Delft University of Technology
Master's Thesis in Embedded Systems

Evolving State Machines as Robot Controllers

Matthijs den Toom

TU Delft Delft University of Technology

Evolving State Machines as Robot Controllers

Master's Thesis in Embedded Systems

Embedded Software Section
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4, 2628 CD Delft, The Netherlands

Matthijs den Toom
m.denToom@student.tudelft.nl

14th August 2019

Author
Matthijs den Toom (m.denToom@student.tudelft.nl)
Title
Evolving State Machines as Robot Controllers
MSc presentation
23th August 2019

Graduation Committee

prof. dr. Koen Langendoen	Delft University of Technology
dr. ir. Chris Verhoeven	Delft University of Technology
dr. Mitra Nasri	Delft University of Technology

Abstract

Automated generation of robot controllers using an Evolutionary Algorithm (EA) has received increasing attention in the last years as it has the potential for a reduction in the development time of a robot. Often these EAs generate Neural Networks (NNs) as robot controllers. Using a NN for automatically generating robot controllers has two important downsides: 1.) A human is not able to fully understand the inner working of a multi-layer NN, and 2.) a NN has only limited abilities to decompose a complex task into sub tasks.

Both of these downsides can be addressed by using a State Machine (SM) instead of a NN as robot controller. Therefore, this thesis introduces an EA called Evolving State Machines As Controllers (ESMAC). ESMAC generates SMs instead of NNs. A SM is understandable for humans because of its modularity and allows for task decomposition by using a state for each sub task. Furthermore, two extensions of ESMAC are proposed: adaptive ESMAC and selector ESMAC. Adaptive ESMAC aims to automatically determines the number of states with which the best fitness for a task can be achieved. Selector ESMAC replaces the transitions that are used in a SM to switch between states with a NN-based switching mechanism. This switching mechanism allows mutations to make more gradual changes to a SM's behaviours, which improves the performance of the EA.

The performance of ESMAC is evaluated on two robotic tasks: *come-and-go* and *phototaxis-with-obstacles*. All three variants of ESMAC show equally good performance as a NN-based EA on the evaluated tasks. The controllers generated with standard ESMAC and adaptive ESMAC hardly make any state transitions and mainly use one state. However, controllers that do use multiple states appear to be more robust to changing scenarios and in noisy environments. Selector ESMAC is able to generate SMs-based controllers that have complementing states and, therefore, shows potential for decomposing a task into sub tasks.

Preface

I have always been fascinated by autonomous robots. In my opinion it would be amazing if robots, just like animals, would cooperate with humans and assist them in their jobs. However, a human is only willing to cooperate with a robot if it can be trusted. An easy way to build trust is to make a robot's behaviour understandable or at least predictable. After dropping my initial thesis topic on creating a real-life swarm, the understandability of robot controllers caught my interest. Advanced robots often use neural networks for control because of their ability to learn a complex task. However, the inner workings of those networks are hard to understand for humans. Therefore, I wondered: is it possible to design a robot controller model that is still able to learn complex tasks and whose inner working is understandable for humans? This question was the beginning of this thesis project. It turns out that this simplification is very hard, but what did I know.

This thesis would never be finished without the support of many to whom I owe a lot of gratitude. Foremost, I would like to thank Koen for being my supervisor, for keeping me on track when I drifted away, and for all the positivity and humor he brought during this process. This thesis would never have been finished without him. I would also like to thank everyone at the Zebro team for allowing me to join their team. In particular, I would like to thank Mattijs, Jeffrey, Frank, Laurens, Thijs, and Charu who where always available for discussion and from whom I learned many things on a wide range of topics. Thanks to Chris for always being a source of inspiration and thought. I am grateful to Mitra for being part of my graduation committee. Many thanks go to parents and siblings for allowing me to stay with them and for cheering me up when the thesis made me sad. I would like to thank my friends, Arian, Johan and Cees and many others, for showing me that there are other important things in life besides a thesis. Last, I would like to thank God for giving me the strength and abilities to study this topic.

Matthijs den Toom

Delft, The Netherlands
14th August 2019

Contents

Chapter 1

Introduction

Manually designing robot controllers is hard and time consuming. The Zebro team [21] of Delft University of Technology experienced this first hand when they were designing robot controllers for their six-legged robot. The aim of the Zebro team is to design a fully autonomous swarm of walking robots that is able to monitor an area. This requires a robot controller that defines how each robot behaves by mapping the sensor inputs to actuator outputs. The time-consuming process of designing a robot controller that allows the robot to walk and interact with neighbouring robots slows down progress in achieving this goal. Therefore, the Zebro team is interested in exploring methods that speed up a controller's design process. One of these methods is to automatically generate a controller using machine learning.

Automated generation of robot controllers using machine learning has received increasing attention over the past years. One of the reasons for this attention is the potential for a reduction in the development time of robots. Furthermore, by using the right tools complex controllers can be generated without human ingenuity. Traditional machine learning tools, such as back propagation for training a neural network, are not suitable for generating robot controllers because they require training data. Training data, such as realistic sensor inputs, can only be gathered when the robot is operational, but making the robot operational is the goal of the learning.

Therefore, artificially designing a robot controller is usually done using one of two methods: a reinforcement learning algorithm or an Evolutionary Algorithm (EA). In reinforcement learning a robot learns to improve its controller by receiving feedback on its performance from the environment or from designers. Based on this feedback the robot changes its controller.

EAs utilize Charles Darwin's theory of natural selection to develop robot controllers. Darwin observed that animals adapt to their environment and he attributed this to, what later has been called, 'survival of the fittest'. In his theory, Darwin stated that random mutations in an individual's gene lead to diversity in a population (a group of animals of the same species).

Because of this diversity some individuals are able to generate more offspring than others. Therefore, the genes of those individuals are more dominant in the next generation. Over many generations this means that the genes of the fittest individuals survive and those of the weak vanish. This idea can easily be used for automated generation of robot controllers: evaluate many controllers and ensure that good performing ones are slightly changed (mutated) and live on.

EAs are applied to robotics in a field that is called evolutionary robotics [19]. Controllers for many standard robotic tasks have been generated using EAs. Among others, this includes the phototaxis task (moving to a light) [4], maze navigation [9], and making a legged robot walk [5]. The use of EAs in robotics, however, is not limited to single robots. EAs have also been successfully applied to multi-robot tasks, such as flocking [1], aggregation [6], multi-agent monitoring [7], and creating patterns of tiles [30].

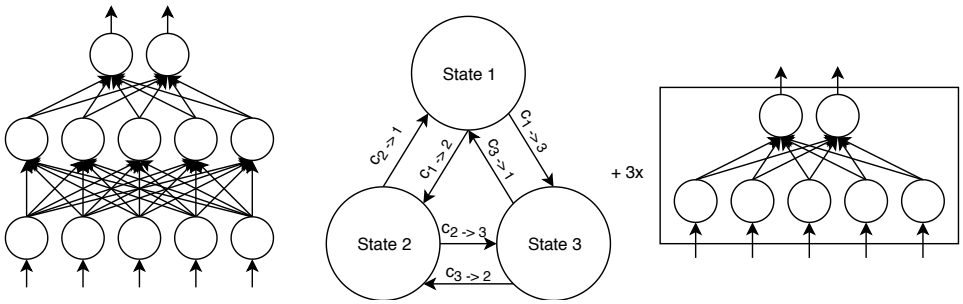
Currently, the type of controller evolved by an evolutionary algorithm usually is a Neural Network (NN) [11]. A NN maps the robot's sensor values to input signals for its actuators using a network of nodes [25]. The detailed working of a NN will be explained in Section 2.1.

1.1 Problem statement

Two problems appear when evolving a NN as a controller for a robotic task using an EA. First, solving complex robotic tasks requires an increase in the number of nodes in a NN, which reduces a human's ability to interpret and verify the working of the network [2]. This difficulty of interpretation is because a NN is a cascade of non-linear functions as will be explained in Section 2.1. Understanding robot controllers can be highly beneficial to engineers as it allows for identification of dangerous situations.

Second, for complex robotic tasks evolutionary algorithms encounter the so called *bootstap problem* [19]. The bootstrap problem means that the optimization problem becomes intractable in complex search spaces because it becomes too hard to rank the controllers and select the better ones. One of the causes of bootstrapping is that the task as a whole is too complex to solve and a decomposition into sub tasks is required. However, for complex tasks it is not always clear what this decomposition looks like and therefore automated decomposition would be beneficial.

Both these problems can be addressed by evolving a State Machine (SM) instead of a NN. A SM consists of states, which can all have different functions, and transitions that allow switching between states. Therefore, a SM naturally decomposes a task in sub tasks. Each state can consist of an understandable controller because the sub tasks are easier than the whole task. Furthermore, the transitions allow understanding of the relations between the sub tasks. Figure 1.1 aims to make intuitively clear why SMs are easier



(a) 2-layer NN (b) SM with three states

Figure 1.1: Example NN and SM with the same number of in- and outputs. The SM is more understandable because different component can be analyzed individually.

to understand than NNs. The relations between all nodes and connections and how they interact already becomes difficult to understand for a NN with one hidden layer, five inputs and two outputs. On the other hand, a SM with the same number of in- and outputs and three states is easier to understand because of its modularity. This will be more so for controllers with 24 sensor inputs as used in this thesis.

Therefore, in this thesis the possibility of evolving a SM instead of a NN for solving a robotic task is investigated. While other works [12, 13] limit the search space by defining a set of possible state behaviours, the aim of this thesis is to autonomously generate the behaviour of each state as well. This allows the creation of behaviours that were not imagined by the designers. This challenge results in a research question, which is formulated as follows:

> Is it possible to automatically generate state machines using an evolutionary algorithm that decompose a task in sub tasks and perform at least as good as a neural network?

1.2 Contributions

This thesis investigates whether a SM can be generated as a robot controller using an EA. The author's key contributions are:

1. Evolving State Machines As Controllers (ESMAC), an evolutionary algorithm that evolves a state machine that is able to solve robotic tasks.

2. Two variants on ESMAC: selector ESMAC, uses a different transition mechanism compared to standard ESMAC, and adaptive ESMAC, which determines the number of states while running the EA.

3. A comparison of the performance of ESMAC against the performance of NN controllers evolved using an EA on two robotic tasks: *come-and-go* and *phototaxis-with-obstacles*.

4. An analysis of the generated controllers in terms of state usage and robustness.

1.3 Thesis Outline

The content of this thesis is as follows: Chapter 2 introduces the sources of inspiration and the building blocks of this thesis. Chapter 3 introduces ESMAC, an EA that generates a state machine as a robot controller. Chapter 4 explains the experimental setup and the robotic tasks that are used for evaluation. Chapter 5 presents the findings of the evaluation of ESMAC on the selected tasks. The conclusions as well as future work are presented in Chapter 6.

Chapter 2

Related Work

This section introduces the building blocks required in this thesis. First, the general layout of a feed-forward neural network, which is often used in combination with evolutionary algorithms, is presented. Second, generation of robot controllers using evolutionary algorithms is explained. Last, research on evolving state machines is reviewed.

2.1 Neural Networks

A robot controller generated using machine learning or evolutionary algorithms is often a Neural Network (NN) [18]. Figure 2.1 shows a perceptron, the smallest NN. A perceptron consists of a set of connections, trough which inputs arrive, and a node that calculates the output. Each input i_i for $i \in \{1, ..., n\}$, where n is the number of inputs, arrives trough a connection that multiplies the input with a weight w_i. The weighted inputs arrive at the node where their weighted sum is calculated. Furthermore, A bias b is added to the weighted sum, which allows for a constant offset.

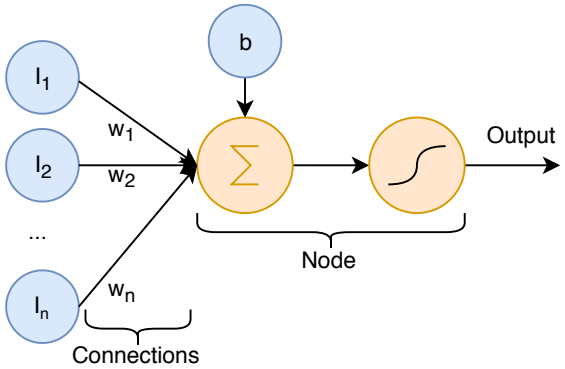


Figure 2.1: Layout of a perceptron.

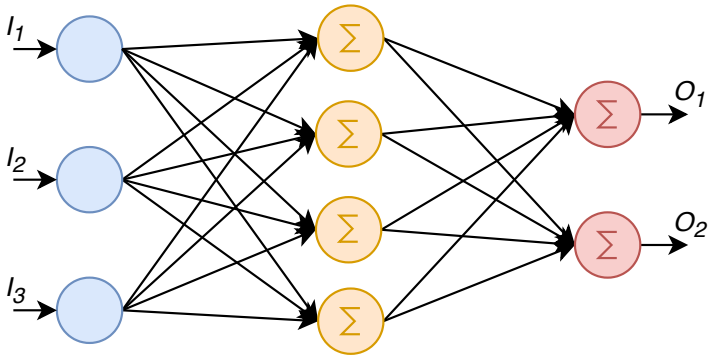


Figure 2.2: Example of a neural network. The input nodes are blue, hidden nodes are orange, and output nodes are red.

The weighted sum is passed to an activation function, which determines the output of the perceptron. In this thesis, the activation function is defined by $g(x) = tanh(x)$, which maps any input in range $[-1, 1]$. In mathematical form a perceptron looks like:

$$o = g(b + \sum_{i=0}^{n} w_i i_i) \tag{2.1}$$

where o is the output of the perceptron, b the bias, w_i the weight corresponding to each input.

The output of the perceptron can be fed to an actuator. Perceptrons can also become nodes in a network when linked with other perceptrons by using the output of one as input to another. Perceptrons are then called neurons. A network of neurons without cycles is called a feedforward neural network, in the following of this thesis indicated as Neural Network (NN). Figure 2.2 shows an example of a multi-layer NN. Layers in a NN are groups of nodes with connections to the previous and next layer. Nodes in the first layer, the input layer, receive the actual inputs of the network. Nodes in the last layer, the output layer, output the actual outputs of the network. Nodes in a hidden layer, a in-between layer, receive inputs from the previous layer and send outputs to the next layer.

