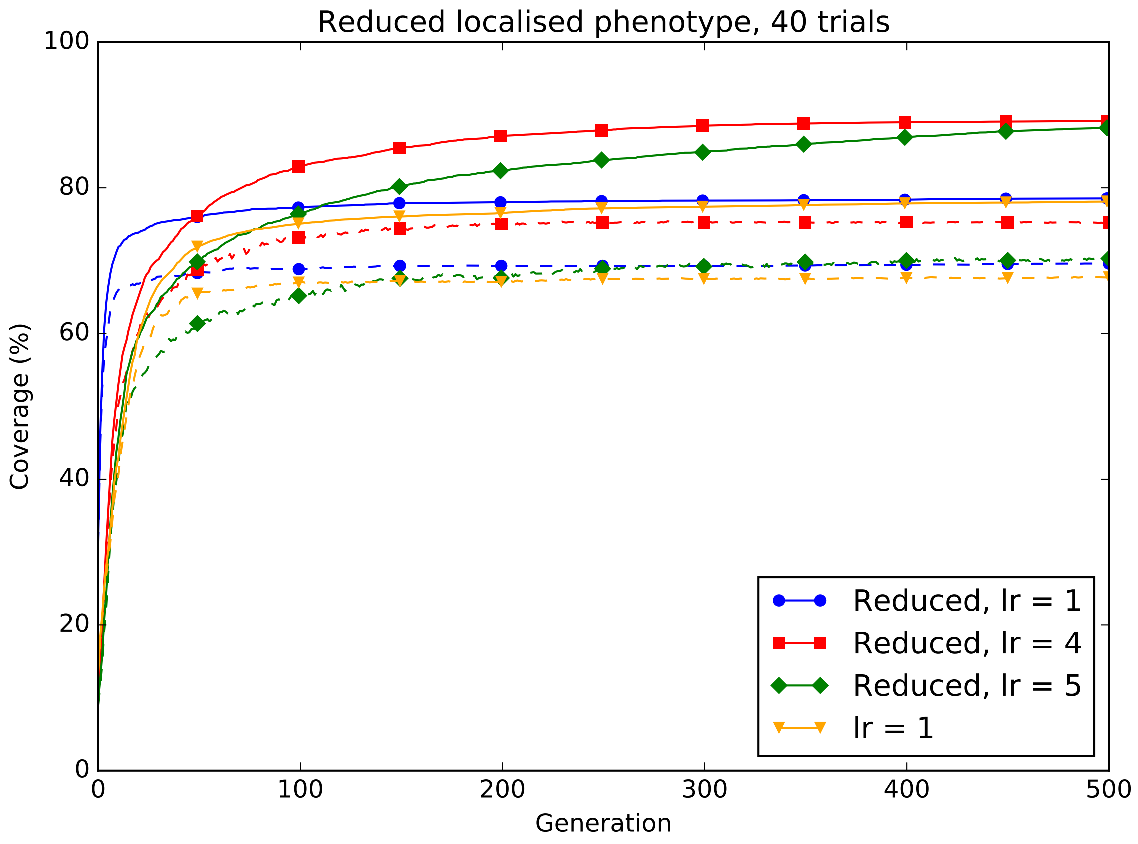


Figure 5.6: Comparison reduced localised networks and the localised networks of the previous section. We compare the average coverage per generation using 50 8×8 training rooms and 1,000 8×8 testing rooms. Solid lines are training performance, dashed lines are testing performance. lr is short for local radius.

5.6 Using a Subset of Training Rooms

Like most other genetic algorithm implementations our program spends the majority of the runtime in the fitness function, in our case room simulations account for 88% of the runtime. We had already improved the runtime by running the fitness function across different CPU cores and using third party natively implemented functions to reduce the time spent in a single simulation. To improve the runtime further we tried to reduce the number of simulations per generation without sacrificing performance on the testing set.

5.6.1 Method

Each generation, we pick a random subset of a predetermined size from the training set. The fitness of all individuals of the current population is found using this subset instead of the entire training set. Since a different subset is used in each generation, large enough subsets will ensure that the fitness function measures the general behaviour and not room specific paths.

Usually we stop the genetic algorithm before the maximum number of generations when one genotype with maximum fitness has been found because there is nothing more we can learn from the training set. When using a training subset we always run for the maximum number of generations as maximum fitness may be achieved by poor subset choice and the training subset of the next generation may show much worse performance.

5.6.2 Results

In Figure 5.7 we can see the increase in testing performance as the size of the training subset is increased. We used 8×8 rooms with a training size of 50, so the rightmost point on this graph is the testing performance achieved without using training subsets. After a subset size greater than or equal to six (mean = 0.73, SD = 0.041) the testing performance does not increase further compared to using the full training set (mean = 0.74, SD = 0.037); unequal variances t(77) = -0.94, p = 0.3486. When using a subset size of six the genetic search runs on average 5 times faster.

5.6.3 Evaluation

We did not use training subsets to generate any of the data used in this report, but the decreased runtime was very useful for testing new ideas.

5.7 Mutation Operator

As in the previous chapter we experimented with different mutation operators to improve the performance further.

5.7.1 Method

We implemented several different mutation operators and searched for the best hyperparameters for each operator.

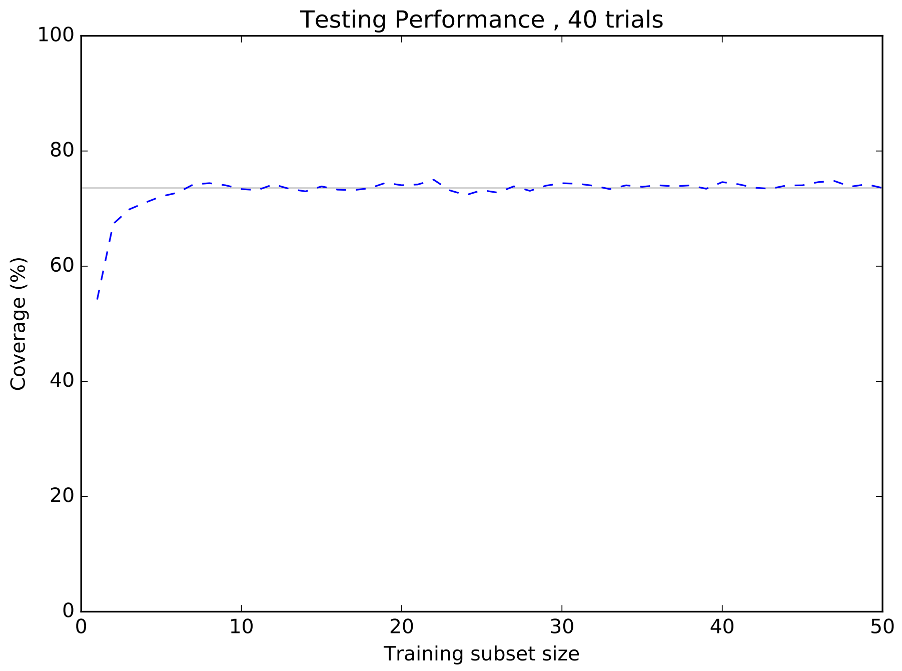


Figure 5.7: Testing performance as training subset size is varied. The average testing performance without subsets is shown as the horizontal line.

5.7.2 Results

The effect of our best operator, scale mutation is shown in Figure 5.8. This operator multiplies a gene by a random number between 0.5 and 1.5. We found that applying this operator to each gene (i.e. 100% mutation rate for this operator) gave the best results. The evolution on a path found using scale mutation can be seen in Figure 5.9.

5.8 Summary

We have extended the training method to handle multiple rooms and improved the performance of the system built in the previous phase (Chapter 4) from 10% to 79.7% in 8×8 testing rooms not present during training.

