ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

Scuola di Ingegneria e Architettura
Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

ONLINE ADAPTATION IN BOOLEAN NETWORK ROBOTS

Tesi in
SISTEMI INTELLIGENTI ROBOTICI

Relatore
Prof. ANDREA ROLI

Co-relatore
Dott. MICHELE BRACCINI

Presentata da
ALESSANDRO GNUCCI

Anno Accademico 2019 – 2020

Abstract

This work focuses on multiple online adaptation processes used on an autonomous robot, which is controlled by a Boolean Network; the goal is to adapt its behaviour to a specific environment and task. Results show that the robot can attain navigation with collision avoidance while also following another moving robot; it can also generalize when put in an arena different than the one used in training. With two of the tested adaptation processes the robot can express multiple phenotypes (behaviours) from the same genotype (Boolean Network's nodes and connections), thus achieving *phenotypic plasticity*. This is done by editing the coupling between the robot's sensors or actuators with the network.

Questa tesi si concentra su molteplici processi di adattamento online utilizzati su un robot autonomo, che è controllato da una rete booleana; l'obiettivo è adattare il suo comportamento ad un ambiente e ad un compito specifici. I risultati mostrano che il robot può adattarsi per navigare l'ambiente ed evitare le collisioni, seguendo inoltre un altro robot in movimento; riesce anche a generalizzare, quando posizionato in un'arena diversa rispetto a quella usata in allenamento. Con due dei processi di adattamento testati, il robot può esprimere più fenotipi (comportamenti) dallo stesso genotipo (nodi e connessioni della rete booleana), ottenendo così la *plasticità fenotipica*. Ciò si ottiene modificando l'accoppiamento tra i sensori o gli attuatori del robot con la rete.

Contents

Introduction

The context of this work is the field of Intelligent Robotic Systems, which deals with complete autonomous agents and Artificial Intelligence. A complete autonomous agent is a physical system capable of behaving autonomously in an environment without human intervention. The idea is to be able to create robots with high level capabilities such as learning, prediction and handling of failure states, planning of complex tasks and ability to deal with novelty, uncertainty and change. Thus, the requirements for robots are that they should be robust, adaptive, autonomous, situated, self-sufficient and embodied.

This thesis focuses on online adaptation. The process of adaptation is similar to the concept of *homeostasis* in Biology and, in general, it enables the system to preserve some structure, in face of changed environmental conditions; this can be done by evolution of the species, by a physiological process, by a sensory adaptation or by learning. In online adaptation, the system has to adapt while it also acts in the environment. This is important in Robotics, because the capability of reaching a goal in an unknown and changing environment is a very common requirement for systems in this field. Offline algorithms return a robot control software usually produced in simulation before the physical robot is actually deployed in the environment. These methods are inadequate when dealing with the described scenarios. The chosen robot's reference model is the *BN-robot*, which is a robot controlled by a *Boolean Network*. Building upon previous recent work [8], this thesis thoroughly analyzes a simple online adaptation process in a BN-robot that has to achieve obstacle avoidance. The objective here is to extend current results by concentrating on more (six in total) online adaptation processes and by considering a more difficult task—which is to closely follow a robot while also performing obstacle avoidance—to better understand the differences of the various techniques and solutions. The underlying automatic design methodology consists in perturbing the structure of the network or its connections to the robot and retaining those perturbations that improve the robot's utility function. The results show that some of these methods lead to robots with high performance; reasons for good and bad performance are also investigated.

The following is the structure of the thesis. Chapter 1 describes what Boolean Networks are and their uses, while chapter 2 explains the techniques used to control the robot and to conduct the adaptation process. Chapter 3 describes the goal tasks and the experimental setting. Chapter 4 examines each simulation result by visualizing the data and describing the resulting robot behaviours. Chapter 5 explains what can be done in the future to further study and do research on this topic. Finally, the Conclusion chapter contains a brief description of the conclusions of this work. Each chapter ends with a section titled "Final remarks", which summarizes the contents of the chapter and links it to the next one. Videos of some robot executions can be found in the appendix.