When the number of hidden layers increases, understanding how the input values lead to the output of the NN becomes harder. When a non-linear activation function such as the tanh function is used, a NN becomes a cascade of non-linear functions. This makes understanding and manually designing a NN hard. Therefore, NNs are often generated using machine learning or evolutionary algorithms, which automatically optimizes the weights and biases in the network. While automated optimization of NNs has proven to provide functional networks, explaining a network's output is hard because there are many parameters and many interactions between nodes.

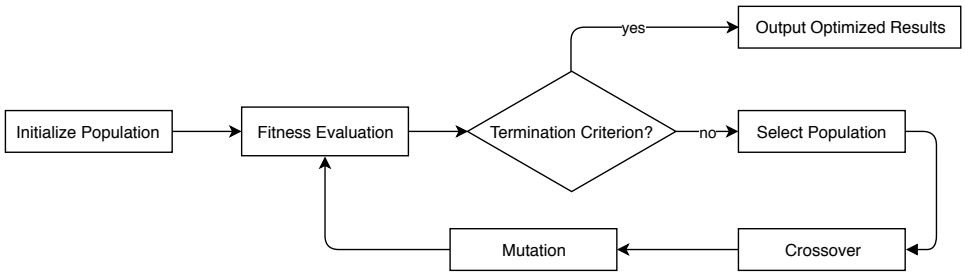


Figure 2.3: Diagram of a standard evolutionary algorithm.

2.2 Generating Robot Controllers using Evolutionary Algorithms

Evolutionary Algorithms (EAs) are a set of algorithms that use concepts from biological evolution to optimize the solution to a problem. EAs are used in many domains, for example in robotics [18] and bioinformatics [10]. When the performance on a robot task can be described by a fitness function, which indicates how well a robot controller executes the task, EAs can be applied to robotics. This method of generating robot controllers is part of a research field called evolutionary robotics [19].

The most-known EA is presented in Figure 2.3. This algorithm consists of the following steps:

1. Initialization: A population of n robot controllers is randomly generated, where n is the number of individuals in the population.
2. Evaluation: Each controller is evaluated with respect to a fitness function that describes the required behaviour.
3. Selection: The best scoring $p\%$ controllers in a generation are selected to seed the next generation of controllers, where p is user defined.
4. Crossover: In this step two controllers are randomly selected, which are then combined to generate a new controller. This is repeated until n new controllers are generated.
5. Mutation: The newly generated controllers are randomly mutated.
6. Repeat from step 2, until a specified number of generations has passed or a suitable solution has been found.

When the algorithm is finished, the best performing controller is returned to the user.

The set of controllers that is evaluated at any point in the algorithm is called a population. Each controller in this population is called an individual. Technically, a population consists of genomes instead of controllers. Genomes are encodings of controllers that can be mutated and which can be crossed over. In EA terminology, a genome is called a genotype and a decoded genome is called a phenotype. These terms are important when a

genome looks completely different than the corresponding controller. For example, grammatical evolution is an EA where the genotype is a binary string and the genotype is a context free grammar [20]. However, in ESMAC the genotype and the phenotype are almost the same.

Every iteration of the algorithm (from step 2 till 5) is called a generation. The fitness of the controllers is expected to improve with generations, because only the scoring individuals of the each generation are used for the next generation.

The standard algorithm presented here, which uses both crossover and mutation, is called a genetic algorithm [33]. In recent works, for example in [32], crossover is often skipped and the selected controllers are only mutated. This type of EA is called evolutionary computation. Elitism is another improvement to the evolutionary process that is often used. When Elitism is used, the elite, the k best performing individuals, are copied to the next generation without crossover or mutation. The size of the elite k is a user parameter. Elitism ensures that in a static environment the highest fitness can only improve, and cannot decrease because of a 'bad' crossover or mutation.

2.2.1 Neuroevolution

Optimizing a NN using an EA is called neuroevolution. Neuroevolution is often used in evolutionary robotics for controller generation, for example in [29], [31], and [32]. A robot controller that consists of a fixed topology NN is generated as follows using an EA: First, a population of n NNs is generated, where the weights and biases of each network are randomly initialized. After evaluation and selection, crossover is performed by cutting two individuals at the same spot and gluing the different halves together, creating two new individuals. Mutation randomly changes the weights and biases of the newly created individuals. In this thesis, numbers such as the weights and the biases can be mutated in two ways: a *replace mutation*, which replaces the number by a random number, or a *gaussian mutation*, which adds a random draw from a Gaussian distribution to the number.

A neuroevolution algorithm that is a source of inspiration for adaptive ESMAC is Neuroevolution of Augmenting Topologies (NEAT). NEAT evolves both the topology of the networks and their weights and biases at the same time. This allows for minimal but not too small networks. Special operators for mutation and crossover are introduced that can cope with different topologies while the EA runs. Furthermore, it introduces *speciation*, which is meant to protect newly evolved topologies. Speciation divides a population in species based on genetic difference, meaning similar individuals are put in the same species. The size of these species is determined by their fitness relative to other species. Adaptive ESMAC uses speciation to protect State Machines (SMs) that just extended their topology.

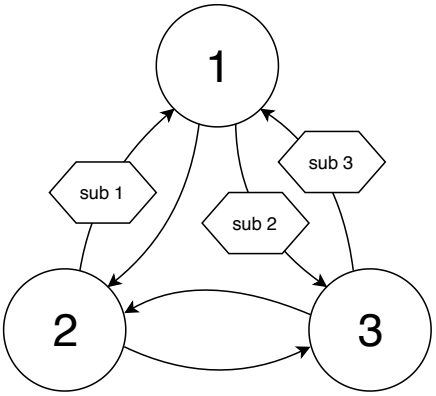


Figure 2.4: Sub modules within the main SM, as proposed in [3].

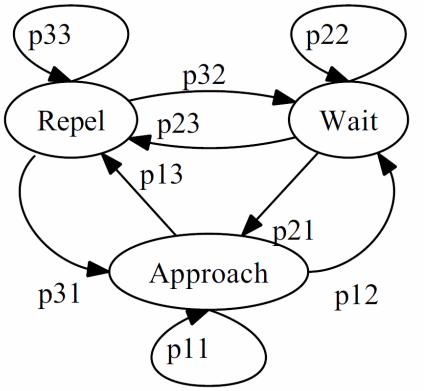


Figure 2.5: Layout of PFSM used for robot clustering in [15].

2.3 Evolving State Machines using an EA

As the goal of this thesis is to generate a SM using an EA, this section presents a review of the work on evolving SMs. A SM is an abstract machine that maps a set of inputs to a set of outputs. In case of a robot controller, a SM maps sensor inputs to actuator outputs. A SM consists of a set of states, one of which the robot is in at any moment in time, a set of transitions, and an action function that maps the current state and the inputs to an action, or output. Transitions allow the robot to change state based on the sensor inputs. In this thesis, Deterministic Finite State Machines (DFSMs) are used, which means that no random transitions are made and every SM consists of a finite number of states.

One of the first works on learning a DFSM using an EA is by Spears and Gordon [27] who aim to learn a resource-capturing game in which an agent is to capture more ground patches than a pre-programmed adversary. The SM representation used by Spears and Gordon cannot be used in this work as their SM's genome requires a small and discrete range of the inputs and can therefore not be used with continuous sensors.

An useful improvement to increase the performance of EAs that generate SMs is introduced by Chellapilla and Czarnecki [3]. In their work they evolve a SM with sub SMs as shown in Figure 2.4. In each generation either a sub module or the main SM is changed. This technique can be applied in our approach, where NNs are sub modules.

Automated generation of state machines as robot controllers is introduced in [22]. In this work, transitions are based on a condition in the form $\{senor_i\ [<, ==, >]\ n\}$, where $sensor_i$ is a sensor input and n is a number. ESMAC uses this condition representation. Contrary to the approach in this thesis, basic behaviours are designed manually and not evolved with

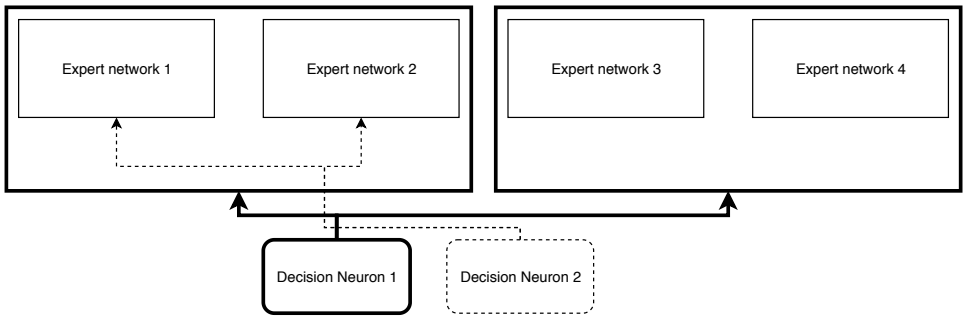


Figure 2.6: Example of an ETDN network [29]. The expert networks are NNs that can control a robot. The decision neurons determine which expert network is used based on the sensor input.

the SM. Furthermore, ESMAC does not use incremental learning, which means that the difficulty of the tasks increases as the algorithm progresses.

AutoMoDe [12, 13] is an initiative towards general-purpose automated design methods and focuses on generating a SM-based controller for swarm robotic tasks. In AutoMoDe the possible transitions and the states (behaviours) are already implemented and the EA is cannot create new transitions or behaviours. In a similar work, a Probabilistic Finite State Machine (PFSM), which runs on every robot, is generated using Particle Swarm Optimisation (PSO) [16]. Only the transition probabilities and the predefined behaviour that each state executes are evolved. In this thesis, both the topology of the SM and the basic behaviours of the robots are evolved.

A source of inspiration for selector ESMAC, which is introduced in Section 3.2.4, is the generalization of the Emergent Task Decomposition Network (ETDN) architecture [29]. This architecture consists of several neural networks, called expert networks, that should specialize in a specific task while running the EA. Besides the expert networks, decision neurons are evolved that aim to select the correct expert network to use based on the sensor inputs. Figure 2.6 shows the layout of this architecture with four expert networks. Selector ESMAC replaces the decision neurons with a perceptron network and puts one in each state to allow for state-dependent state changes.

In summary, research in generating state machines using an EA is in its early stages. A compact form of representing conditions on transitions has been identified, which will be used in this thesis. Furthermore, this work will, to the best of our knowledge, be the first to introduce a method to simultaneously evolve a SM's topology and each state's behaviour.

Chapter 3

Evolving State Machines As Controllers

The goal of this thesis is to introduce an automated generation procedure for robot controllers that are more understandable for humans than multi-layer Neural Networks (NNs). This generation procedure utilizes an Evolutionary Algorithm (EA) to evolve a State Machine (SM) that will be used as controller. This section introduces the layout of the SM-based controller and an automated generation procedure that evolves these controllers using an EA. The overall system is called after what it does, namely Evolving State Machines As Controllers (ESMAC).

3.1 State machine-based controller

Figure 3.1 shows the layout of the high-level robot controller used in this thesis. The controller's inputs are the robot's sensors indicated by the set $I = \{I_1, ..., I_n\}$ with n being the number of sensors. These sensors can range from a simple distance sensor to a more complex LIDAR system or radio receiver. The controller's outputs are signals to the robot's actuators and other peripherals, such as motors and radio transmitters. Output signals are indicated with $O = \{O_1, ..., O_m\}$ with m being the number of actuators.

At the base of the controller is a finite Mealy state machine [26], which is defined by a 6-tuple, $(S, s_0, \Sigma, \Lambda, T, G)$, consisting of a finite set of states S, an initial state s_0, a set of input signals Σ, which are all combinations of inputs from the sensors, a set of output signals Λ, which are all combinations of the outputs of the controller, a transition function $T : S \times \Sigma \to S$ and an output function $G : S \times \Sigma \to \Lambda$.

The output function G for each state is defined by single-layer NN or, in other words, a perceptron for each output. A single-layer NN makes the working of a state understandable. Every state has its own single-layer NN, which makes the output function state dependent.

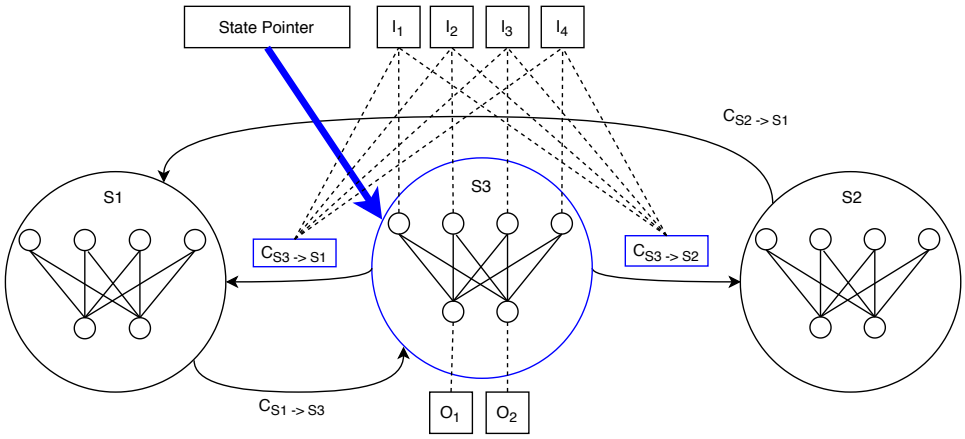


Figure 3.1: Layout of the state machine-based robot controller. I_i indicates sensor i. O_j indicates actuator j. The states of the state machine are the large circles marked with S_i. $C_{S_i \to S_j}$ indicates the condition for branching from state i to state j. The state pointer (indicated by the blue arrow) points to the current state of the robot.

In the following the NN of state s_i will be indicated with NN_{s_i}. At every controller step t, the NN of the controller's state at time t is executed with as inputs the sensor readings and its outputs are fed to the actuators. As stated before, NN_{s_i} for all states $s_i \in S$ consists of a perceptron for each output. Therefore, output j given s_t, the controller's state at t, can be calculated as follows:

$$O_{j,s_t} = g(b_{j,s_t} + \sum_{i=1,\ldots,n} w_{i,j,s_t} I_i) \tag{3.1}$$

In this equation w_{i,j,s_t} is the weight of NN_{s_t}'s connection from input i to output j and b_{j,s_t} is NN_{s_t}'s bias of output j.

The state pointer keeps track of the current state of the controller. A state transition from state i to state j is defined by the tuple $T_{S_i \to S_j} = (o_b, c)$ where o_b is a boolean operator in $\{and, or\}$ that is used for combining the conditions and $c = \{c_1, \ldots c_n\}$ is a set of conditions. Each condition $c_k \in c$ is of the form:

$$I_l \ [<, >, =] \ v_k \tag{3.2}$$

where I_l is a sensor input and v_k is a numeric value. Every controller step, the transition function T evaluates all outgoing transitions of the current state. If all transitions evaluate to false the current state is returned. Otherwise the transition function returns the destination of a randomly selected transition that evaluates to true.