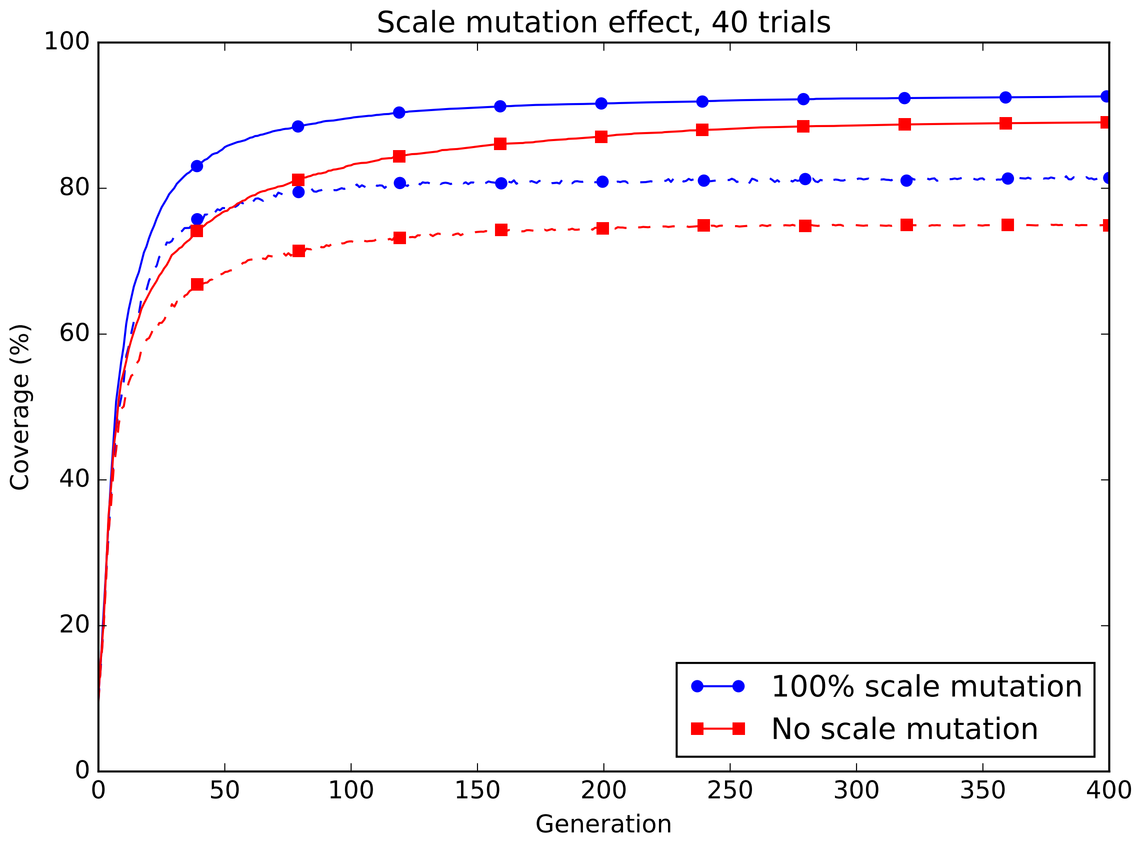
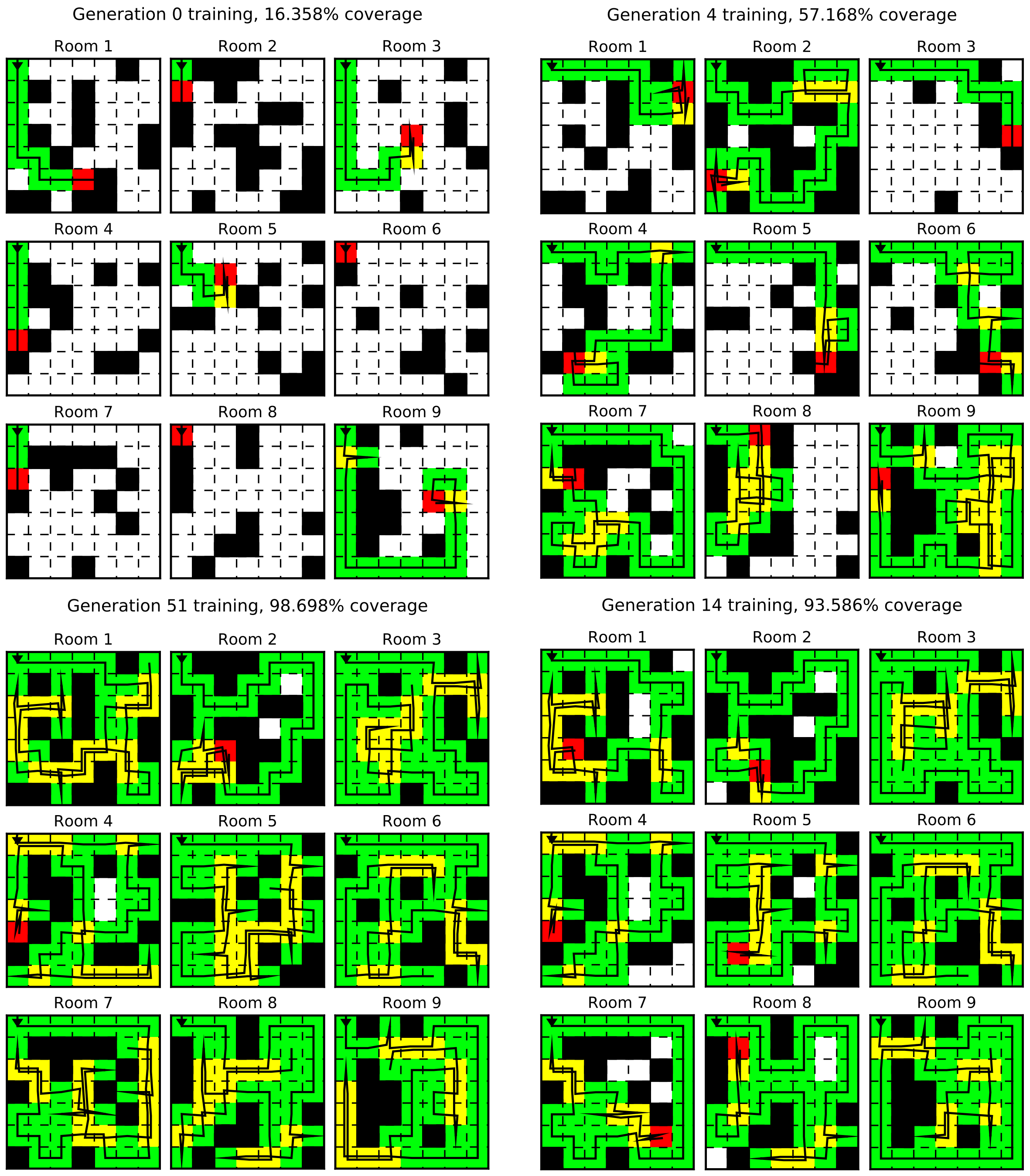


Figure 5.8: Solid lines are training performance in 50 8×8 rooms and dashing lines are testing performance in 1,000 8×8 testing rooms.



Generation 0 training, 16.358% coverage
Generation 4 training, 57.168% coverage
Generation 51 training, 98.698% coverage
Generation 14 training, 93.586% coverage
Room 1
Room 2
Room 3
Room 4
Room 5
Room 6
Room 7
Room 8
Room 9

Figure 5.9: Path evolution in 9 7×7 training rooms. Clockwise from the top right we can see the improvement in the quality of the training paths. Good performance is quickly achieved (e.g. generation 14) but increasing the coverage further is slower. The robot begins in the top left corner and travels in the direction indicated by the arrow. Any cells visited exactly once are coloured green and cells visited more than once are coloured yellow. Cells coloured in red are the location where the robot made a mistake causing the end of the simulation.

Chapter 6

Evaluation

In Chapters 4 and 5 we compared our performance to older versions of our training method for every improvement. In this chapter we will evaluate our training method by comparing the performance of the trained robots to robots with different navigation strategies.

6.1 Evaluation Robots

We implemented several alternative navigation strategies to compare against robots trained by our genetic algorithm.

6.1.1 TSP Bot

This robot converts a given room into a travelling salesman problem (TSP) and uses an existing solver to find an approximate solution which is converted back into a path through the room. The TSP solver used is an approximate solver to reduce runtime, therefore the robot will visit all cells but may take more simulation steps than necessary to so. We discuss this robot further in Section 6.2.

6.1.2 Random Bot

This robot picks a random direction at each simulation step until 100% coverage is reached. For this robot the simulation setup was modified such that a simulation will not end if the robot picks an invalid move (i.e. the robot bumps into the edge of the room or an obstacle). Instead, invalid moves will be ignored and the robot will not be moved from its current position during the simulation step. This is similar to randomised robots currently on the market such as the Roomba.