Chapter 1

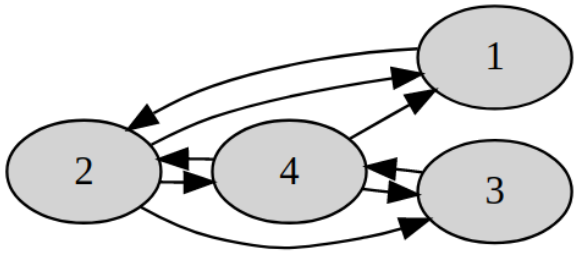
Boolean Networks in Robotics

In this chapter the focus is on the description of Boolean Networks and their uses. In section 1.1 the concept of Boolean Network is introduced, along with the most known models and the possible software design approaches of BNs. In section 1.2 there is a brief illustration about the fields of scientific research where Boolean Networks are studied; in the same section we can also find a description of the features that can be exploited to automatically design BNs and the definition of *Ensemble Approach*.

1.1 Definition and models

Boolean Networks are dynamical systems suitable to represent the dynamics of biological GRNs to many levels of abstractions and they can also reproduce dynamics of systems from different domains, such as natural, artificial and social ones [6]; *Gene Regulatory Networks* model the interactions among genes. The central dogma of Molecular Biology is that the DNA sequence of a gene is copied (*transcribed*) to create a RNA molecule, which then creates a functional product, the proteins [16]. This process is called *Gene Expression* and all steps of it can be modulated by a complex network, the GRN [5]. They can activate or repress the genes, which can allow the control of its internal and external functions, while also driving the process of cell differentiation, so that all cells in an organism can contain the same genetic material but show different phenotypes, by having different subsets of active genes [47]. GRNs can produce complex behaviours even when their description is compact, so they can be effectively used as robot programs [50]. Other than Boolean Networks, also Eggenberger's artificial evolutionary system (AES) and ANNs can be used as models of GRNs [5]. Furthermore, Dynamic Bayesian networks can also be used for this purpose [29], as well as RNNs with a Kalman filter, Directed Graphs, Petri Nets, ordinary differential equations, machine learning approaches and so on [46].

More precisely, a Boolean Network is a discrete-state and discrete-time dynamical system, structured as a directed graph (an example can be seen in figure 1.1). In such networks (conceived first by Kauffman [25]), each node in the graph computes a boolean function using the values of the nodes whose outgoing connections go to the node. The state of the network can be encoded as an array of N booleans, with N being the number of nodes in the network. Therefore, the possible network's states are 2^N.



Node 1

In		Out
0	0	1
0	1	1
1	0	0
1	1	1

Node 2

In		Out
0	0	0
0	1	1
1	0	1
1	1	1

Node n

In		Out
0	0	
0	1	
1	0	
1	1	

FIGURE 1.1: A Boolean Network with 4 nodes and K=2 (two inputs for each node). The truth tables of some nodes are also shown. The number of rows in each truth table is 2^K.

The chosen update scheme is frequently synchronous, meaning that all the node values are updated at the same time, using the values of the previous state of the network. An asynchronous update scheme would update the nodes one at a time, always using the latest state of each node to calculate the new node's state. Usually, the update process uses no noise and is also deterministic, so that the probability that noise changes the deterministic result of a boolean function is always zero.

Since Boolean Networks are dynamical systems, it's possible to analyze them by using concepts such as attractors, attractor length, basins of attraction (along with their size) and more. An attractor is a set of states towards which a system tends to evolve. In fact, these networks can exhibit a transient of states, fixed points and state cycles [18] (figure 1.2 shows an example). A transient is the ordered list of states visited before reaching an attractor. A basin of attraction represents all the states which lead to an attractor, so the state space is divided into one or more basins, based on the amount of attractors.

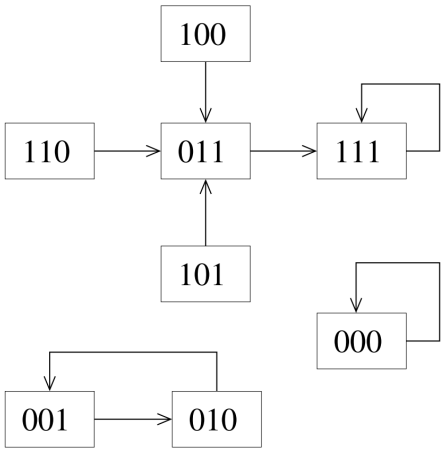


FIGURE 1.2: Example of the state space of a boolean network with 3 nodes. At the top is visible a basin that ends up in a cycle. At the bottom there is a cycle (left) and a fixed point (right). Picture taken from [50].

To better understand the properties of BNs, it's possible to convert them to different models. In particular, BNs can be the intermediate step to synthesize a FSM, which has the advantages of being compact, readable and modifiable [18]. Another possibility is to convert the BN to a *normal logic program*, which is a set of logic rules which support negation, to get a logic-based representation of the BN's dynamics [24]. BNs can also be reduced in size and simplified while preserving important dynamical properties and topological features, by using techniques such as [54].