3.2 Evolving a state machine-based controller

To automatically generate state machines as proposed in the previous section a specially designed EA, called Evolving State Machines As Controllers (ESMAC), is introduced. There are three versions of ESMAC: standard ESMAC, adaptive ESMAC, and Selector ESMAC. Adaptive ESMAC extends standard ESMAC by dynamically changing the number of states and selector ESMAC slightly changes standard ESMAC by replacing the transtitions with NN-based state switching. But first, the genotype of a SM-based controller is introduced.

3.2.1 Genetic encoding

The genotype of a state-transitioned controller is shown in Figure 3.2. Each state is represented by a *state gene*. A state gene contains an identifier, the weights for each of the connections in the neural network $w_{i,j}$ and a bias b_j for each of the output nodes of the network. The state's identifier is unique and cannot be changed by mutation or crossover.

A transition is represented by a *transition gene*. A transition gene refers to the state where the transition comes from, S_{from}, and to the state where the transition goes to, S_{to}. Furthermore, it contains a set of conditions in the form introduced in the previous section and a boolean operator for combining them. In addition to those, it contains a boolean to signal whether the transition is enabled, that is the transition can be taken. This enable bit allows the (temporal) removal of transitions without losing the conditional information of the transition.

3.2.2 Standard ESMAC

Standard ESMAC optimizes the states and transitions of a fully connected SM with user-defined number of states n_{states}. A population of SM genomes is initialized by randomly initializing genomes with n_{states} state genes. Furthermore, a transition gene with one random transition condition is initialized between every pair of states. The resulting genome encodes a controller that consists of a fully-connected SM.

The standard evolutionary process, as introduced in Section 2.2, is used for generating an optimized controller. Every generation the best 20% of controller genomes is used as a seed for the next generation. Elitism, introduced in Section 2.2, ensures that the k best genomes continue to the next generation without modification. The other $n - k$ genomes of the next generation are generated using crossover.

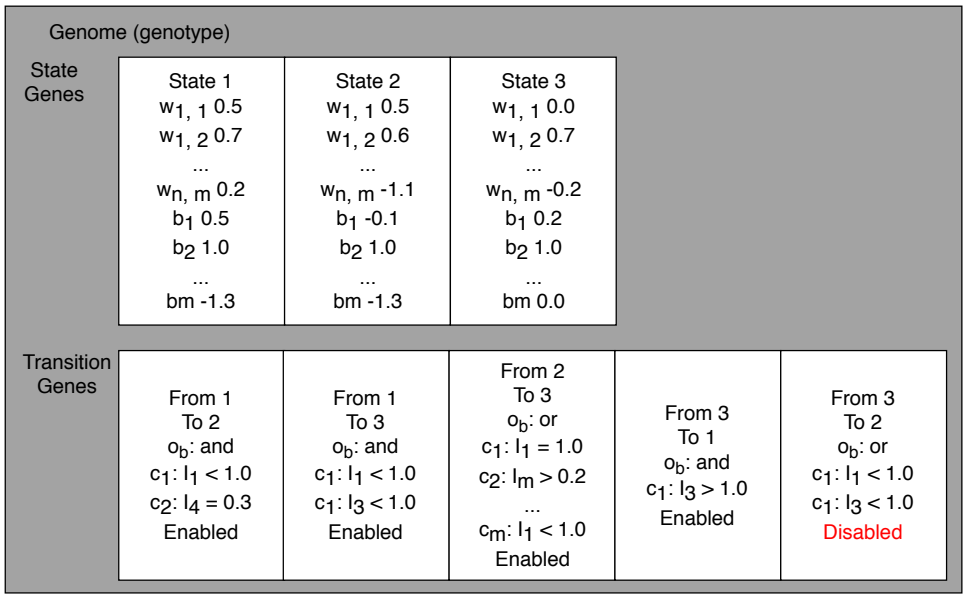


Figure 3.2: Genotype of a robot's controller.

Crossover

Crossover works on 2 randomly-selected seed genomes. The states are sorted on identifier and a random cut divides the states in both genomes in two parts. The states in the first part are in the cut; the others are not. Two new genomes are generated by swapping the states that are not in the cut. Transitions genes that do not have endpoints in the cut are also swapped. Each transition that crosses the cut is swapped with a 50% probability. Figure 3.3 shows an example crossover on 4-state SMs.

Mutation

Mutation randomly mutates the state genes and transition genes in every newly generated genome. The NN in each state gene is mutated as described in Section 2.2.1. Transition genes can be mutated in five ways: a *toggle enable mutation*, a *change condition operator mutation*, an *add condition mutation*, a *change condition mutation*, and a *remove condition mutation*. Each mutation has a user-defined probability of occurring. The enable bit of a transition can be toggled with the *toggle enable mutation*. The *change condition operator mutation* changes the boolean operator o_b, which is used for concatenating the conditions. Adding a new randomly initialized condition to the transition is done with the *add condition mutation*. A *change condition mutation* randomly changes one of the conditions. A condition can be changed by randomly changing the sensor used on the condition, by randomly changing the condition's comparison operator, or by mutating the

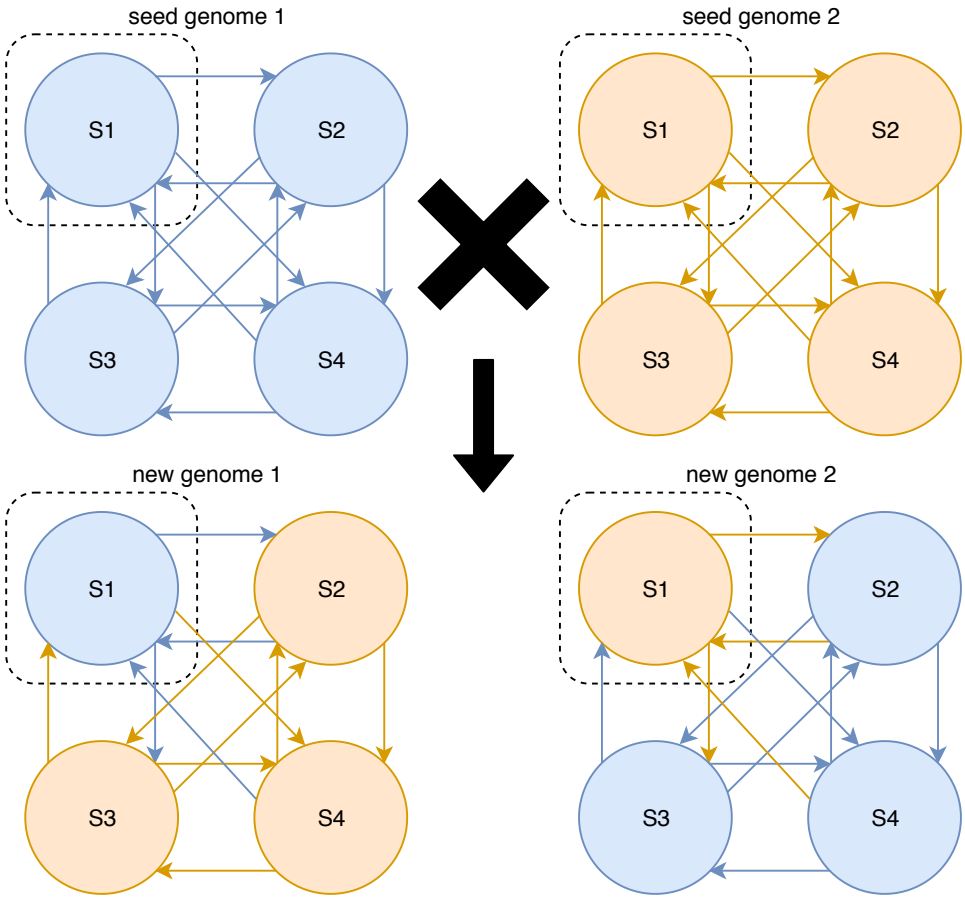


Figure 3.3: Crossover of two controllers in standard ESMAC. The dashed line indicates the cut of the SMs. States that are not in the cut are swapped. Transitions that cross the cut are swapped with a 50% probability.

comparison value. For example, the condition $I_1 == 1.0$ can be mutated to $I_2 == 1.0$, $I_1 > 1.0$ or $I_1 == 0.5$. The *remove condition mutation* removes one of the conditions from the transition.

3.2.3 Adaptive ESMAC

In standard ESMAC the number of states are fixed and each state is connected to all other states by transitions. Adaptive ESMAC is inspired by Neuroevolution of Augmenting Topologies (NEAT) [28] (Section 2.2.1) and allows for adding and removing states and transition with the aim to generate minimal SMs. Initially a random population of genomes with only one state is generated. Mutations extend the genomes while the EA runs.

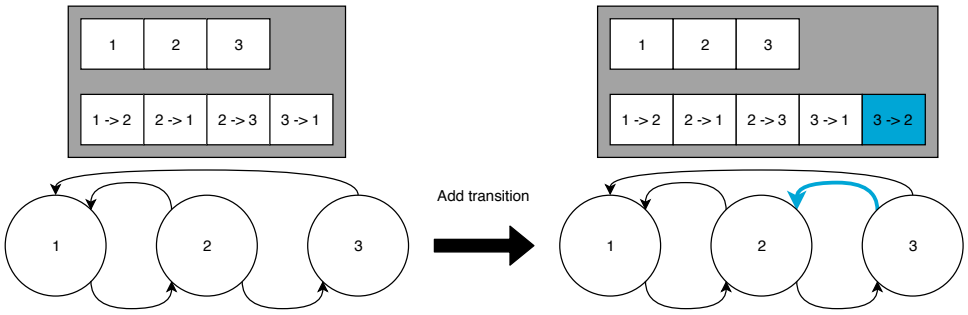


Figure 3.4: Illustration of the *add transition* mutation, with the genome shown above the corresponding state machine. Changes are colored blue.

Mutation

Adaptive ESMAC extends standard ESMAC with two mutations: an *add transition mutation* and an *add node mutation*.

The *add transition mutation* selects two random states and creates a transition between them. In case a transition was already present it is enabled. A new transition contains one condition, which is randomly initialized. Figure 3.4 shows an example of such a mutation.

The *add state mutation* introduces a new state, with a randomly initialized neural network, and connects it to the SM by an outgoing transition and an incoming transition both from a random state. As an identifier the current number of states plus one is used. This allows for diverse crossover because states are matched based on this identifier, which does not say anything about the state's functioning. An example of this mutation is shown in Figure 3.5.

Crossover

Extending individual genomes means that there can be genomes with a different number of states or a different number of transitions within a population. Therefore, a new crossover operator is required that is able to deal with different genome topologies. This crossover operator matches the states and transitions, the building blocks of a genome, of two seed genomes, g_{p1} and g_{p2}, and combines them into a child g_{child}.

States are matched based on their identifiers. State identifiers are well suited for matching, because every newly introduced state gets an identifier. For states that are present in both parents, one of the states is selected at random to be included in the child. Furthermore, the states that are present in the fittest parent, but not in the other, are also included. Using this selection method the state's functioning, i.e. its neural network stays intact. This is in line with the assumption that the functioning of the state

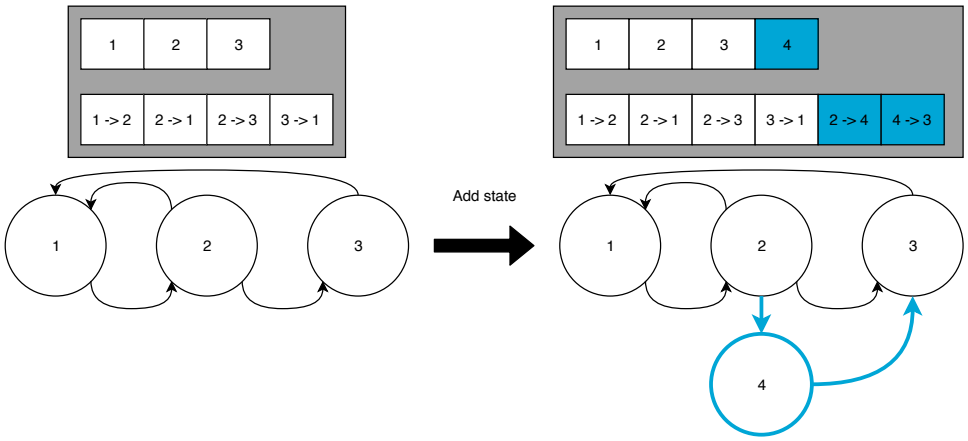


Figure 3.5: Illustration of the *add state* mutation, with the genome shown above the corresponding state machine. A state is added and connected to two random existing states. The blue color indicates change.

contributes to the better fitness and crossover within a neural network can break useful parts of the network [8].

The transitions are matched based on the pair (S_{begin}, S_{end}), their begin- and end state. As with states, when the same transition is present in both parents one of the transitions is selected at random. Transitions that are only included in the fittest parent are also selected. Though the origin of matched states and transitions is similar because of matching identifiers, their inner working can be completely different due to mutation. However, only identifiers are used for matching because they allow for easy topology matching.

Figure 3.6 shows the crossover operation of two parents with an equal fitness. It shows that a random state or transition is selected in case of matching identifiers. States or transitions without a match are always included in the child, because the parents are equally fit.

Speciation

New structures are given a change to evolve trough speciation similar as in NEAT [28]. The population of controllers is divided in groups based on similarities in topology. This means that controllers with a similar topology end up in the same species. Speciation protects a new SM topology and allows it to optimize the weights and biases of the NNs and the conditions of the transitions. Without speciation a new topology that does not perform well because it is not (yet) optimized will not survive one generation.