Random choice is the simplest navigation strategy which can, if given enough time, clean any room. The performance of this robot will provide a baseline which any robot should match.

6.1.3 Heuristic Bot

Implements a simple heuristic to navigate. If there is a cell next to the robot which has not been visited, the robot will go to that cell. Otherwise the robot will pick a random valid direction. This is repeated until 100% coverage is reached.

This heuristic is extremely simple to implement and will perform better than the random bot. Therefore the performance of this robot will provide a better baseline which other navigation strategies should be able to beat.

6.2 TSP Bot Method

6.2.1 Conversion to TSP

The robot's task is a variant of the travelling salesman problem where the start node is given, each node must be visited at least once and the end node is not required to be the start node. We convert this problem to a standard travelling salesman problem so we can compare our solution to an existing solver.

Step 1

We can express a room as an instance of this variant by creating a undirected graph with a node for each empty cell and connecting adjacent nodes with equal weighted edges. An example of this conversion is shown in Figure 6.1.

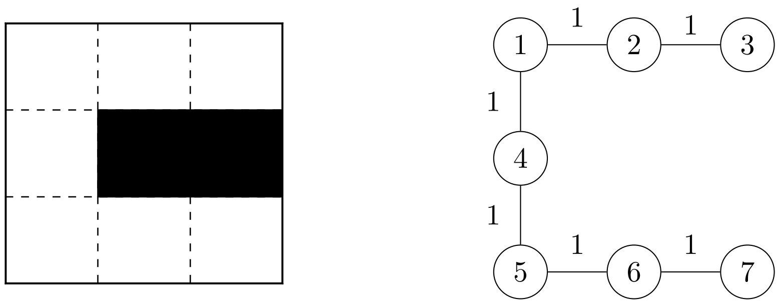


Figure 6.1: First step in TSP conversion. The room to convert is shown on the left and the initial graph on the right. We create one node per empty cell and connect adjacent nodes.

Step 2

We apply two transformations to the graph to create a TSP. First we add the requirement that each node must be visited exactly once. We convert the graph to a complete directed graph by adding the missing edges and setting the new weights to the cost of the shortest path between the nodes [20] (Figure 6.2). We also save the shortest path between all pairs of nodes, this is used when we convert the TSP solution to a path.

We initially used a fast implementation of the Floyd-Warshall algorithm from the `scipy` library to calculate the shortest paths with a time complexity of $O(V^3)$ for V nodes in the graph. The graphs we create are sparse as no node has more than four edges (average degree before conversion to a complete graph is 3.25), therefore we run Dijkstra's algorithm from each node in the graph for the lower time complexity of $O(V^2 \log V + VE)$, for V nodes and $E < 4V$ edges. This reduced the runtime of the shortest path calculation by 15%.

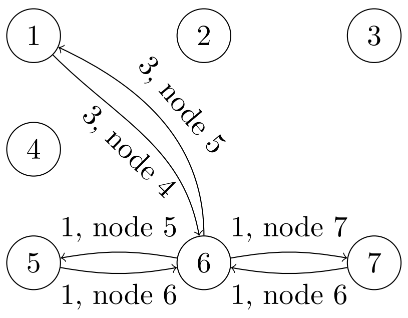


Figure 6.2: Second step in TSP conversion. We calculate the shortest distance (the first number on each edge) and which node to go to next (after the comma on each edge) for all pairs of nodes. We only include edges between node 6 and nodes 1, 5 and 7 for clarity.

Step 3

We now have a shortest Hamilton path problem. To complete the conversion to a travelling salesman problem, we remove the restriction on the start node and search for a cycle instead of a path. This is done by setting the weight of all edges leading to the start node to zero [20] (Figure 6.3).

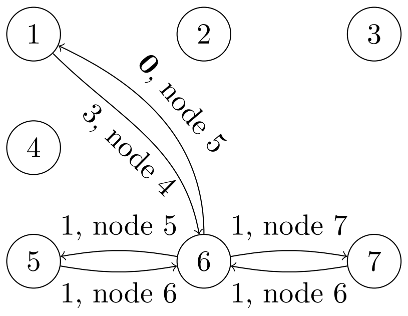


Figure 6.3: Third and final step in TSP conversion. The weights of all edges leading to the start node (node 1) are set to zero. As before, we only include a subset of the edges for clarity.

6.2.2 Solving

After the conversion each problem is saved as a TSPLIB [21] data file. This file is read by our chosen solver, LKH-2 [22] an implementation of the Lin-Kernighan heuristic [8]. Although this is an approximate solver the author has claimed to have reproduced optimal solutions for all solved problems he could find with this program [10]. This method should therefore provide a good estimate of the optimal path in all rooms.

6.2.3 Conversion to path

The output of the solver is the order in which the nodes of the graph should be visited. We convert this list to a path so we can compare the performance of this robot to other robots. Not all cells in node list are adjacent so we use the saved shortest paths to fill any gaps in the path, then we translate the node list into a list of (x, y) coordinates. The last node in this list is the starting cell as TSP solutions must be a cycle, but we can ignore this since any move to the starting cell is free by the construction of the problem.

For example, the solution to the travelling salesman problem in Figure 6.1 is the list of nodes beginning with [1, 3, 2, 4, ...]. Using the saved shortest paths we expand these to [1, 2, 3, 2, 1, 4, ...] and the coordinates [(0, 0), (0, 1), (0, 2), ...]. This is a path which can be followed by the robot without breaking the rules of our simulation by moving more than one cell per step, this is shown in Figure 6.4.

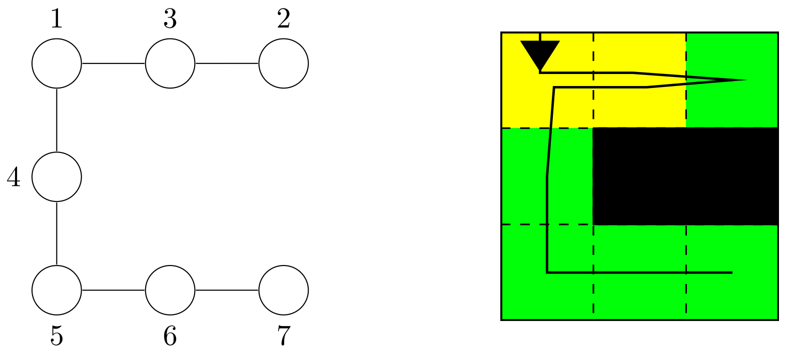


Figure 6.4: Path after TSP conversion. The solution to the TSP (left) is the order in which to visit nodes, this is shown outside of each node. The resulting path (right) is created by following the shortest paths to each node in the order specified by the TSP solution. In this example the TSP robot has found the optimal path.

Examples of the TSP bot solving large rooms is shown in Figure 6.5.

6.3 Experiment Setup

We trained our robots with 200 training rooms for 300 generations. We used 5×5 rooms with local radius of 2, 8×8 rooms with local radius of 4 and 15×15 rooms with local radius of 5. We then picked the robot with the highest coverage on the training set for each room size and ran this robot on 10,000 testing rooms of the same size, recording the path taken through each room. We also ran each of the evaluation robots described above in Section 6.1 through the same testing rooms, again recording all paths taken. From these paths we can calculate the average coverage over the testing set at each simulation step for each robot and for each room size. We stop calculating the average coverage when all simulations for a robot have ended, which will vary for each robot. We trained 10 robots for each room size and ran on different testing sets then averaged to results to reduce the randomness in the results. 10 trials is more than enough as every trial gave a line close to what is shown in Figure 6.5 and each trial would have led to the same conclusions.

We also recorded the average time spent in a room for the TSP solver and our trained robots for room sizes between 5×5 and 17×17.