There is a conjecture [39] about the critical regime, which is at the boundary between order and chaos: living cells, and living systems in general, are critical. Such networks show positive features, for example they can achieve the best balance between evolvability and robustness [3] and they can maximise the average mutual information between their nodes [48].

1.1.1 Random Boolean Networks

RBNs are Boolean Networks generated randomly, by specifying whether to allow self loops and the parameters N, K and p, to generate the topology and the boolean functions. It's worth noting that having a moderate amount of self-loops makes RBNs more suitable to show differentiation phenomena [7] and that increasing the amount of self loops raises the attractor count but reduces robustness and stability [35]. The parameter K indicates the number of inputs for each node, while P is the probability that each bit in the truth tables is true. To obtain critical networks, the parameters K and P must satisfy the following formula (c denotes the critical values) [13]:

$$K_c = [2p_c(1 - p_c)]^{-1}$$

1.1.2 Model variants

Various alternative models exist. The two main variation types are:

- **different topologies**: as seen in the field of *Complex Networks*, connecting nodes using different ways and distributions can result in networks with very different features. For example, *Random Graphs* differ radically from *Scale Free Networks*;
- **different updating schemes**: there are multiple ways to update the state of the network, at each time step.

Both types are examined below.

Different topologies

The following are some of the topologies that could be used in a BN:

- **Homogeneous**: the nodes are connected to each other to form a grid-like structure, with very high order;
- **Scale-Free**: a scale-free network is a network whose degree distribution follows a power law. More precisely, the number of vertices with degree k is close to $k^{-\gamma}$, where γ is a parameter that is usually in the (2, 3) range. This means that few nodes have many connections and many nodes have only a few connections. This networks are robust to accidental damages but fragile to specific attacks to hubs. Scale-free networks also have a node rank (node input count) distribution that is in the middle between uniform and skewed distributions, thus balancing the amount and length of attractors with more correlations, such as mutual information [20]. Finally, they are often related to small-world phenomena;
- **Small World**: these networks have low mean path length between nodes. They achieve the balance between random networks and grid-like networks (such as cellular automata) [56]. They also show high clustering, since two nodes connected to the same node are usually also connected to each other.

Scale-Free topologies can be used effectively in Boolean Networks, since "the fine-tuning usually required to achieve stability in the dynamics of networks with homogeneous random topologies is no longer necessary when the network topology is scale-free" [2]. Small world networks also show satisfactory results in Boolean Networks, since they "have a propensity to combine comparably large information storage and transfer capacity" [31]. Finally, when considering the network's topology, it's important to consider that "when viewed as biological decision-making networks, those with either the most uniform or the most skewed rank distributions have disadvantageous properties" [40].

Different updating schemes

Some of the most prominent updating schemes that could be used in a Boolean Network are shown here [19]:

- **CRBN**: Classic Random Boolean Network; these networks are deterministic and synchronously updated. When modelling physical or biological systems, synchronicity may not always be satisfactory, so asynchronous and semi-synchronous alternatives were created. These alternatives can result in indeterminism, if the nodes are updated in a random order. The behaviours of synchronous and asynchronous networks can vary substantially [23];
- **ARBN**: Asynchronous Random Boolean Network. At each time step one node is randomly selected and updated, therefore the updating scheme of the network is not deterministic. The indeterminism means that there are no cyclic attractors, only point or *loose* attractors (they happen when the network continuously goes through a subset of the possible states). Also, if these networks have a point attractor then there are frequently 2 or 3 of them. Finally, the basins of the point attractors usually cover most of the state space [23]. Even if these networks can't show strictly cyclic behaviour because of their lack of determinism, they can have pseudo-periodicity between states, by sufficiently relaxing statistical constraints, obtaining rhythmic phenomena, which are called *Rhythmic Attractors* [14];
- **DARBN**: Deterministic Asynchronous Random Boolean Network. These networks are the same as ARBN, except that they do not select at random which node to update. Each node has two natural values p and q, so that the node is updated whenever `t mod p == q`. If more than one node needs to be updated at the same time, then the first (in some arbitrary but deterministic order) gets updated, then the second is also updated, considering the new value of the first one, etc. These networks can have both cycle and point attractors, since they are deterministic. The advantage here is that it's possible to model asynchronous and non random phenomena, which would be difficult to do with ARBNs;
- **NRBN**: Noisy Random Boolean Network. Here the network does not always obey its deterministic update rule. In fact, each node may be perturbed with probability p during one time step, flipping its value. Therefore, these networks are not deterministic. Attractors of Noisy BNs are unstable: even with one value flip that lasts for one time step it's possible for an attractor to transition to another one [9]. Therefore, the abstractions of *Ergotic Sets* (ES) and *Threshold Ergotic Sets* were defined: an Ergotic Set is a set of attractors in which the dynamics of the network remain, while a Threshold Ergotic Set is an Ergotic Set with the added hypothesis that attractor transitions with probability less than a specified threshold are not feasible. The addition of the TES abstraction is needed, since NRBNs usually have only one ES. In fact, the idea is to associate each ES to a cell type, to model the process of cell differentiation. Since low probability transitions may never happen in the lifetime of the cell, it makes sense to neglect such transitions [55];
- **PBN**: Probabilistic Boolean Network. Given multiple good competing functions for a given gene (node), there is little reason to just choose one of them. To overcome this rigidity, the classical BNs are extended to use more than one function for each node [51]. Such a network is essentially a discrete collection of Boolean networks in which at any discrete time point, the state transitions according to the functions of one of the