A compatibility distance δ is used to indicate the compatibility or similarity between two genomes. The compatibility distance is a linear combination of the number of unique states (D), unique transitions (T), the difference

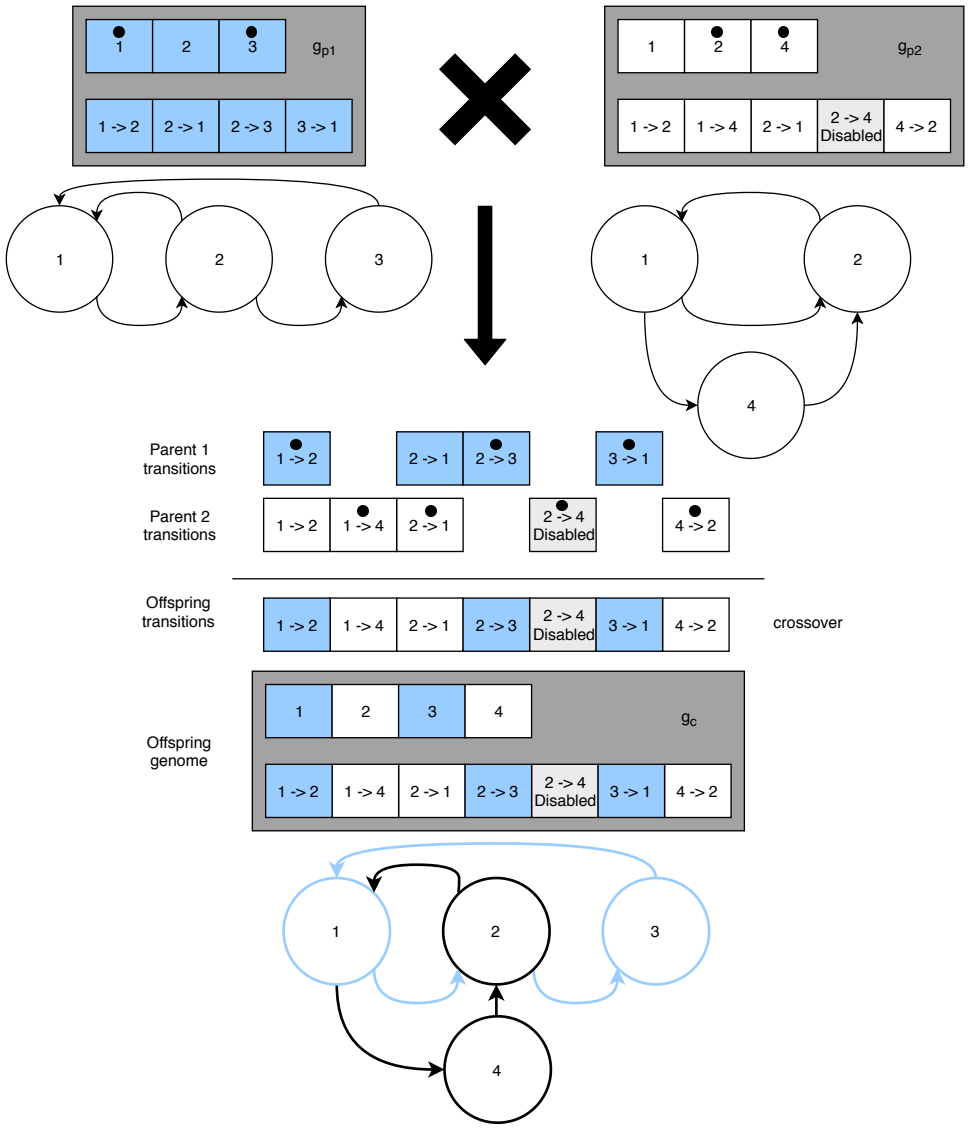


Figure 3.6: Crossover operation of ESMAC. The states and transitions are matched based on their identifier. Selection in the child is indicated by a black dot. In this example the parents are equally fit so matches are randomly resolved.

between the number of conditions in matching transitions (δ_C), and the average weight and bias difference of the neural networks in the matching states ($\overline{\delta_W}$):

$$\delta = c_1(D + T) + (1 - c_1)(\delta_C + \overline{\delta_W}) \tag{3.3}$$

where c_1 is a coefficient to adjust the importance of the factors. For two matching states S_i and S_j the average weight and bias difference, $\overline{\delta_W}$, is

calculated by:

$$\overline{\delta_W} = \frac{\sum_{k=1}^{n}\sum_{l=1}^{m}|w_{k,l,S_i} - w_{k,l,S_j}|}{m * n} + \frac{\sum_{l=1}^{m}|b_{l,S_i} - b_{l,S_j}|}{m} \tag{3.4}$$

where w_{k,l,S_i} is weight $w_{k,l}$ in state S_i, b_{l,S_i} is bias b_l in state S_i, m is the number of outputs, and n is the number of inputs. For two matching transitions T_i and T_j the difference between the number of conditions is calculated by:

$$\delta_C = \frac{\big||C_i| - |C_j|\big|}{\max\{\big||C_i| - |C_j|\big|\Big|\text{for all matching } C_i, C_j\}} \tag{3.5}$$

where $|C_i|$ gives the number of conditions of transition i.

The genomes are divided in species as in NEAT. A genome is put in the first species where the compatibility with a random genome in that species is less than δ_t, the compatibility threshold. If δ_t is exceeded for each species a new species is created.

For reproduction explicit fitness sharing is used [14]. Explicit fitness sharing allows small species to survive in a population by dividing the fitness of each genome by the number of individuals in its species. Therefore the adjusted fitness of an individual i in species j is: $f_{ij} = \frac{f_i}{|\text{species}_j|}$, where f_i is the fitness of the individual as evaluated. Species are then resized using the following formula:

$$N_j^{'} = \Big\lfloor \frac{\sum_{i=1}^{N_j} f_{ij}}{\overline{f}} \Big\rfloor, \tag{3.6}$$

where N_j and $N_j^{'}$ are the old respectively the new number of individuals in species j, f_{ij} is the adjusted fitness of individual i in species j and $\overline{f}$ is the mean adjusted fitness in the entire population. When rounding has caused the population size to decrease, the size of species with the highest average adjusted fitness is increased so that the population has its original size.

$N_j^{'}$ offspring is generated for each species performing crossover and mutation among the best r% of the species, where r is a human input.

3.2.4 Selector ESMAC

The above described implementations of ESMAC use state transitions to switch between states. However, experiments show that transitions are hardly used, which causes the generated controllers to function as a single-state controller. The sheer amount of possible transitions might be one of the causes that transitions are hardly used. Another possible cause is that adding a transition can completely change the controllers behaviour, which means that similar controllers do not produce similar fitness. Therefore,

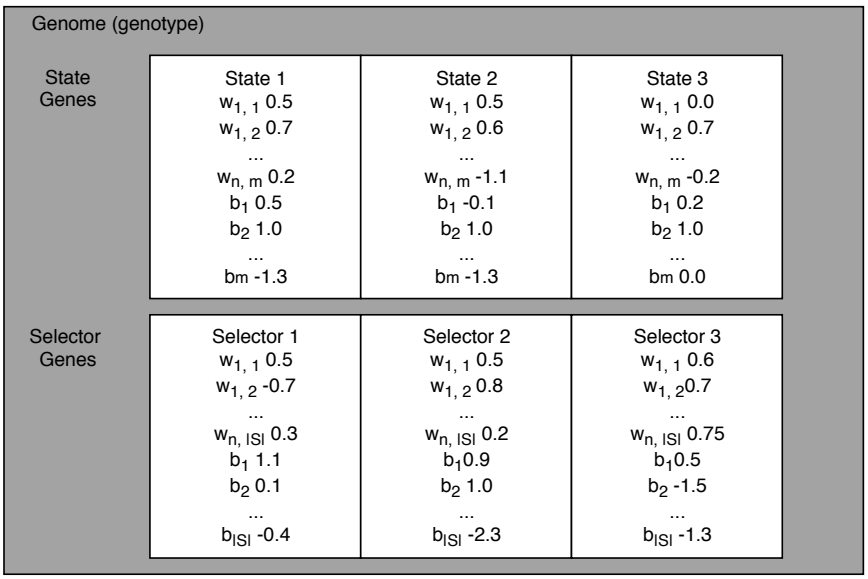


Figure 3.7: Genotype of a robot's controller when selectors are used for state switching.

selector ESMAC replaces transition genes with selector genes, which is assumed to make changes to the controllers behaviour more gradual. Selector genes, one for each state gene, contain the weights and biases for a perceptron network with as inputs the sensor values and as many outputs as there are states. A controller executes the perceptron network represented by the selector gene of the current state and the index of the highest score is the next state. Using this method of selecting the next state it is expected that the number of state changes increases, because all sensors influence the decision and not just those that are on the transitions. Furthermore, a next state is always selected, and a lack of transitions (or transitions with conditions that cannot become true) does not influence the number of possible next states.

As an example for how state changes are made with this controller type consider a 3-state controller that is in state 1 at time t. Then after feeding the output of state gene 1's perceptron network to the actuators, it also runs the perceptron network represented by selector genome 1. Suppose the output of this network is $[0.5, 0.0, 0.6]$. The highest output is at index 3 (starting from 1) and therefore the state at $t + 1$ will be state 3.

Figure 3.7 shows the genome of a state-selected controller. Crossover goes as in standard ESMAC, by making a cut and swapping the state genes (and selector genes) that are not in the cut. Both the state genes and the selector genes are mutated as the state genes in standard ESMAC.

Chapter 4

Experimental Setup

Repeatable experiments are important because they allow results to be verified by others. Therefore, it is of sheer importance to understand how experiments are conducted and what experiments are conducted. This chapter focuses on how the experiments in this thesis are conducted, the next chapter focuses on the experiments and their results. This chapter, first, explains the software components that are part of the experiments and how they interact. Second, the scenarios used to evaluate the performance of the robot controllers are introduced.

4.1 Software Components

Two parts have been implemented to evaluate the performance of Evolving State Machines As Controllers (ESMAC): ESMAC's Evolutionary Algorithm (EA) and a simulation engine. Those parts are interconnected using the Robot Operating System (ROS) [24], a platform often used in robotics to provide support communication between components within a robot. Figure 4.1 shows how the EA interacts with the simulation engine. The code for these components, including usage details, can be found on: `https://github.com/matthijsdentoom/ros-ea`.

4.1.1 Implementation of ESMAC's EA

The task of ESMAC's EA is to generate controllers and randomly modifying them while keeping the best controllers. Chapter 3 describes this process. The performance of the controllers created by the EA is evaluated in a simulation engine that runs the required task. ESMAC's evolutionary algorithm has been implemented in Python mainly because Python's ease of use and ease of modification. Another advantage of using python for the EA is that many other EAs are implemented in python, which, therefore, allows for direct comparisons.

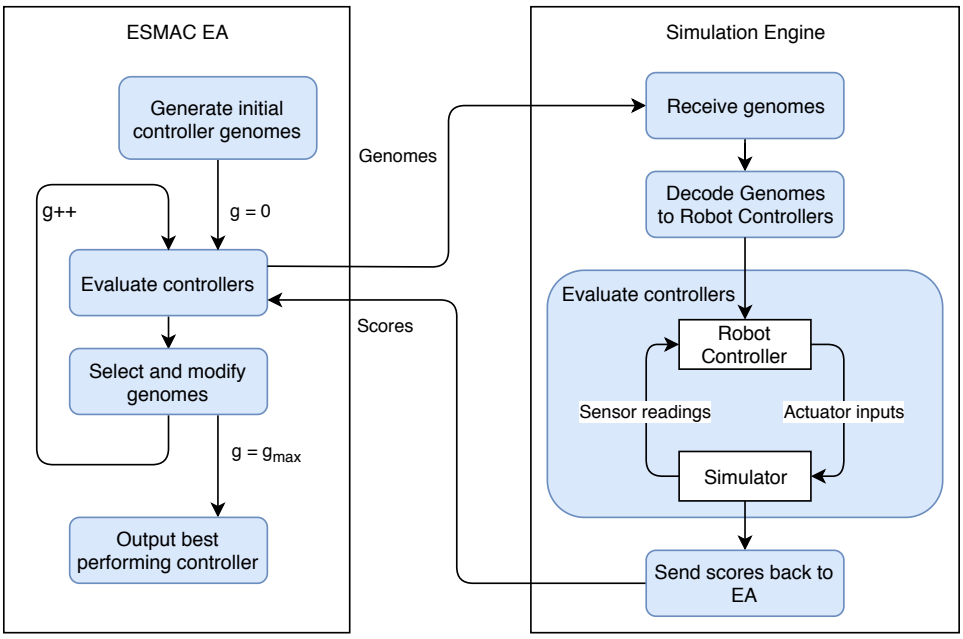


Figure 4.1: Interaction of the software components of ESMAC. g stands for generation and g_{max} for the number of generations the algorithm runs.

As a basis for implementation a Python library called neat-python [17] is used. This library contains an implementation of the Neuroevolution of Augmenting Topologies (NEAT) algorithm (Section 2.2.1). Many elements of ESMAC are already implemented in this library because of ESMAC's resemblance to NEAT. Using this library, mutation and crossover of controllers is implemented as described in the previous chapter. Every iteration of the algorithm these genetic operators are used to generate new controllers. The generated controllers are sent to the simulation engine that evaluates the performance of the controllers on a simulated robot and returns the results to the EA. These results are used to select the controllers in the next generation.

4.1.2 Implementation of the simulation engine

The simulation engine's task is to decode a genome sent by the EA and run the resulting controller on a simulated robot for a user-defined amount of time. After running the controller, the controller's performance with respect to predefined metrics is reported to the EA.

For the evaluation of the controllers, a robot simulator called Autonomous Robots Go Swarming (ARGoS) is used. ARGoS [23] is a simulation platform for (swarm) robotics research. This simulator has been chosen because it is extensively used in robotic research and because it already includes physics

engines and robot models. Furthermore, ARGoS is easily adapted to be used as evaluation tool for an EA. Multiple instances of the simulation engine can be launched, allowing the time consuming task of evaluating the performance of the robot controllers to be executed in parallel.

The robot used in simulation is foot-bot, a model of which is already included in ARGoS [23]. The foot-bot has 24 proximity sensors and 24 light sensors both in a ring around its body. To reduce the complexity of the controllers, the number of proximity sensor and the number of light sensors are both reduced to 12 in this project. Therefore, there are 24 sensor inputs to a controller. The range of a light sensor is unlimited (with an exponential reduction in detection strength as distance increases) and the range of a proximity sensor is 0.5 meter. All sensor readings are normalized in the range [0, 1]. The robot uses differential drive to move around, which means that the robot has 2 actuators, a left and a right motor. A controller needs to provide a normalized motor speed in the range [-1, 1] for each of the two motors.