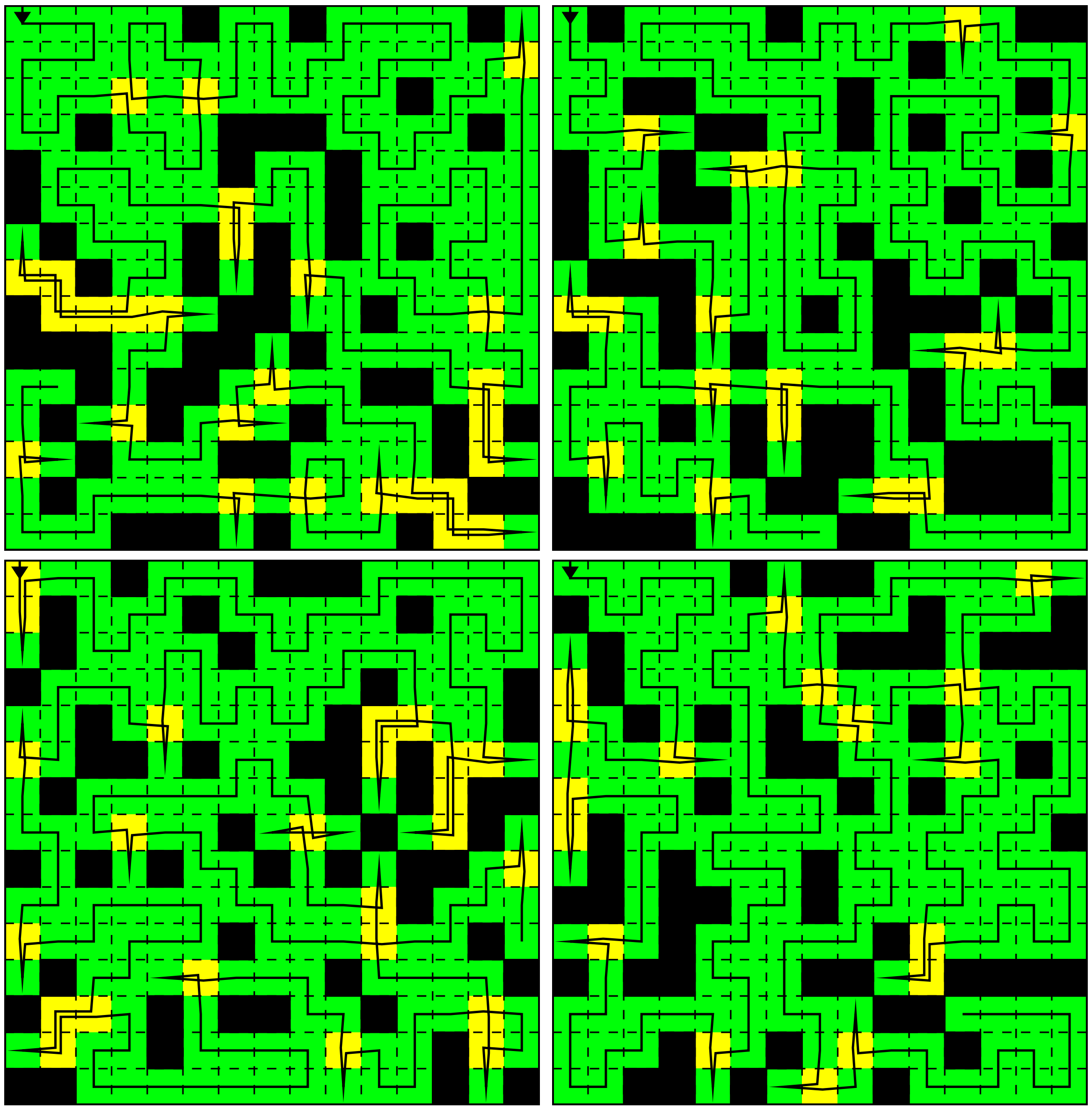


Figure 6.5: Paths through four 15×15 rooms found by the TSP bot. The robot begins in the top left corner and travels in the direction indicated by the arrow. Any cells visited exactly once are coloured green and cells visited more than once are coloured yellow.

6.4 Results

6.4.1 Coverage per step experiment

The results of this experiment are shown in Figure 6.5. We record how many steps each of the evaluation robots needed to achieve the maximum coverage of the robot trained with the genetic algorithm (Table 6.1). We refer to our trained robot as GA/NN in the graphs.

The randomised evaluation robots (Heuristic Bot and Random bot) will eventually cover any room by chance. The TSP robot is guaranteed to cover the room by design, as an approximate TSP solver will reach all nodes in the graph but may not find the optimal order in which to visit nodes. Only the robot trained with the genetic algorithm may become stuck in a loop or stop running after hitting an obstacle and is therefore the only robot which fails to reach 100% coverage in these tests. We can see this by the last data point on each graph, the TSP, Heuristic and Random bots end only when all rooms have been solved, we do not include end of the randomised robots as the x scale would hide the interesting comparisons between our robot and the TSP bot which happen within the first 100 steps. All robots begin on an empty cell so the coverage at the start of the simulation (step 0) is greater than 0%, this causes the non zero y intercept on each graph in Figure 6.5.