BNs. Multiple versions exist: for example, the governing Boolean network can be randomly chosen at each time point, obtaining a non deterministic update rule. The result of this behaviour is that these networks are robust when dealing with uncertainty;

- **SBN**: Stochastic Boolean Network. These networks are an implementation of PBNs based on the notions of stochastic logic and stochastic computation, to efficiently represent and simulate PBNs. Stochastic computing represents continuous values as streams of random bits, so that computations can be done by simple bit-wise operations on the streams; stochastic computing was a historical failure but it may still be relevant for certain problems, in particular in low precision applications [1]. SBNs reduce the complexity needed to compute the state transition matrix of the network [30]: this is a matrix whose product with the state vector x at an initial time $t0$ gives x at a later time t;
- **GARBN**: Generalized Asynchronous Random Boolean Network, these BNs are like ARBNs but they can update any number of nodes (from 0 to all the nodes, instead of always one), chosen at random, at each time step. The nodes updated in the same time step are updated synchronously, which means that these networks behave like CRBNs when all the nodes are selected for an update, and as ARBNs when only one node is chosen. These networks are not deterministic, like ARBNs, so they also have no cyclic attractors;
- **DGARBN**: Deterministic Generalized Asynchronous Random Boolean Network. These BNs are generalized ARBNs but with a deterministic update scheme, by using again the parameters p and q, like DARBNs.

1.1.3 Code design alternatives

Boolean Networks can be implemented in at least two different ways:

- **Object Oriented way**: the network is modeled as a class with one main method, with the simple following signature (in pseudocode):

  ```
  (input) => (output)
  ```

  so that the network's users don't have to handle the class states: they simply provide inputs and receive the outputs. This can be done by encapsulating inside the class the notion of network state and of input and output nodes. This design makes the class simple to use but it has a few flaws, as described below;
- **Functional Programming way**: the network is thought as a simple function with the following signature (in pseudocode):

  ```
  (state) => (new_state)
  ```

  Therefore, to use it it's required to manually write the current input into the input nodes, then pass this modified state to the network's function, obtain the new network's state and extract the output from that state. To do this, the current state also has to be saved somewhere and updated at each iteration. This function is deterministic and "functionally pure", meaning that it doesn't use mutable state and that it doesn't

use any side effects. This can avoid bugs with state handling and it also enables the use of these networks by applying them to a list of states to calculate their next state (this would be complicated to do with the OOP design). For example:

```
new_states = current_states.map(state => boolean_network(state))
```

The problem with this design is that it does not provide a simple `(input) => (output)` method like the OOP alternative. To solve this, the network can be wrapped in a "BooleanNetworkRunner" class that contains the current state and that knows what nodes are input (and output) nodes. This functional design has good *Separation of Concerns* and relegates the state handling in a small and easily understandable class. It is also possible to avoid using classes and mutable state by implementing BooleanNetworkRunner as a tail recursive function, in the following way (pseudocode):

```
@tailrec
def run_boolean_network(network, network_state) {
  new_inputs <- get_inputs()
  updated_state <- force_inputs_in_state(network_state, new_inputs)
  new_state <- network(updated_state)
  outputs <- extract_output(new_state)
  // use here the output as needed
  run_boolean_network(network, new_state) //tail recursive call
}
```

Here instead of mutating the same variable, the function continuously passes the new network's state to the next recursive call, thus avoiding mutable state. This example can also be expanded to use a `output_consumer` function as a parameter and to stop the recursion after a certain amount of steps.