4.2 Evaluation Tasks

To investigate whether it is possible to evolve a State Machine (SM) controller that outperforms a Neural Network (NN) controller, robot tasks that (to the eye) need to be executed with a multi-state controller are required. Such tasks have two characteristics: 1.) different actuator outputs need to be generated with the same sensor inputs at different phases of the task, i.e. the task consists of mutually exclusive sub tasks, and 2.) the robot controller can deduct which sub task to execute based on the sensor input. Unfortunately there are no reproducible benchmark tasks have these characteristics and can be used to experiment with automated task decomposition. Therefore, this section introduces two simple robotic tasks that comply with these characteristics: *come-and-go* and *phototaxis-with-obstacles*.

4.2.1 The come-and-go task

A simple task with potential for sub-task decomposition is *come-and-go*. This task requires a robot to go to a light and once the light has been reached move as far as possible away from the light. In real-life this task occurs when a robot needs to find a bomb, pick it up and get as far away as possible from its original location in order to ensure that location's safety. This task can benefit from multiple states as the light sensors are first needed to find the light and once the light has been found to steer away from it.

Figure 4.2 shows the layout for a *come-and-go* scenario. The initial orientation and location of the robot is randomized to avoid overfitting. At the beginning of each simulation the robot is dropped somewhere in the blue area.

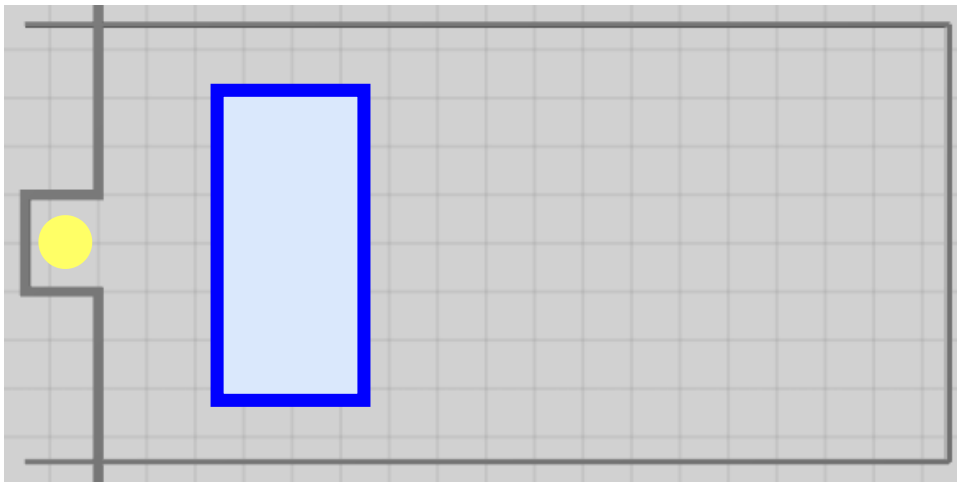


Figure 4.2: Layout of *come-and-go* scenario. The yellow circle indicates the location of the light and the gray elements are obstacles. The robot is dropped randomly in the blue area at the beginning of each simulation.

Fitness Evaluation

The fitness $f_{c,i}$ of a controller, c, in a specific *come-and-go* scenario i is indicated by the distance between the robot and the light after the light has been found. Controllers that are not able to find the light are not rewarded. Equation 4.1 gives the fitness function for a *come-and-go* scenario, where loc_{light} is the location of the light, $loc_{robot,t}$ is the location of the robot at time t and t_{sim_end} is the duration of the simulation, which is either the number of time steps until the first collision or t_{max}, the maximum duration of a simulation. A higher fitness indicates better performance.

$$f_{c,i} = \begin{cases} 0, & \text{if } |loc_{light} - loc_{robot,t}| > 0.1 \text{ for } t = 0, ..., t_{sim\_end} \\ |loc_{light} - loc_{robot,t_{sim\_end}}|, & \text{otherwise} \end{cases} \tag{4.1}$$

The fitness of a controller, f_c, is the average performance on the evaluated scenarios:

$$f_c = \frac{\sum_{i=1}^{\#\text{ scenarios}} f_{c,i}}{\#\text{ scenarios}} \tag{4.2}$$

An EA uses this fitness for selection among controllers.

4.2.2 The phototaxis-with-obstacles task

Another simple benchmark task used to evaluate the effectiveness of evolving a SM instead of a 'normal' NN is *phototaxis-with-obstacles*. The robot's task is to go to a light using its light sensors, without colliding with obstacles.

This task has been chosen because it can be decomposed into sub tasks, for example *avoid obstacles* and *go towards the light*. While this decomposition does not consist of mutually exclusive sub tasks, because a robot can avoid obstacle while going to the light, the sensor outputs can clearly indicate when a state transition is required, namely when the robot gets close to an obstacle. Two variants of this task have been used for the evaluation of ESMAC: a set of static scenarios and randomly generated scenarios.

Static scenarios

Figures 4.3 (a) - (d) show the layout of the four static scenarios used to evaluate a controller's performance while the EA is running. In this thesis, these scenarios are called training scenarios. Using fixed scenarios instead of randomized scenarios during generation makes the experiments repeatable, but it comes with the risk of overfitting on the selected scenarios. Overfitting, which means that the generated controllers work well in the training scenarios but not in other scenarios, is undesirable because the EA is not able to generalize the task. To reduce the risk of overfitting, the four scenarios are of incremented difficulty, while the obstacles of different scenarios do not overlap completely.

Scenarios 5 and 6, as seen in Figure 4.3 (e) - (f), are evaluation scenarios that are only used to evaluate the robustness of the controllers generated using an EA. The evaluation scenarios are selected because they rely on the same principles as the training scenarios, but are significantly different. Scenario 6 is also more complex than the training scenarios as multiple obstacles need to be avoided.

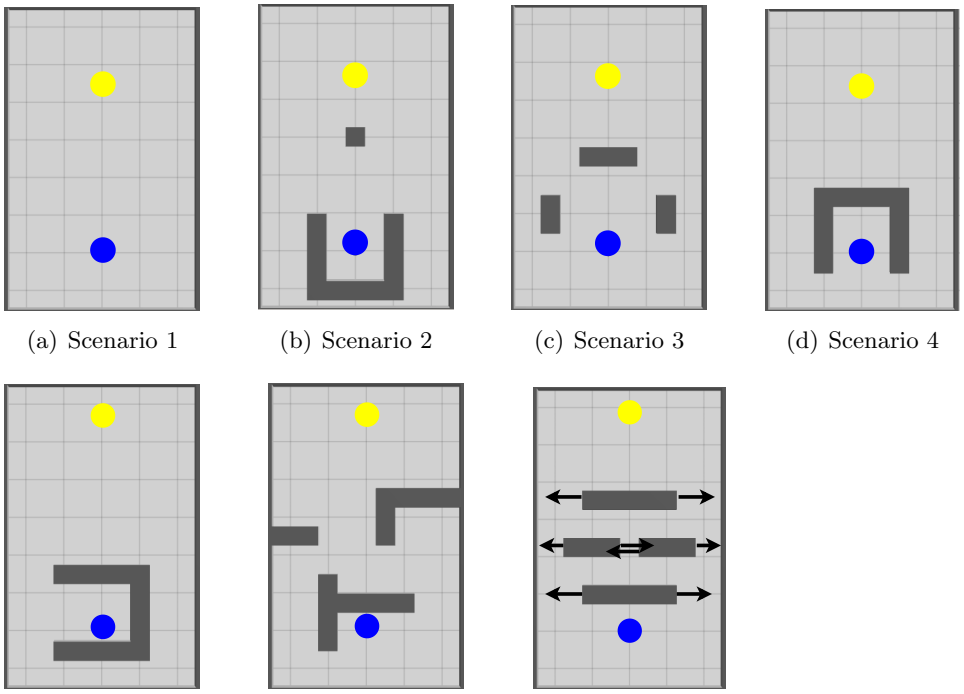
Random scenario

Because evaluating the controllers on fixed scenarios increases the risk of overfitting, a scenario with randomly positioned obstacles is also created. Figure 4.3 (g) shows this scenario. Four obstacles are positioned between the robot and the light, and are moved along the x-axis by a random number in range [-1, 1] when the environment is reset. ESMAC evaluates each controller on four randomly generated scenarios.

Fitness Evaluation

The performance (or fitness) of a controller, c, on a specific phototaxis scenario i in the simulator is calculated by:

$$f_{c,i} = \sum_{t=1}^{\min(t_{max}, t_{collision})} 1 - \frac{|loc_{light} - loc_{robot,t}|}{distance_{max}} \tag{4.3}$$



(a) Scenario 1 (b) Scenario 2 (c) Scenario 3 (d) Scenario 4

(e) Scenario 5 (f) Scenario 6 (g) Random scenario

Figure 4.3: The layout of six static scenarios and one random scenario of the phototaxis task with obstacles used to evaluate ESMAC's performance. The yellow circle is the light, the blue circle is the starting point of the robot, and the grey blocks are obstacles. The arrows in the random scenario indicate the possible deviation of the obstacles.

where t_{max} is the user-defined maximum running time of the simulation, $t_{collision}$ is the time of the first collision, loc_{light} the location of the light, $loc_{robot,t}$ the location of the robot at time step t and $distance_{max}$ the maximum possible distance between the robot and the light. A higher fitness indicates better performance.

As with the *come-and-go* task, the fitness of a controller in ESMAC (or another EA) is the average score of the evaluated scenarios and is calculated by Equation 4.2.

Chapter 5

Evaluation and Results

The goal of this thesis is to investigate the possibilities of evolving a State Machine (SM) instead of a Neural Network (NN) using an Evolutionary Algorithm (EA). To this end, an EA that evolves SMs, named Evolving State Machines As Controllers (ESMAC), has been introduced in Section 3. This section presents the performance of ESMAC on the evaluation tasks that have been introduced in Section 4.2. A detailed overview of the parameters used for these experiments is given in Appendix A.

5.1 ESMAC on the come-and-go task

The *come-and-go* task requires two complementary actions to be executed, depending on whether the light has been found or not. To achieve this, a multi-state SM is assumed to be a good controller choice. Figure 5.1 shows the performance of the best scoring controllers for different controller types for every generation. ESMAC is used to generate SMs controllers with three and with five states. The NN-based controllers, which are the *NN with one hidden layer* and the *perceptron network*, are generated with the standard EA introduced in Section 2.2.1.

None of the EAs are able to solve this task since a successful run with 600 time steps would allow a robot to travel at least 20 meters after finding the light, i.e. have a fitness of at least 20. The spikes in the fitness are caused by controllers that are not able to find the light, which results in a fitness score of 0 and has a visible effect on the average with only 10 randomly seeded evolutionary runs. This can be deduced from the high average standard deviation in the fitness of the best controllers over all controller types and all generations, which is 3.6 (std: 0.93).

The standard deviation of the best 3-state and 5-state controllers at the last generation are 1.52 respectively 3.95. The better performance of ESMAC with 5 states compared to ESMAC with 3 states is a mere coincidence as its standard deviation is high and the low number of repetitions of the

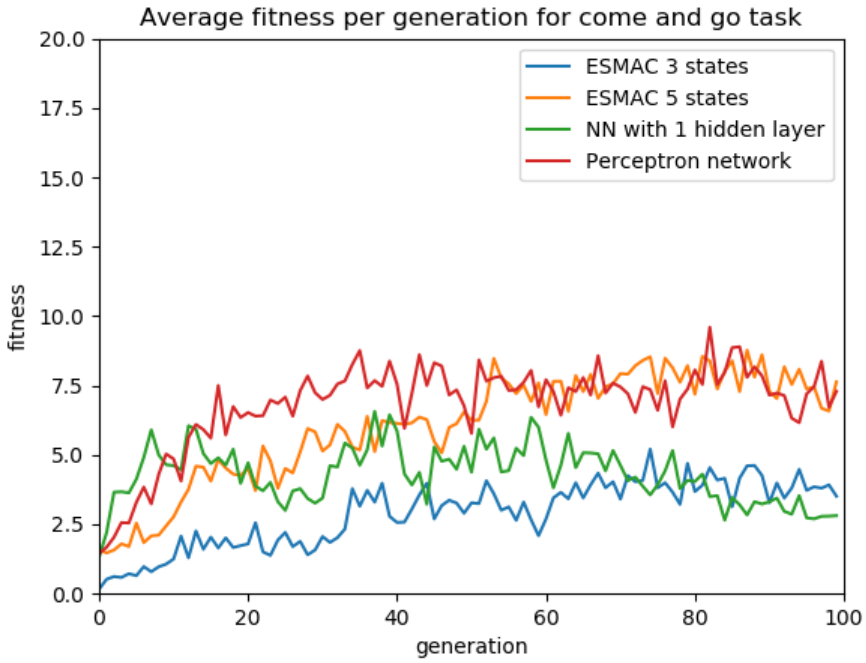


Figure 5.1: Average fitness of best scoring controllers (over 10 evolutionary runs) for different controller types with respect to the number of generations the EA ran on the *come-and-go* task.

evolutionary runs. This has been confirmed by generating 10 new controller, which resulted in completely different outcomes and a lower average fitness. A valid performance comparison is hard because too little evolutionary runs are executed for a good average fitness, but the performance of ESMAC and the NN-based EA are similar in terms of obtained fitness.

It is interesting to see whether the generated SM-based controllers actually use different states. Table 5.1 shows the average number of states used by the generated controllers and the percentage of time these controllers spend in the most-used state for 1000 randomly initialized simulations. This table shows that the controllers spend most of their time in one state. Other states are used only 1% of the time average for the 3-state SMs, which comes down to 6 time steps in this experiment. This is not enough time to make the robot considerably change behaviour and is therefore not counted as a meaningful usage of different states.

The evolutionary runs of ESMAC for a 5-state SM clearly show how the usage of different states is reduced over generations. Figure 5.2 shows the average time spend in each state during simulation sorted by duration of use for the best controllers per generation. It can be seen that while the best

	3-state controllers	5-state controllers
Average number of states used	1.44	1.15
Fraction of time in most-used state	99.0%	99.9%

Table 5.1: The average number of states used and the fraction of time spend in the most-used state for 3-state and 5-state controllers.