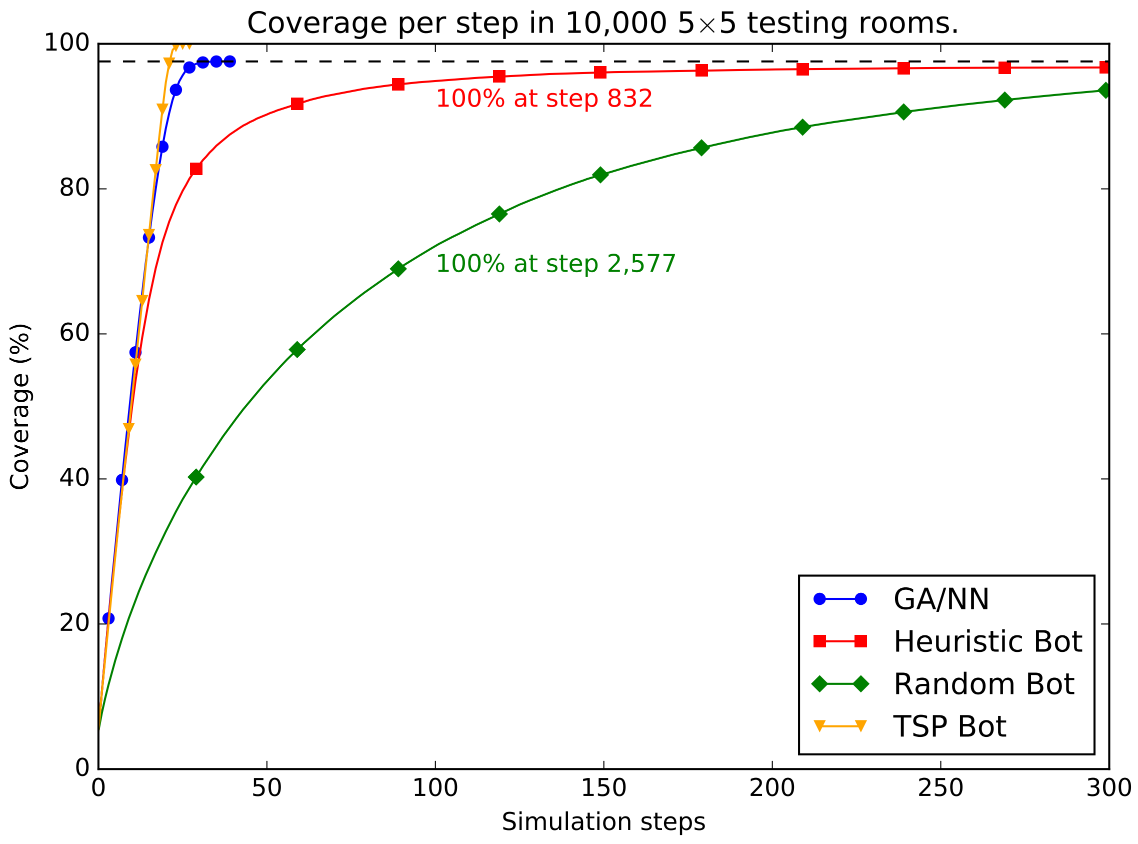
6.4.2 Runtime comparison

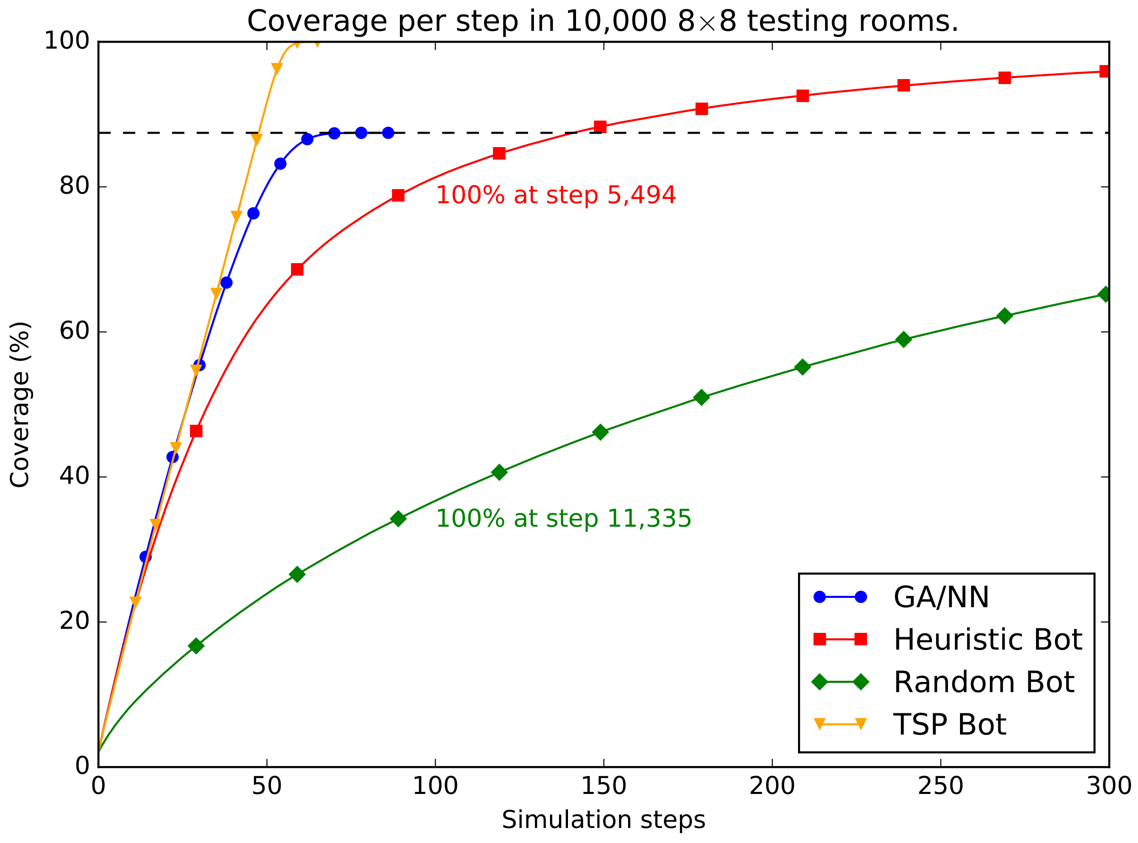
The result of the runtime comparison are shown in Figure 6.8. Here we can clearly see the quadratic time complexity of the LKH-2 solver ($O(n^{2.2})$). We increased the localised radius with the room size for a fair comparison which doubled the runtime of robots, from 0.48 milliseconds in 5×5 rooms to 1.9 milliseconds in 17×17 rooms. This difference in performance is not due to differences in the quality of the implementation. LKH-2 is written in C and has been optimised and improved over several years by an experienced programmer. Our neural networks are run with an optimised third party matrix multiplication function but the simulation (robot movement, network input calculation and valid move checks) which is included in the GA/NN runtime is written in Python and has not been optimised beyond running on more cores and more machines, both of which were disabled for this experiment. The difference is not caused by the conversion from and to the TSP representation either, the LKH-2 solver accounts for 99.4% of the time spent solving a 17×17 rooms. The remaining 0.6% is mainly due to IO latency from reading and writing the large problem files required to communicate with LKH-2.

6.4.3 Mostly empty rooms

During this experiment we noticed our robot performed much better in empty rooms. To investigate this we modified our room generator to place obstacles in each cell with a probability of 1% instead of sampling from the entire range of valid rooms. In 15×15 rooms this creates 2.25 obstacles per room on average. We trained and ran all robots through a set of 10,000 of these rooms to repeat these rooms. The average coverage per step is show in Figure 6.6.

The path taken by the robots is shown in Figure 6.7. The spiralling behaviour is not unique to this robot or these rooms, all robots we trained learnt to spiral inwards in empty rooms.


Coverage per step in 10,000 5×5 testing rooms.
Coverage (%)
Simulation steps
100% at step 832
100% at step 2,577
GA/NN
Heuristic Bot
Random Bot
TSP Bot


Coverage per step in 10,000 8×8 testing rooms.
Coverage (%)
Simulation steps
100% at step 5,494
100% at step 11,335
GA/NN
Heuristic Bot
Random Bot
TSP Bot

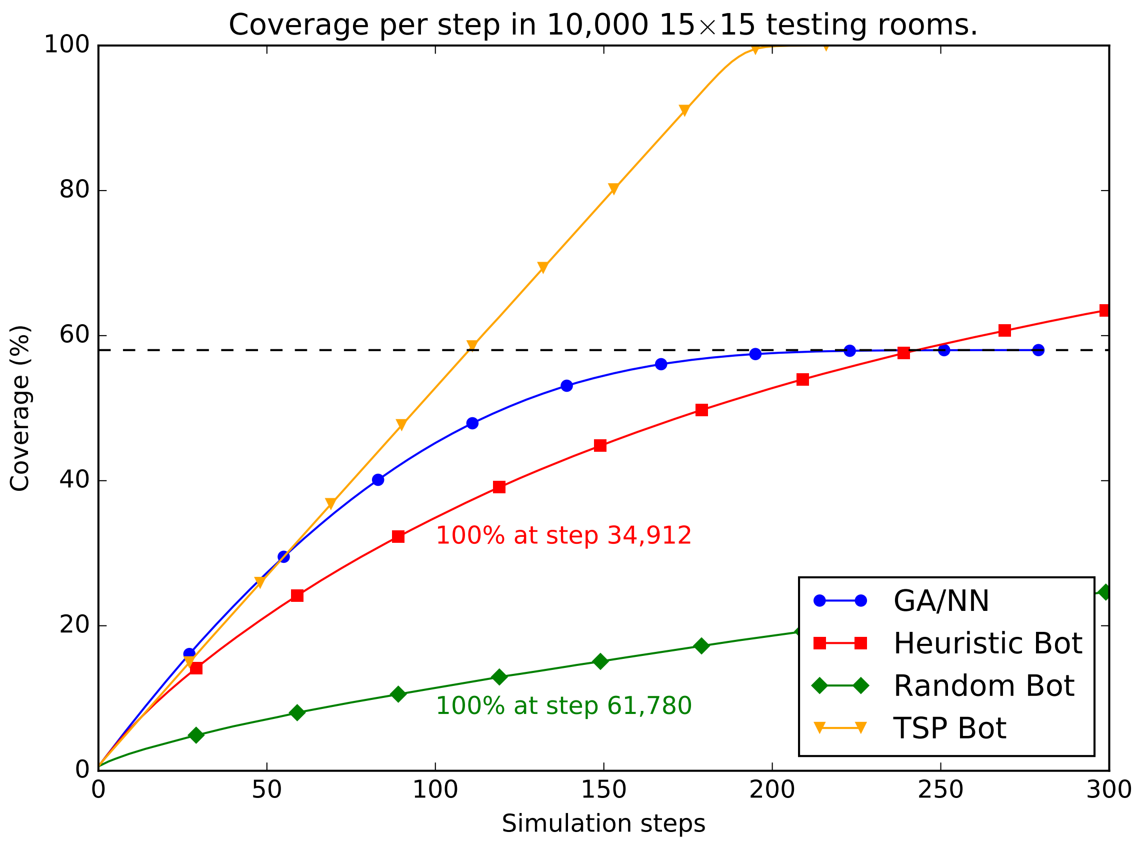


Figure 6.5: Average coverage per simulation step for different navigation strategies and room sizes.

Room Size	Max GA/NN Coverage	Steps to match max GA/NN coverage (% of GA/NN)			
		GA/NN	Random Bot	Heuristic Bot	TSP Bot
5×5	97.57%	39	831 (2077.50%)	475 (1187.50%)	21 (52.50%)
8×8	87.45%	81	749 (860.92%)	141 (162.07%)	48 (55.17%)
15×15	58.00%	276	1232 (440.00%)	243 (86.79%)	110 (39.29%)
Empty 15×15	98.49%	448	3180 (672.30%)	1020 (215.64%)	219 (46.3%)

Table 6.1: Steps taken to match GA/NN coverage.