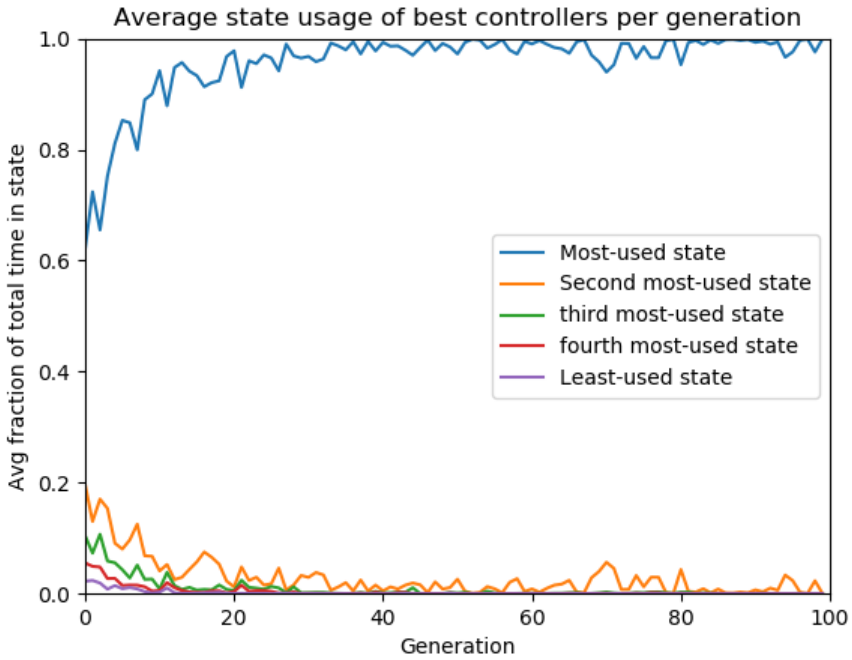


Figure 5.2: The average time spend in each state (sorted by fraction of time used) for the best 5-state SMs generated using ESMAC for each generation.

controllers in the first generations use other states besides the most-used state, the best controllers in later generations almost only use one state.

In conclusion, ESMAC is not able to generate SMs with multiple states for this task. While in early generations there are some state switches, these are filtered out in later generations. The most likely cause for this it that well performing single-state SMs are able to obtain the same fitness as multi-state SMs. As there is no benefit in using multiple states, controllers that use only one state dominate the population. This leads to the conclusion that for an easy task that only requires one state ESMAC is able to produce a simple controller that only uses one state. Therefore, the next section evaluates the performance of ESMAC on a better understood problem.

5.2 ESMAC on phototaxis-with-obstacles

The *phototaxis-with-obstacles* task is chosen because it can best be solved by a multi-state controller (see Section 4.2.2). Therefore, it is expected that SM controllers generated using ESMAC perform better than NN controllers generated using a standard EA.

Figure 5.3 shows the average fitness of the best scoring controllers for different controller types for each generation of the EA. Unless specified otherwise, the averages in this section are taken over 10 evolutionary runs to filter the random elements of the EA. This number of repetitions is chosen mainly because an analysis takes one night to finish on the used computers. More repetitions lead to more consistent results, which is important in future work. ESMAC is used to generate SMs controllers with three and with five states. The NN-based controllers are generated with the standard EA introduced in Section 2.2.1. From Figure 5.3 it can be seen that all EAs find equally good solutions. The slightly better performance of the 5-state SM controllers is due to the EA finding a controller that performs very good on scenario 4 in one of the runs, while all others EA runs (including the other nine of runs for a 5-state SM) perform medium on that scenario.

Inspection of the state usage of the best SM-based controllers shows that two of the ten generated 3-state controllers make a state switch in the training scenarios. These controllers are compared against the two best controllers that do not switch state in the training scenarios. Table 5.2 shows the average performance of the controllers that make state switches and those that do not on the training scenarios and on the evaluation scenarios. The table also shows the performance of the controllers when the sensors are noisy. Noisy sensors are simulated by adding a uniform random number in range [-0.2, 0.2] to the actual sensor value. The fitness of the entries indicated with *noisy sensors* are averages over 100 runs. The performance of the controllers with noisy sensors are an indication of their robustness.

From Table 5.2 it can be concluded that without noise the non-switching controllers perform significantly better than the controllers that do make state-switches. However, when noise is added to the sensors the switching controllers perform better in the more difficult scenarios 3, 4 and 6. Inspection of the robot's trajectory in these scenarios shows that switching controllers prevent the robot from getting stuck in a corner, whereas the non-switching controllers do get stuck.

Since controllers that utilize state transitions perform better than non-switching controllers on some scenarios when noise is added, controllers that use state transitions might be more robust than single-state controllers. This claim is strengthened when comparing the controllers in the *random* scenario, which is also introduced in Section 4.2.2. To evaluate whether state-switching controllers are more robust than controllers that uses only one state, the controllers have been run 100 times in the random scenario,

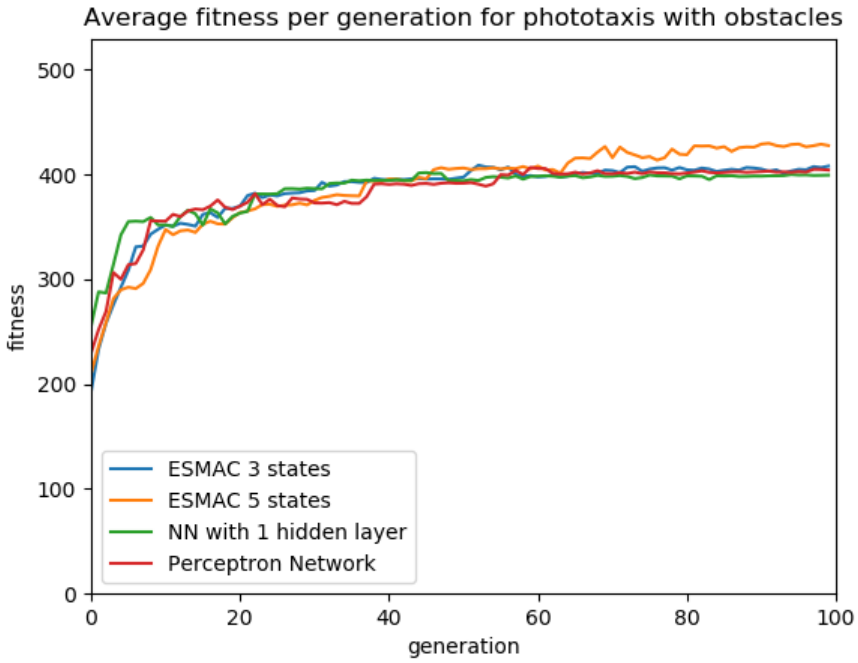


Figure 5.3: Average fitness of best scoring controllers (over 10 evolutionary runs) for different controller types with respect to the number of generations the EA ran on the training scenarios of the *phototaxis-with-obstacles* task.

Controller type / **Average Score**	switching	non-switching	switching (noisy sensors)	non-switching (noisy sensors)
Scenario 1	292.3	**353.2**	274.3	**331.1**
Scenario 2	320.5	**357.0**	308.5	**344.8**
Scenario 3	308.8	**347.2**	**297.1**	289.9
Scenario 4	320.1	**340.1**	**296.2**	270.2
Scenario 5	322.3	**372.6**	319.3	**329.8**
Scenario 6	**319.5**	251.6	**261.7**	188.5

Table 5.2: The average score of state switching controllers and non-state switching controllers with precise sensors and with noisy sensors for different scenarios. Better scores are in bold.

leading to 200 data points per controller group. Table 5.3 shows the average fitness of the controllers in each group and the number of successful simulations in which the light is reached. It shows that the controllers

	Switching controllers	Non-switching controllers
Average fitness	**431.0**	383.2
Successful simulations	**141**	121

Table 5.3: The average score of state-switching controllers and non-switching controllers and the number of successful simulations in the random *phototaxis-with-obstacles* task. Better scores are in bold.

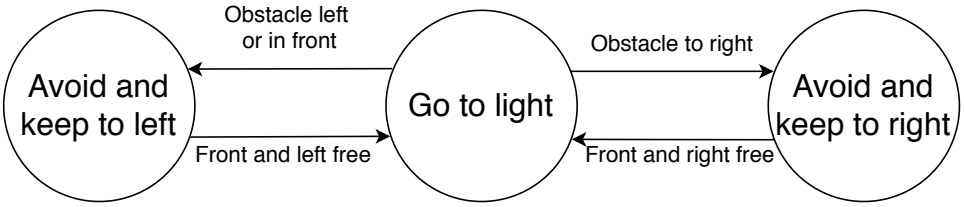


Figure 5.4: A manually-designed controller for the *phototaxis-with-obstacles* task.

that switch state on average perform better in the random *phototaxis-with-obstacles* scenario. This is shown in both a higher average fitness and more successful simulations than the non-switching controllers.

5.2.1 Using ESMAC with manual seed

As mentioned in the previous section, only two out of ten generated 3-state controllers make state transitions in the evaluation scenarios. The possibility of generate a SM using ESMAC that utilizes multiple states and is able to solve that task is investigated by running ESMAC with a manually designed controller as seed. This controller uses multiple states and is able to solve the task for the four training scenarios.

Figure 5.4 shows the SM of the manually designed seed controller. This controller is based on the intuitive solution to *phototaxis-with-obstacles*, namely to drive towards to light unless an obstacle is detected in which case the obstacle is avoided first. In this controller, *obstacle in front*, *obstacle left* and *obstacle right* are indications that respectively the front, the left-front and the right-front proximity sensor senses an object within 0.5 meters. *Front and left free* indicates that none of the sensors in front and to the left sense an object. Similarly, *front and right free* is triggered when none of the sensors in front and to the right sense an object. The performance of the seed controller is reasonable and it has been verified that this controller makes at least one state switch on all scenarios except scenario 1.

Figure 5.5 shows the evolution of the average best fitness for ESMAC initialized with a manual seed and for ESMAC initialized with randomly generated 3-state controllers. It shows that seeded ESMAC, which is standard ESMAC initialized with the manual seed described above, has a head

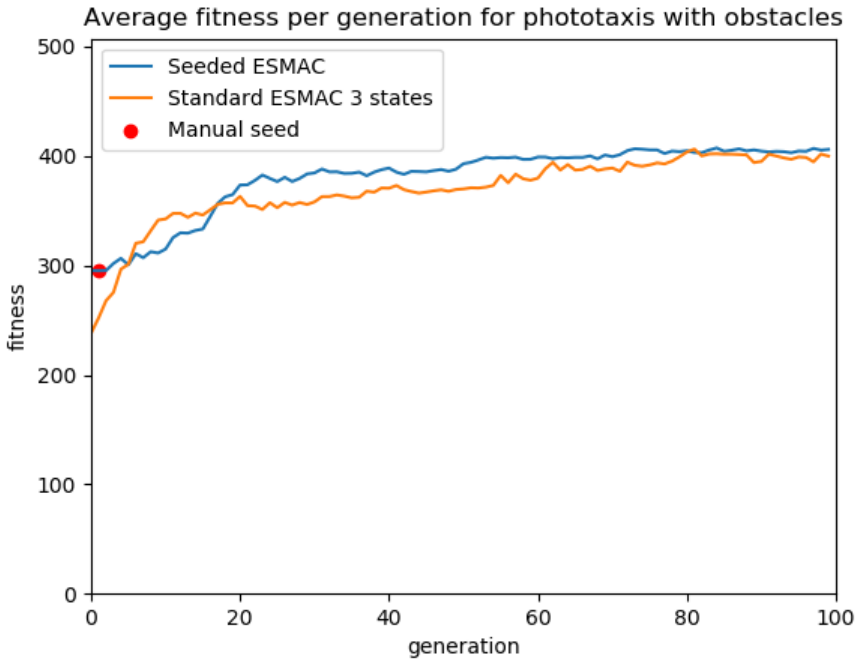


Figure 5.5: Evolution of the best fitness (averaged over 10 evolutionary runs) for ESMAC with manual seed and ESMAC with random initialization on the **training** scenarios. The average fitness of the manual seed on the training scenarios is indicated with a red dot.

start with the scores, but is not able to find significantly better performing controllers.

Comparison of generated controllers

The controllers generated using seeded ESMAC are compared with the controllers generated using standard ESMAC to find the benefits of using a seed and to find out whether it is possible to generate a controller that switches states. Table 5.4 shows the fitness and percentage of successful simulations on the evaluation scenarios and on 1000 simulations of the random scenario for the controllers generated with standard ESMAC and with seeded ESMAC. It shows that the controllers generated with seeded ESMAC perform slightly better than standard ESMAC in terms of fitness and in terms of success rate, which means the light has been found. This leads to the conclusion that seeding ESMAC with a functional, but not optimized, controller helps in finding good controllers for this task, but does not lead to large improvements.

Table 5.5 shows the average percentage of time the generated controllers

Avg fitness score	Random population	Seeded population
Scenario 5	253.1 (40%)	**322.7** (40%)
Scenario 6	124.7 (20%)	**140** (20%)
Random scenario (1000x)	263.3 (4.5%)	**330.8** (**18%**)

Table 5.4: Average fitness score for controllers generated with seeded ESMAC and standard ESMAC on the **evaluation** scenarios and 1000 runs on the **random** scenario. The percentage of successful runs is indicated between the brackets. Better scores are in bold.

Percentage in most-used state	Random population	Seeded population
Scenario 5	96.0%	97.8%
Scenario 6	99.0%	97.8%
Random scenario (1000x)	97.0%	97.9%

Table 5.5: Percentage of time the generated controllers spend on average in their most-used state for seeded ESMAC and standard ESMAC.

spend in their most-used state for both seeded ESMAC and standard ESMAC. For both seeded ESMAC and standard ESMAC these values indicate that hardly any state changes are made and most of the time is spend in one state. Inspection of the state changes shows that the initial state is not necessarily the most-used state, as some controllers make change state at $t = 0$. However, the best performing controllers are those that have the initial state as most-used state. Because all controllers hardly make any state changes, the better performance of seeded ESMAC cannot be due to the use of different states. Furthermore, because the best performing controllers have their initial state as most-used state, it can be concluded that the initial state can evolve better when initialized with a seed. This has been verified by using only the *go-to-light* state of the manually designed controller as seed, which also shows an improvement in fitness compared to standard ESMAC. This is most likely because the seed gives a good starting point in the search space.