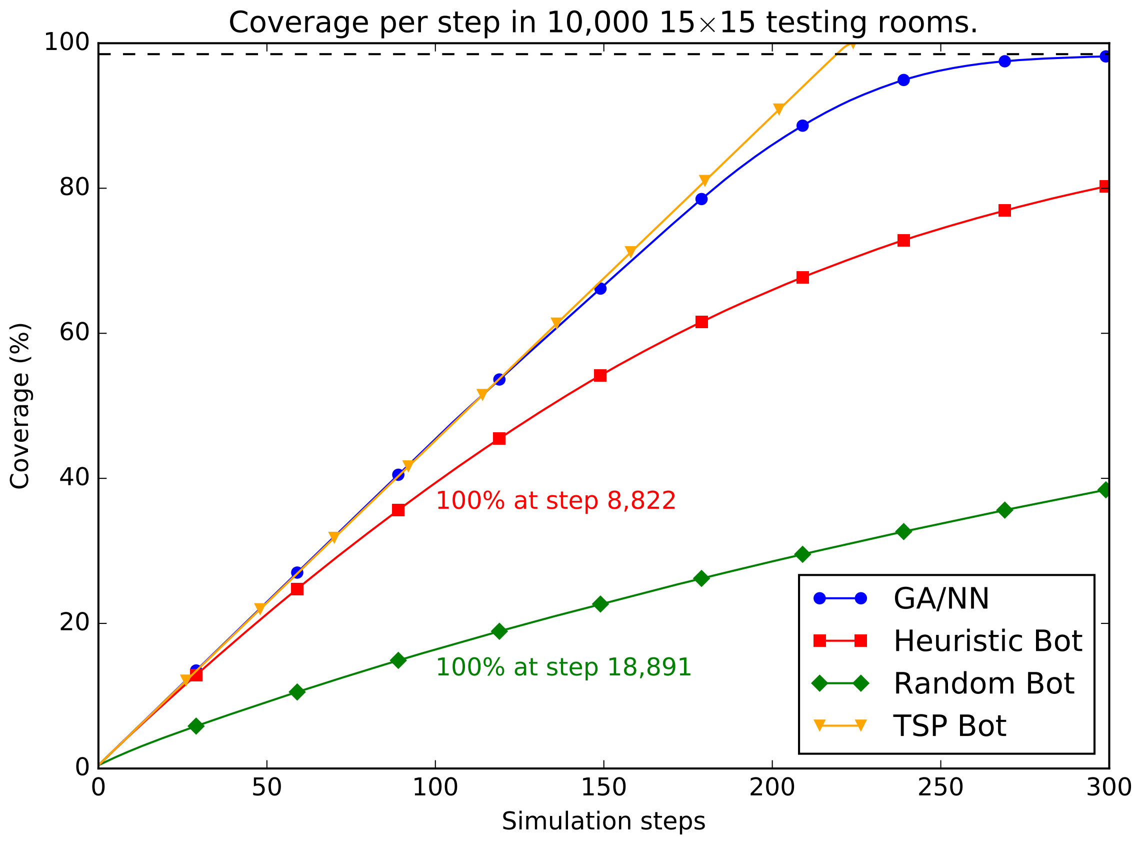


Figure 6.6: Average coverage per simulation step with 1% obstacle probability in 15×15 rooms, or an average of 2.25 obstacles per room.

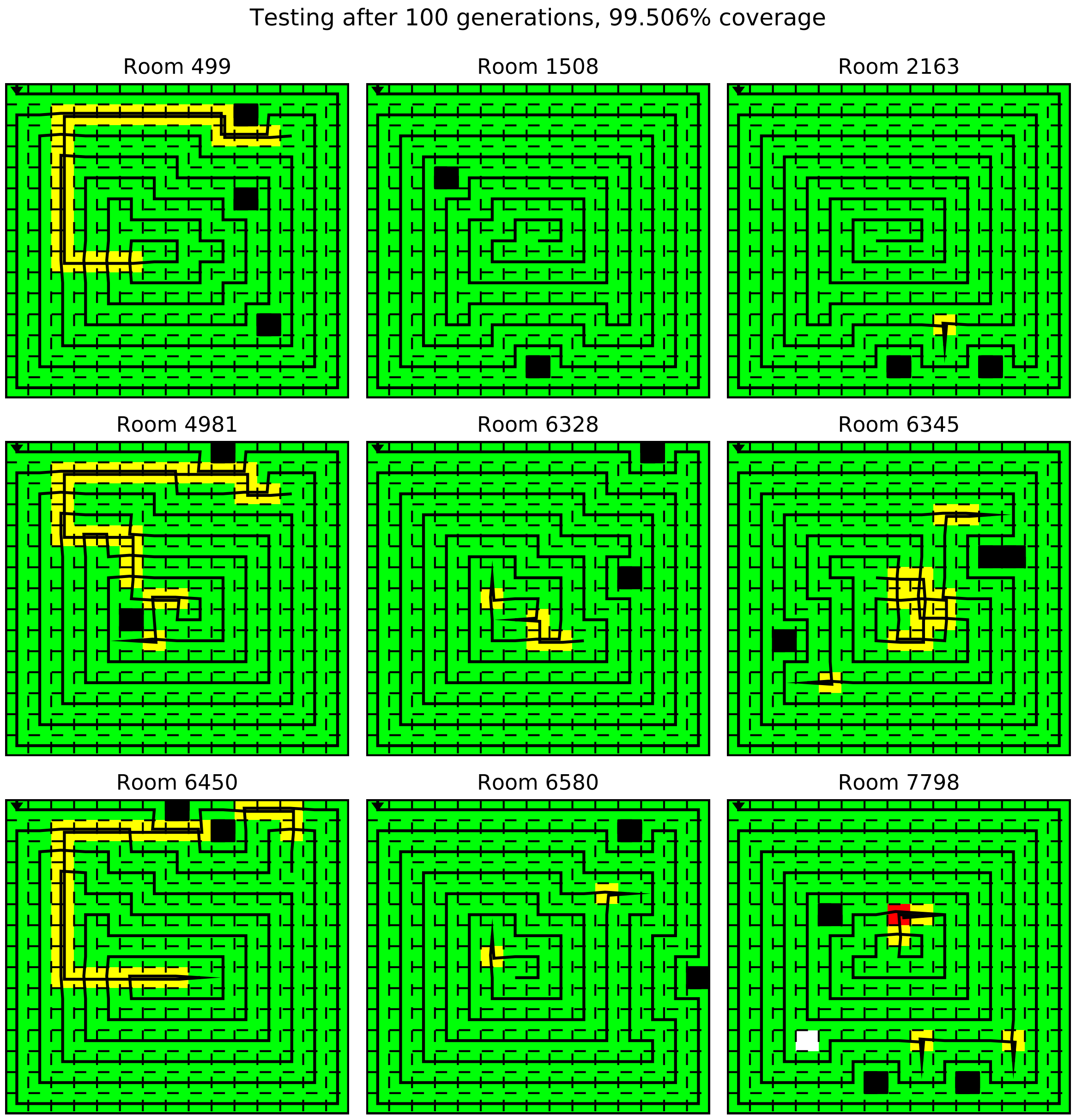


Figure 6.7: Path taken by our trained robot through a random sample of nine of the 10,000 empty 15×15 testing rooms. The robot begins in the top left corner and travels in the direction indicated by the arrow. Any cells visited exactly once are coloured green and cells visited more than once are coloured yellow. Cells coloured in red are the location where the robot made a mistake causing the end of the simulation.

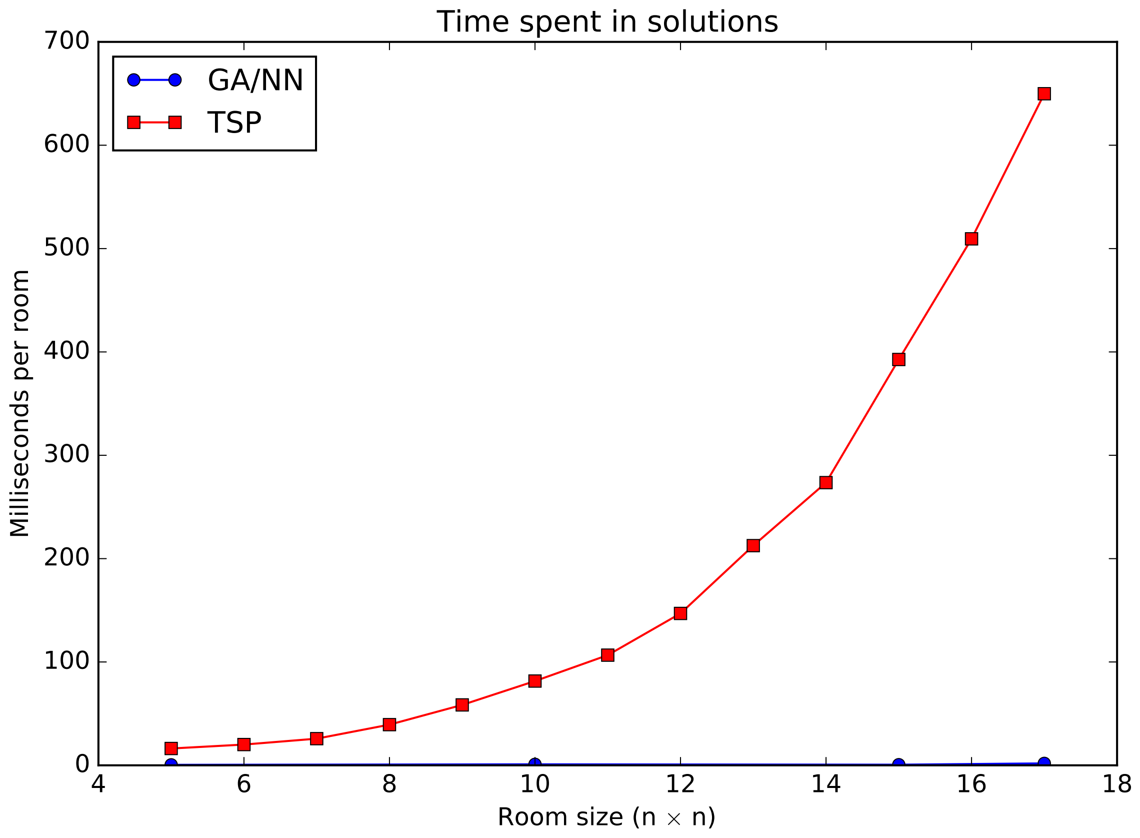


Figure 6.8: Change in runtime as room size is increased for the neural network and TSP bot. Averaged over 100 rooms for each room size. GA/NN times include the room simulation time, not just the time to run the neural network. GA/NN runtime doubles between 5×5 and 17×17.

6.5 Discussion

In large 15×15 rooms the effect of room size on the GA/NN robot is clearly visible and our robot can only outperform the Random robot. In these large rooms we are unable to train a robot which can use the large number of inputs effectively, despite using the reduced phenotype. To solve this, more work must be done on combining obstacle avoidance, which we can do well with the localised networks, with a network which can create a high level plan of the path to take, which we do not have. In these large rooms the robot often manages to clean some part of the room well but fails to move on to new areas causing low coverage (Figure 6.9). What is missing is another network or signal which can detect large empty areas of the room and guide the robot toward these areas. Another issue with large rooms, also visible in Figure 6.9 rooms 2,663 and 8,494 is the scattering of uncovered cells, a better planning method than the one implemented by our localised networks will systematically cover the room instead of constantly ignoring close by uncovered cells.

Empty rooms do not require high level planning as the spiralling behaviour is done by driving along the edge of obstacles or cells which have already been visited, this will visit all areas of a room uniformly.

In the small rooms (5×5 or less) there is no need for high level planning, as these rooms are too simple. The GA/NN robot performs well and reaches the majority of cells but the TSP robot achieves the same coverage almost twice as quickly (52.5% of the steps). However the GA/NN robot does not need information about the entire room, only the surrounding area. If applied to a real robot, this would allow the GA/NN robot to function without requiring any prior knowledge of the room, which usually created and stored by the robot during operation increasing the complexity of the development of the robot. Real rooms are unlikely to stay static and the GA/NN robot can adapt to changes in the room.

In the medium 8×8 rooms the maximum coverage of the GA/NN robot has dropped enough to leave large areas of the room uncovered. The Heuristic robot may be a better choice than the GA/NN robot for rooms of this size as it will eventually achieve high coverage, and requires only 1.6 times as many steps as the GA/NN robot to reach the maximum coverage of GA/NN (compared to 11 times as many steps in the 5×5 case). The Heuristic robot has the same advantages over the TSP robot as the GA/NN robot (i.e. no prior knowledge required and is not affected by changes in the room during operation).

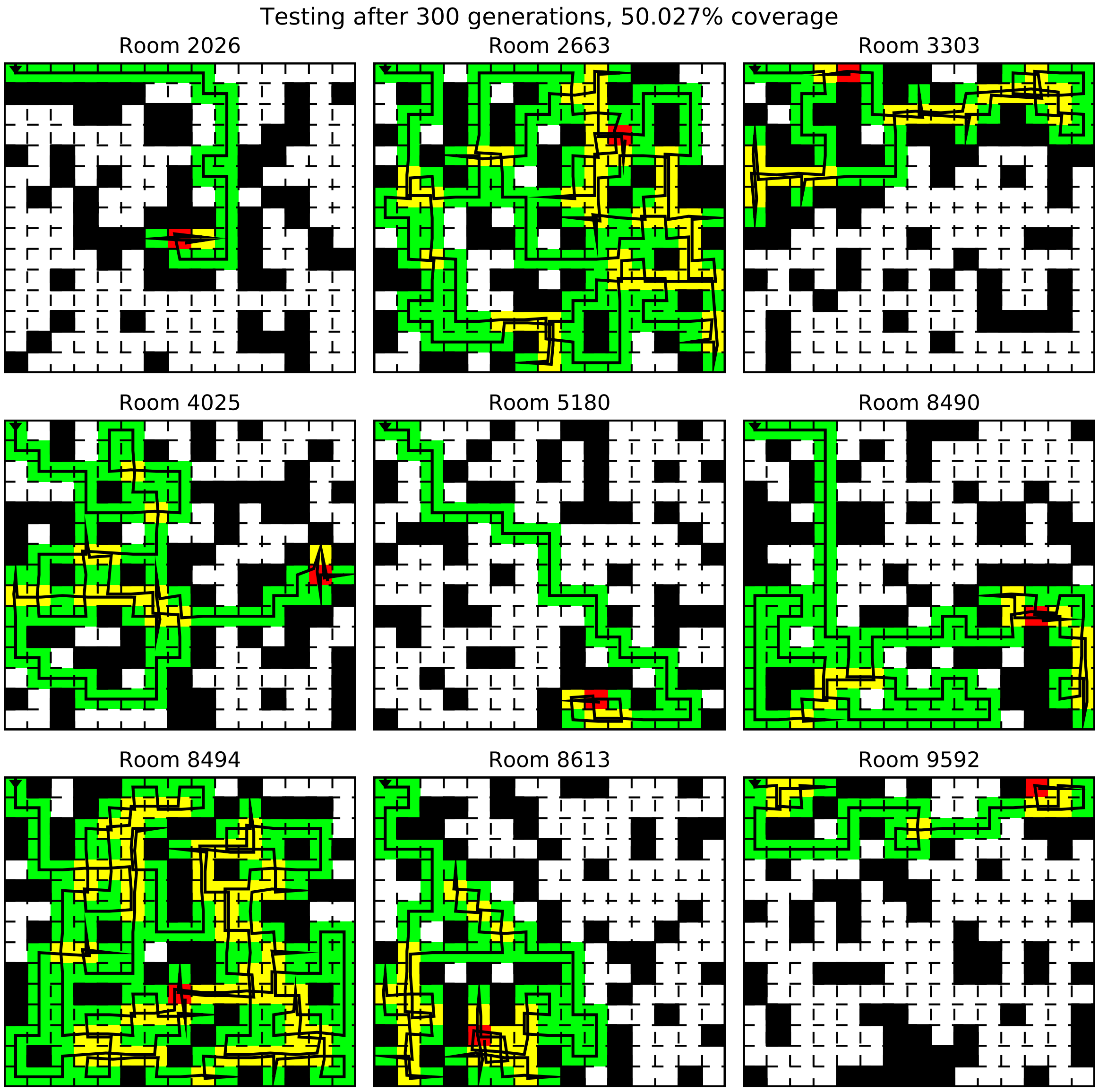


Figure 6.9: Typical example of cleaning through 15×15 testing rooms. The robot begins in the top left corner and travels in the direction indicated by the arrow. Any cells visited exactly once are coloured green and cells visited more than once are coloured yellow. Cells coloured in red are the location where the robot made a mistake causing the end of the simulation.