Seeded ESMAC throughout generations

Figure 5.6 shows the average number of states used by the best performing controllers of for each generation for both seeded ESMAC and standard ESMAC. This figure shows that the number of states that is used on average by the best controllers drops already in the first five generations for seeded ESMAC. Figure 5.7, which shows the average number of states used by all controllers in each generation, shows a similar pattern. Inspection of the genomes shows that the state changes introduced by the seed controller are removed from most individuals within five generations. Figure 5.7 also shows that controllers in the non-seeded algorithm use more states than

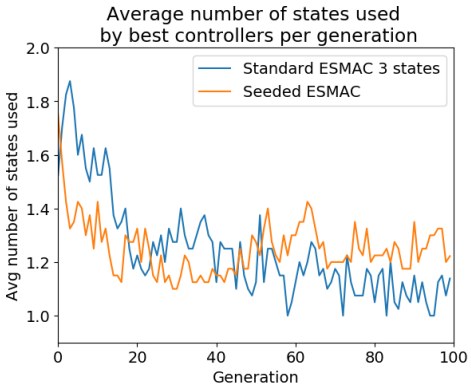


Figure 5.6: Average number of states used by best performing controllers for each generation for seeded ESMAC and standard ESMAC.

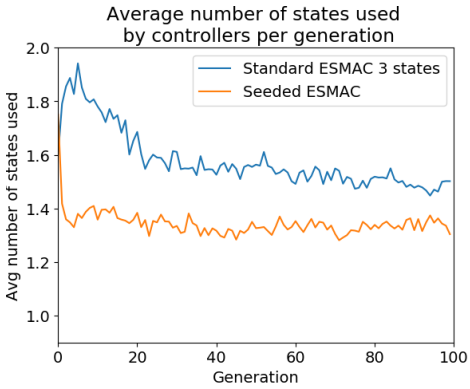


Figure 5.7: Average number of states used by all controllers for each generation for seeded ESMAC and standard ESMAC.

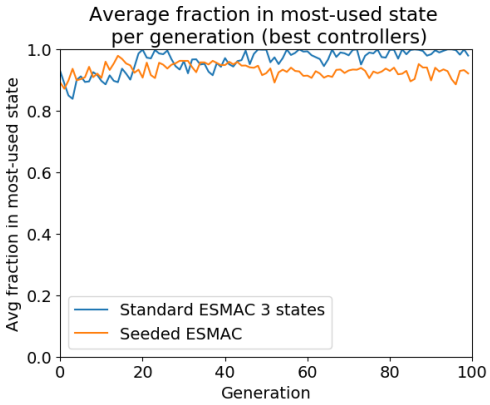


Figure 5.8: Average fraction of time spend in most-used state by best performing controllers for each generation for seeded ESMAC and standard ESMAC.

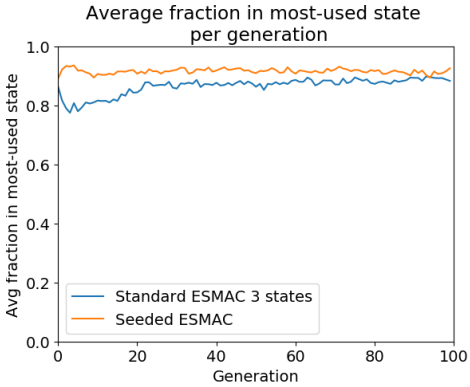


Figure 5.9: Average fraction of time spend in most-used state by all controllers for each generation for seeded ESMAC and standard ESMAC.

the controllers generated with a seed. It is unclear why this is the case. A possible explanation is that the controller genomes are more uniformly divided over the search space and therefore the likelihood of creating a valid transition during crossover or mutation increases.

Figure 5.9 shows the average fraction of simulation time that the controllers in the population spend in the most-used state during evaluation over generations. From this figure it is clear that the amount of time spend in the most-used state does not deviate much over generations. Furthermore, a dip is shown in the first 20 generations of standard ESMAC, which is in accord-

ance with the increase in number of states used that is seen in Figure 5.7. Seeding the EA increases the amount of time spent in the most-used state. This is unexpected, as the seed spends a considerable time in other states for the training scenarios (especially in scenario 4). However, as already mentioned above the seed's transitions do not cause more transitions to be made when ESMAC runs.

When only looking at the best controllers for each generation, even more time is spend in the most-used state. Figure 5.8 shows the average fraction of simulation time that the best controllers in each generation spend in the most-used state. Compared to standard ESMAC, seeded ESMAC spends more time in a different state. However, the amount of time spend in other states is still low, with about 10% in later generations. All this leads to the conclusion that while seeding ESMAC leads to controllers that seem more robust, it does not lead to controllers that make better use of multiple states than controllers generated with standard ESMAC.

Conclusions on the phototaxis-with-obstacles task

The analysis of the state-switching controllers and the lack of state switches when a functional seed controller is used, lead to the conclusion that it is highly likely that this task is best solved with a single-state SM or a perceptron network and that adding states does not improve the performance on this task. On the other side, the controllers that utilize multiple states appear to be more robust to noisy sensors and changing scenarios.

5.3 Selector ESMAC on phototaxis-with-obstacles

One of the problems with transitioned SMs is that one sensor can determine whether a transition is made or not. This might result in large jumps in the search space, as adding one transition might completely change how a controller works. Selector ESMAC has been introduced with the assumption that it makes state transitions change more gradual over generations. This should lead to a better exploration of the search space and therefore to better controllers. This assumption is evaluated by comparing selector ESMAC to standard ESMAC.

Figure 5.10 shows the evolution of 3-state controllers for selector ESMAC and standard ESMAC. This figure shows that selector ESMAC over the whole simulation finds slightly worse scoring controllers. The claim that the controllers perform slightly worse is partially enforced by Table 5.6, which shows the performance on the evaluation scenarios and the random scenario. For the evaluation scenarios the selector-switching controllers do perform worse than the transition-switching controllers, but for the random scenarios the selector-switching controllers outperform the transition-switching controllers.

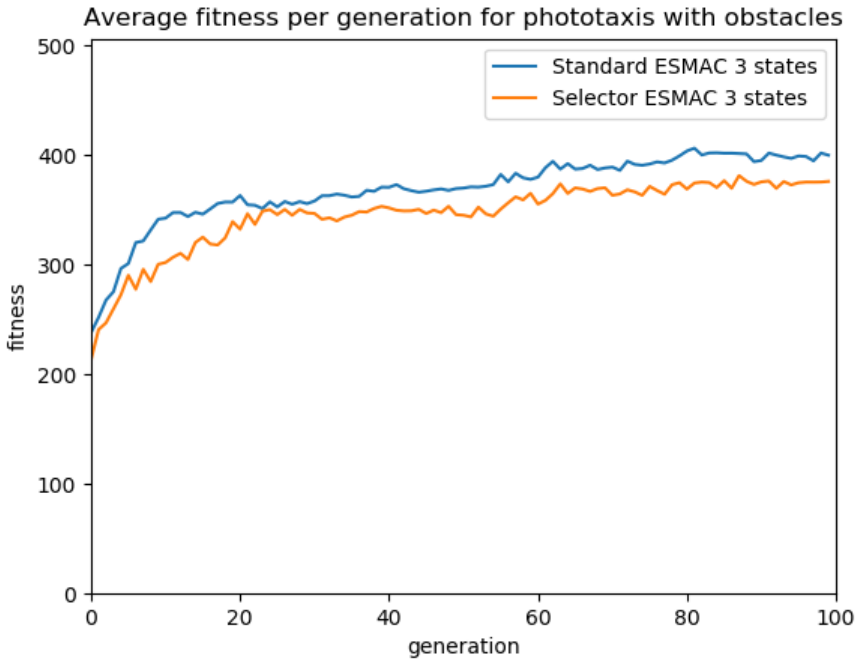
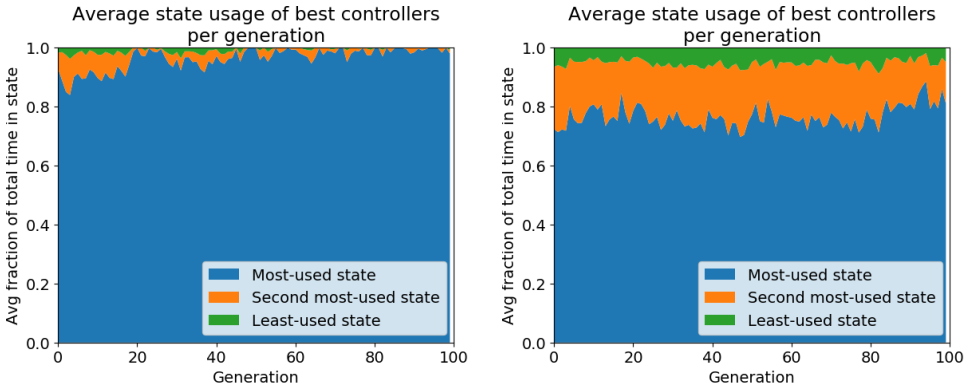


Figure 5.10: Performance best performing controllers generated using selector ESMAC and standard ESMAC on the *phototaxis-with-obstacles* task for each generations.

Avg fitness score	standard ESMAC	Selector ESMAC
Scenario 5	**253.1** (**40%**)	196.0 (20%)
Scenario 6	124.7 (**20%**	**165.3** (10%))
Random scenario (1000x)	263.3 (4.5%)	**325.6** (**21.4%**)

Table 5.6: Average fitness score for controllers generated with standard ESMAC and with selector ESMAC on the evaluation scenarios and 1000 runs on the random scenario. The percentage of successful runs is indicated between the brackets. Bold values are better values.

Selector ESMAC causes considerably more time spend in other states compared to normal ESMAC as is apparent from Figure 5.11. The limited switching caused by conditioned transitions is removed by using a perceptron network for state-switching. Whether this also causes the better performance in the random scenarios is investigated by running the full genome as well as the most-used state on the random scenarios. All controllers are run on the same 1000 generated scenarios for fair comparison. Figure 5.12 shows the performance of the 10 controllers generated with selector ESMAC along with the performance of their most-used state. The most-used state



(a) Standard ESMAC

(b) Selector ESMAC

Figure 5.11: Percentage in time in each state where the states are sorted on usage for standard ESMAC and selector ESMAC for each generation.

of four controllers (0, 1, 8 and 9) have a fitness similar to the full controller. Inspection shows that these controllers are mainly in the most-used state. Two controllers (3 and 5) do not benefit from state-switches. This is interesting, because inspection shows that these controllers spend about 90% of the running time in these states, which is a large fraction. This means that the 10% in other states has a significant effect on the fitness. The other four controllers benefit from switching away from their most-used state, so selector ESMAC is able to find controllers that make beneficial state switches, even though it does not lead to an increase in fitness compared to standard ESMAC. Controller 6 is interesting because the most-used state has such a low fitness. Inspection of the state shows that it collides with a wall most of the time. In the full controller this is prevented by a state switch before this happens, which results in the other states utilized almost a much as the most-used state. Therefore, this controller is a good example of how multiple less functional states can form a functional SM.

In conclusion, while selector ESMAC is not able to improve the fitness score on the *phototaxis-with-obstacles* task, it is able to generate controllers that use multiple states. This goes as far as controllers that use multiple inferior states to form a functional controller.

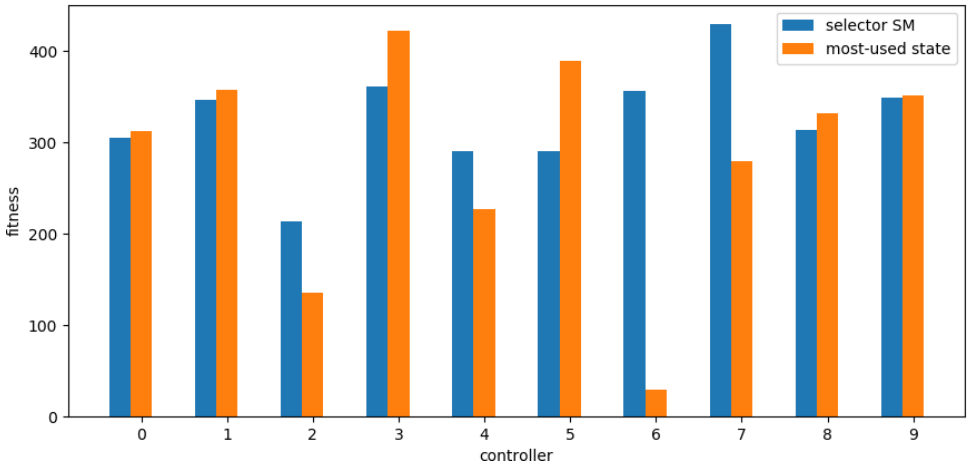


Figure 5.12: The average fitness score on 1000 random *phototaxis-with-obstacles* scenarios of the controllers generated with selector ESMAC and the most-used state of each controller.

5.4 Adaptive ESMAC on phototaxis-with-obstacles

Adaptive ESMAC's aim is automatically determine the number of states required to solve a task. Figure 5.13 shows the fitness of the best controllers for each generation for both adaptive ESMAC and standard ESMAC. Adaptive ESMAC is able to find slightly better controllers on average. The performance on the alternative and random scenarios, which is shown in Table 5.7, is slightly better than standard ESMAC.

The generated controllers exist on average of 2.5 states, with the size of the smallest controller and largest controller being 1, respectively, 5. However, the state usage is 99% in the most-used state and hardly any state changes are made. This explains the similarity in scores between adaptive ESMAC and standard ESMAC; their behaviour in simulation is similar. A task that better allows for task decomposition is required for a more detailed analysis of the species and the functionality of specialization. With such a task the effect of dynamically adding and removing states and the improvement in fitness is more visible. It has been found that the average number of states tends to grow over generations and does not stabilize at an optimal number. This is most likely because larger state machines can do the same as smaller ones. Therefore, a cost for adding a state is required, which prevents the SMs from growing unbounded.

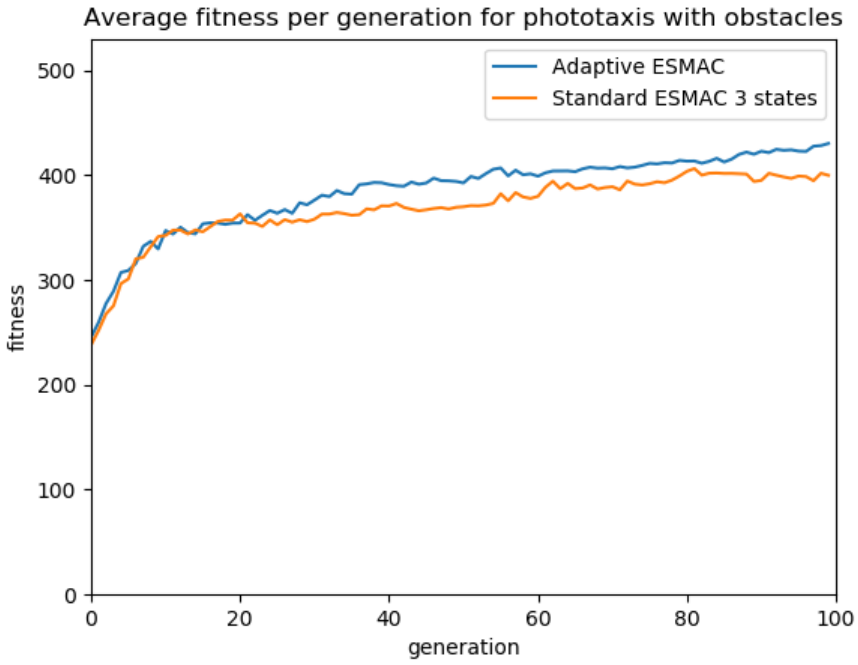


Figure 5.13: Performance best performing controllers generated using adaptive ESMAC and standard ESMAC on the *phototaxis-with-obstacles* task for each generations.