6.6 Summary

The main advantages of using a genetic algorithm to train an neural network to solve our navigation problem are:

- Achieves coverage performance comparable to the optimal (as approximated by the TSP bot) in small rooms and empty rooms.
- Good runtime performance which scales linearly with room size after training. As discussed earlier we only optimised when our experiments slowed down our work, optimisation was never a goal of this project. So there is potential for better performance if required.

While the main disadvantages are:

- Poor performance in large complex rooms.
- Never achieves perfect coverage in these testing sets.

Chapter 7

Conclusion

We have shown that, for our application, genetic algorithms can be used to train a neural network with close to optimal performance in certain rooms. In small rooms (i.e. 5×5 rooms) robots created with our method find a path close to the optimum, although these rooms were not present during training and the robot does not receive information about the entire room. The robots can be trained on 50 5×5 rooms and achieve 95% coverage on 100,000 different 5×5 rooms. These robots can be trained in under a minute on a conventional multi core CPU and are fast to run once trained. The average proportion of the room covered by the robot drops sharply as the size of the room is increased. The exception is large empty rooms where the genetic algorithm unexpectedly evolves an inward spiralling behaviour which, although not optimal, is effective at cleaning these rooms.

An oft quoted disadvantage of genetic algorithms is the large number of hyperparameters which must be set. However, despite our hyperparameter sweep experiments to find the optimum value for each hyperparameter we gained little performance compared to picking reasonable values for each (e.g. 1% - 5% mutation rate, 60% - 100% crossover rate). If we repeated this project for another application we would not spend time setting up and conducting these sweep experiments. The only possible exception is the mutation rate where, for this application, we found a sharper peak in performance around the optimal value than for other hyperparameters. We could therefore try several mutation rates in a small range such as 1% - 10%. This is true for both the canonical genetic algorithm and the CoSyNE algorithm, if using a different variant of a genetic algorithm we would consider conducting the same parameter sweeps as these algorithms may be more sensitive to hyperparameter values.

The disconnect between the genotype (the vector optimised by the genetic algorithm) and phenotype (the neural network built from the genotype) is an especially useful advantage as we can easily set constraints in the network such that some weights must always be equal without modification to the algorithm. Genetic algorithms make no assumption on the problem which allows complete freedom in building the phenotype.

The choice of inputs is important as the genetic algorithm was not able to train a network which can give more importance to certain inputs according to the location of the robot. In our project we saw that the network did not base its outputs on the inputs of the cells closest to the current position. Instead the signals from the cells near the robot were ignored due to the far more numerous signals from all other cells in the room, this caused the robot to end a testing simulation frequently (99.3% of cases) by colliding with an obstacle. This mistake should be easy to avoid as we can tell if a move is valid from a single input per position. We achieved the single largest boost in performance by restricting network inputs to the most important inputs to address this problem, increasing the coverage in 8×8 testing rooms from an average of 9.5% to 68%. We would spend more time experimenting with different methods of presenting the inputs

7.1 Future Work

Instead of continuing to improve the coverage in the multiple room case, given more time we would take the project further with the following ideas:

- **Other Applications**: A limitation is the single application to test genetic algorithms and neural networks instead of testing on a wide range of real world problems. Given more time, we would build a single system which could train different neural networks for each problem.

- **Recurrent Neural Network**: An interesting idea to explore is to try recurrent neural networks to store the state of the robot instead of providing the state as additional inputs to the network. This would further highlight the advantage of the disconnect between the genotype and phenotype as we can train an arbitrary complex network with no modification to the genetic algorithm.

- **Dynamic Rooms**: As discussed in the evaluation, assuming rooms are static during operation is unrealistic as people and pets may move around the room during operation. We would create obstacles which have some probability of moving at each simulation step and occasionally create and remove obstacles to test whether our robots could adapt to dynamic environments.

- **Supervised Learning**: We would train a network to mimic the paths taken by the travelling salesman robot (TSP robot). During training we present the network with inputs describing the state of the room and use the output of the TSP robot as the expected result. Backpropagation or another supervised learning algorithm could be used for training. This may give better performance on large rooms than the our reinforcement learning approach and if successful we would have combined the performance of the TSP robot with the adaptability of our neural network.

- **Connected Rooms**: We would generate grids which are more similar to the rooms in a house with several different rectangular rooms connected together and obstacles randomly strewn throughout.

- **Continuous Rooms**: The discretized rooms used throughout this project are a convenient representation but lack realism. Grid rooms can represent any room to arbitrary precision by increasing the width and height of the room, however the area under the robot which is cleaned during operation becomes unrealistically small. Furthermore, we restricted the robot to four directions. A future project could address these issues with a continuous room representation in which obstacles and the area cleaned under the robot are polygons. The inputs to the network could come from simulating a laser distance sensor spinning on the top of the robot similar to the Neato, which continually measures the distance to the nearest obstacles at all angles around the robot.

Bibliography

[1] Dan Cireşan, Ueli Meier, Jonathan Masci, and Jürgen Schmidhuber. A committee of neural networks for traffic sign classification. In *Neural Networks (IJCNN), The 2011 International Joint Conference on*, pages 1918–1921. IEEE, 2011.

[2] Dan Cireşan, Ueli Meier, Jonathan Masci, and Jürgen Schmidhuber. Multi-column deep neural network for traffic sign classification. *Neural Networks*, 32:333–338, 2012.

[3] Kelvin Xu, Jimmy Ba, Ryan Kiros, Aaron Courville, Ruslan Salakhutdinov, Richard Zemel, and Yoshua Bengio. Show, attend and tell: Neural image caption generation with visual attention. *arXiv preprint arXiv:1502.03044*, 2015.

[4] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *Signal Processing Magazine, IEEE*, 29(6):82–97, 2012.

[5] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[6] Genetic algorithm. `http://uk.mathworks.com/discovery/genetic-algorithm.html`. Accessed: 2016-06-11.

[7] Richard Bellman. Dynamic programming treatment of the travelling salesman problem. *Journal of the ACM (JACM)*, 9(1):61–63, 1962.

[8] Shen Lin and Brian W Kernighan. An effective heuristic algorithm for the traveling-salesman problem. *Operations research*, 21(2):498–516, 1973.

[9] Lkh results for soler's atsp instances. `http://webhotel4.ruc.dk/~keld/research/LKH/Soler_results.html`. Accessed: 2016-06-10.

[10] Keld Helsgaun. An effective implementation of the lin–kernighan traveling salesman heuristic. *European Journal of Operational Research*, 126(1):106–130, 2000.

[11] Simon Haykin. Neural networks, a comprehensive foundation. *Neural Networks*, 2(2004), 2004.

[12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

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

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

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

[16] Faustino Gomez, Jürgen Schmidhuber, and Risto Miikkulainen. Accelerated neural evolution through cooperatively coevolved synapses. *The Journal of Machine Learning Research*, 9:937–965, 2008.

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

[18] Dario Floreano, Peter Dürr, and Claudio Mattiussi. Neuroevolution: from architectures to learning. *Evolutionary Intelligence*, 1(1):47–62, 2008.

[19] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction.* MIT press, 1998.

[20] Der-San Chen, Robert G Batson, and Yu Dang. *Applied integer programming: modeling and solution.* John Wiley & Sons, 2010.

[21] Gerhard Reinelt. Tsplib95. *Interdisziplinäres Zentrum für Wissenschaftliches Rechnen (IWR), Heidelberg*, 1995.

[22] Keld Helsgaun. General k-opt submoves for the lin–kernighan tsp heuristic. *Mathematical Programming Computation*, 1(2-3):119–163, 2009.