Avg fitness score	Standard ESMAC	Adaptive ESMAC
Scenario 5	**253.1** (**40%**)	217.6 (10%)
Scenario 6	124.7 (20%)	**170.4** (20%)
Random scenario (1000x)	263.3 (4.5%)	**318.0** (**6.1%**)

Table 5.7: Average fitness score for controllers generated with standard ESMAC and with adaptive ESMAC on the evaluation scenarios and 1000 runs on the random scenario. The percentage of successful runs is indicated between the brackets. Bold values are better values.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

This thesis' goal is to investigate the possibility of automatically generating State Machines (SMs) as robot controllers using an Evolutionary Algorithm (EA). To this end, Evolving State Machines As Controllers (ESMAC) has been introduced. ESMAC optimizes a set of SMs for a specific (robotic) task using an EA. A key difference between ESMAC and other SM-based EAs is that ESMAC does not use predefined states and transitions. While ESMAC is a good start towards automatically generating SMs using an EA, it does not yet succeed in automatically generating SMs that use multiple states.

Because automated task decomposition and evolving SMs as robot controllers are still in their infancy there are no benchmark tasks for comparing different task decomposition approaches. Therefore, ESMAC has not been compared to other task decomposition approaches such as, for example, the Emergent Task Decomposition Network (ETDN) architecture (Section 2.3). ESMAC has been compared with Neural Network (NN)-based EAs on two different tasks: *come-and-go* and *phototaxis-with-obstacles*.

The *come-and-go* task requires a robot to find the light and once the light has been found get as far away as possible from it. For this task, ESMAC is not able to outperform NN-based approaches. Furthermore, only minimal state changes are made by the best scoring controllers.

The more-difficult *phototaxis-with-obstacles* task, where a robot needs to find a light without colliding with obstacles, shows a similar picture. For this task only the minority of controllers generated using ESMAC use more than one state. The controllers that use more states use only two states, and their performance is slightly worse than the controllers that use one state. Promising is that the state-switching controllers are more robust

to noisy sensors and changing scenarios in the evaluated tasks. Using a manually designed controller that is functional on the task as seed for the initial population of ESMAC does not improve the quality of the generated controllers.

Selector ESMAC, which changes transition-based state switching for state switching based on the output of a perceptron network, is not able to improve the fitness score on the *phototaxis-with-obstacles* task. However, it is able to generate controllers that use multiple states and make useful state switches. One of the generated controllers even consisted of inferior states that cooperated such that the overall controller is able to perform the task. Therefore, even though transitions are less understandable with selector ESMAC, it is a good direction for future work.

Another extension to ESMAC is adaptive ESMAC, which aims to automatically determine the optimal number of states required for a task. Automatically determine the number of states required has large benefits, because it prevents users from analyzing unnecessary large SMs without having the risk of generating SMs with to few states, which makes the task intractable. Because of the challenges with standard ESMAC, little effort is put in the analysis of adaptive ESMAC. One of the findings is that a counter mechanism is needed that prevents the size of the SM to grow unlimited. Adding a cost for the size of the SM will prevent this unlimited growth, as large SMs will have a lower fitness score. This cost needs to be overcome by an increase in the fitness score on the evaluated scenarios. A detailed analysis of adaptive ESMAC is left for future work. An interesting idea for future work would also be to combine selector ESMAC with adaptive ESMAC.

6.2 Future Work

Several aspects of ESMAC remain to be investigated and improved. As many generated controllers did not make any state changes, state changes need to be rewarded by the EA. States that are not used can better be removed from the controller, however that decreases the possibility of task decomposition. A reward for state changes or state usage causes more states to be reached and thus optimized, and with that comes the opportunity of useful task decomposition. Research is required into how states can be explored without changing the actual task and reward function.

Another aspect of ESMAC that can be investigated in more detail is the encoding of SMs. In this thesis, the encoding of a SM is almost the same as the actual SM. Other encodings, for example a binary encoding, possibly allow for faster learning as the information can be represented more concise.

An important shortcoming of this thesis is the lack of real-life verification. All experiments and verification have been done in simulation. Therefore, it is unknown whether the generated controllers are able to perform their

task on an actual robot. A real-life scenario has been approximated by adding noise to the sensors, but simulation lacks important aspects of the real world. For example, what happens when one of the sensor starts outputting rubbish? These are questions that need to be answered in future work.

In the same line, ESMAC needs to be evaluated on tasks that have value in the real world. For example, can ESMAC be used to find controllers that perform well on a foraging task and with that can they help in cleaning the streets? As the idea of using SMs is to make the learned controllers understandable, robots using these controllers should be able to interact with humans and animals without unexpected behaviour.

In the grand scheme of automatically generating understandable robot controllers for complex tasks, there is a need of well-selected benchmark tasks. These benchmark tasks should capture essential aspects of real-world robot tasks. One important aspect that these tasks should have is the potential to be decomposed in sub tasks. Good candidate tasks are the tiling pattern task [30] and the swarm robotic tasks introduced in [12]. These benchmark tasks have to be accepted by experts in the field. In addition to benchmark tasks, metrics are required to measure the understandability of a robot controller. For example, one of these metrics might judge the modularity of a controller as modularity allows humans to understand a controller part by part. With those tools in hand, research can progress towards robot controllers that are understandable and good in what they need to do.

Bibliography

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

[2] Aaron M. Bornstein. Is artificial intelligence permanently inscrutable? `http://nautil.us/issue/40/learning/is-artificial-intelligence-permanently-inscrutable`, 2016. Online; accessed 7 Februari 2019.

[3] Kumar Chellapilla and David Czarnecki. A preliminary investigation into evolving modular finite state machines. In *Evolutionary Computation, 1999. CEC 99. Proceedings of the 1999 Congress on*, volume 2, pages 1349–1356. IEEE, 1999.

[4] Anders Lyhne Christensen and Marco Dorigo. Evolving an integrated phototaxis and hole-avoidance behavior for a swarm-bot. In *Proceedings of the 10th International Conference on the Simulation and Synthesis of Living Systems (Alife X)*, pages 248–254. Citeseer, 2006.

[5] Antoine Cully and J-B Mouret. Evolving a behavioral repertoire for a walking robot. *Evolutionary computation*, 24(1):59–88, 2016.

[6] Marco Dorigo, Vito Trianni, Erol Şahin, Roderich Groß, Thomas H Labella, Gianluca Baldassarre, Stefano Nolfi, Jean-Louis Deneubourg, Francesco Mondada, Dario Floreano, and Luca M Gambardella. Evolving self-organizing behaviors for a swarm-bot. *Autonomous Robots*, 17(2-3):223–245, 2004.

[7] Miguel Duarte, Vasco Costa, Jorge Gomes, Tiago Rodrigues, Fernando Silva, Sancho Moura Oliveira, and Anders Lyhne Christensen. Evolution of collective behaviors for a real swarm of aquatic surface robots. *PloS one*, 11(3):e0151834, 2016.

[8] Christos Emmanoulidis and Andrew Hunter. A comparison of crossover operators in neural network feature selection with multiobjective evolutionary algorithms. Technical report, University of Lincoln, 2000.

[9] Dario Floreano and Francesco Mondada. Automatic creation of an autonomous agent: Genetic evolution of a neural network driven robot. In *From Animals to Animats 3: Proceedings of the Third International Conference on Simulation of Adaptive Behavior*, pages 421–430. The MIT Press, 1994.

[10] Gary B Fogel and David W Corne. *Evolutionary computation in bioinformatics*. Elsevier, 2002.

[11] Gianpiero Francesca and Mauro Birattari. Automatic design of robot swarms: achievements and challenges. *Frontiers in Robotics and AI*, 3:29, 2016.

[12] Gianpiero Francesca, Manuele Brambilla, Arne Brutschy, Lorenzo Garattoni, Roman Miletitch, Gaëtan Podevijn, Andreagiovanni Reina, Touraj Soleymani, Mattia Salvaro, Carlo Pinciroli, Franco Mascia, Vito Trianni, and Mauro

Birattari. Automode-chocolate: automatic design of control software for robot swarms. *Swarm Intelligence*, 9(2-3):125–152, 2015.

[13] Gianpiero Francesca, Manuele Brambilla, Arne Brutschy, Vito Trianni, and Mauro Birattari. Automode: A novel approach to the automatic design of control software for robot swarms. *Swarm Intelligence*, 8(2):89–112, 2014.

[14] David E Goldberg and Jon Richardson. Genetic algorithms with sharing for multimodal function optimization. In *Genetic algorithms and their applications: Proceedings of the Second International Conference on Genetic Algorithms*, pages 41–49. Hillsdale, NJ: Lawrence Erlbaum, 1987.

[15] Yoshiaki Katada. Evolutionary design method of probabilistic finite state machine for swarm robots aggregation. *Artificial Life and Robotics*, 23(4):600–608, 2018.

[16] James Kennedy. Particle swarm optimization. In *Encyclopedia of machine learning*, pages 760–766. Springer, 2011.

[17] Alan McIntyre, Matt Kallada, Cesar G. Miguel, and Carolina Feher da Silva. neat-python. `https://github.com/CodeReclaimers/neat-python`.

[18] Stefano Nolfi, Josh Bongard, Phil Husbands, and Dario Floreano. Evolutionary robotics. In *Springer Handbook of Robotics*, pages 2035–2068. Springer, 2016.

[19] Stefano Nolfi, Dario Floreano, and Director Dario Floreano. *Evolutionary robotics: The biology, intelligence, and technology of self-organizing machines.* MIT press, 2000.

[20] Michael O'Neill and Conor Ryan. Grammatical evolution. *IEEE Transactions on Evolutionary Computation*, 5(4):349–358, 2001.

[21] Mattijs Otten. Tu delft zebro swarm robotics. `http://zebro.org/`, 2018. Online; accessed 13 May 2019.

[22] Pavel Petrovic. Evolving behavior coordination for mobile robots using distributed finite-state automata. In *Frontiers in evolutionary robotics*. InTech, 2008.

[23] Carlo Pinciroli, Vito Trianni, Rehan O'Grady, Giovanni Pini, Arne Brutschy, Manuele Brambilla, Nithin Mathews, Eliseo Ferrante, Gianni Di Caro, Frederick Ducatelle, Mauro Birattari, Luca Maria Gambardella, and Marco Dorigo. ARGoS: a modular, parallel, multi-engine simulator for multi-robot systems. *Swarm Intelligence*, 6(4):271–295, 2012.

[24] Morgan Quigley, Brian Gerkey, Ken Conley, Josh Faust, Tully Foote, Jeremy Leibs, Eric Berger, Rob Wheeler, and Andrew Ng. Ros: an open-source robot operating system. In *Proc. of the IEEE Intl. Conf. on Robotics and Automation (ICRA) Workshop on Open Source Robotics*, Kobe, Japan, May 2009.

[25] Stuart J Russell and Peter Norvig. *Artificial intelligence: a modern approach.* Malaysia; Pearson Education Limited,, 2016.

[26] Michael Sipser. *Introduction to the Theory of Computation*, volume 2. Thomson Course Technology Boston, 2006.

[27] William M Spears and Diana F Gordon. Evolving finite-state machine strategies for protecting resources. In *International Symposium on Methodologies for Intelligent Systems*, pages 166–175. Springer, 2000.

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

[29] Jekanthan Thangavelautham and Gabriele MT D'Eleuterio. A neuroevolutionary approach to emergent task decomposition. In *International Conference on Parallel Problem Solving from Nature*, pages 991–1000. Springer, 2004.

[30] Jekanthan Thangavelautham and Gabriele MT D'Eleuterio. A coarse-coding framework for a gene-regulatory-based artificial neural tissue. In *European Conference on Artificial Life*, pages 67–77. Springer, 2005.

[31] Vito Trianni. *Evolutionary swarm robotics: evolving self-organising behaviours in groups of autonomous robots*, volume 108. Springer, 2008.

[32] Vito Trianni and Manuel López-Ibáñez. Advantages of task-specific multi-objective optimisation in evolutionary robotics. *PloS one*, 10(8):e0136406, 2015.

[33] Darrell Whitley. A genetic algorithm tutorial. *Statistics and computing*, 4(2):65–85, 1994.

Appendix A

Parameter Assignment

The parameters used by the EAs during the experiments can be found in the table below.

Parameter	Value
General EA Parameters (for ESMAC as wel as NN-based EA)	
Number of generations	100
Population size	100
Number of evolutionary runs	10
Elistim	2
General controller parameters	
Activation function	tanh
Aggregation function	sum
Bias mutation probability	0.7
Bias replace probability	0.1
Bias value range	[-1, 1]
Weight mutation probability	0.8
Weight replace probability	0.1
Weight value range	[-2, 2]
Standard ESMAC parameters	
Toggle enabled mutation probability	0.01
Transition boolean operator mutation probability	0.05
Add condition probability	0.2
Remove condition probability	0.2
Condition comparator mutate probability	0.5
Condition input sensor mutate probability	0.5
Condition comparator mutation probability	0.8
Condition comparator replace probability	0.1
Condition comparator value range	[0, 1]

Parameter	Value
Selector ESMAC parameters (in addition to standard ESMAC parameters)	
Selector bias mutation probability	0.7
Selector bias replace probability	0.1
Selector bias value range	[-1, 1]
Selector weight mutation probability	0.8
Selector weight replace probability	0.1
Selector weight value range	[-2, 2]
Adaptive ESMAC parameters (in addition to standard ESMAC parameters)	
Number of intial states	1
Maximum number of states	5
Add state probability	0.05
Remove state probability	0.05
Add transition probability	0.05
Remove transition probability	0.05
c_1	0.5
Compatibility threshold δ_t	0.5