1.2 Uses of Boolean Networks and research

1.2.1 Fields of research

Boolean Networks are currently one of the research subjects in Robotics and Biology. The two fields have different goals but BNs have useful properties for both. The goal in Robotics is to use BNs as the main component in the control software of robots, to design robotic and multi-agent systems that show the positive features found in biological organisms; this is because of the richness of dynamics, along with adaptiveness and robustness that BNs show [50]. For Biology, BNs are a computational model that could provide insights into the overall behavior of GRNs [22].

1.2.2 Automatic design of RBNs

Metaheuristic optimization algorithms such as Evolutionary Algorithms can be used to automatically design RBNs which are optimized for specific target requirements [50]; this can be done to get the network to perform some behaviour or to have a series of positive features, such as having the right number of attractors, having attractors with a specific length

or basin size or having capabilities such as self-organization, robustness, adaptability, evolvability, etc.

Multiple factors of BNs can be exploited by engineers or natural selection to guide their self-organization, other than simply modifying the parameters P, K and the topology (along with link distribution, regularity and modularity) [20]:

- **canalizing functions**: they are boolean functions where at least one input (called a canalizing variable) is able to determine the output, regardless of the values of the other inputs [52]. This means that the non-catalizing inputs are ignored, so removing them does not affect the dynamics of the BN. RBNs with nested canalizing functions have stable (ordered) dynamics, approaching the critical regime when setting K to low values [27].
- **silencing**: single genes can be switched off; for BNs, this means keeping the value fixed for a subset of nodes, which can be assumed to make the dynamics more stable;
- **redundancy**: having redundant nodes prevents the propagation of the effects of mutations, thus increasing neutrality [37];
- **degeneracy**: it's "the ability of elements that are structurally different to perform the same function or yield the same output" [15], which helps when dealing with failures, increasing robustness [57].

1.2.3 Ensemble approach

BNs can be useful when used with the *Ensemble Approach*, proposed by Kauffman [26]. It is a way to find models with the same properties as the system of interest, by imposing constraints on the parameters used to create the instances of the model. This concept can be used on BNs by setting constraints on their topology or on the boolean functions, thus generating different ensembles, to find out the ones with features similar to organisms or real cells. One example is the critical ensemble of RBNs, which can show features similar to the differentiation process [12, 10].

1.3 Final remarks

Boolean Networks are a model for GRNs. In particular, BNs are a dynamical system, with multiple nodes in a directed graph, with each one computing a boolean function. Random BNs can be generated by choosing the values for the parameters N, K and P. Multiple model variations of BNs exist, by using specific topologies or updating schemes. BNs can be effectively used as robot programs and they are currently the subject of research in both Robotics and Biology. RBNs can be automatically generated by exploiting features such as topology, canalizing functions, silencing, redundancy and degeneracy; also, the Ensemble Approach can be used to find models matching the system of interest.

The next chapter focuses on how the RBNs are used to control a robot in this work and how the adaptation process occurs.

Chapter 2

Robot Control and Online Adaptation Process

This chapter explain the mechanisms and techniques used to control the robot and to conduct the adaptation process. The simulated robot uses a *Boolean Network* (BN) as the main control mechanism, by connecting this network to robot's sensors and actuators. The online and automatic adaptation processes tested here modify the network itself or the coupling between the network and the robot, to obtain a robot controller with a high evaluation, so that it can reach the target requirement of the chosen task.

Section 2.1 lists some of the seminal works that are currently the state of the art regarding online adaptation; section 2.2 explains how the Boolean Network interacts with the robot, section 2.3 shows the automatic design methodology used and finally the adaptation processes that are tested in this work are examined in section 2.4.

2.1 State of the art of online adaptation

Adaptation has the goal of keeping some structure from changing because of environmental conditions, similarly to the concept of *homeostasis* in Biology; in the fourth chapter of [41] it is said that "if an agent is to sustain itself over extended periods of time in a continuously changing, unpredictable environment, it must be adaptive", while also writing that adaptivity and intelligence are directly related. In the same book it is said that adaptation can happen by evolution of the species, by activating a physiological process (like sweating in humans), by sensory adaptation or by learning. Online adaptation is a special case of adaptation, where the robot has to simultaneously adapt and act in the environment, to achieve its goal. Here, the adaptation process never stops.

Relevant works which list the possible techniques that can be used are [17] for robot swarms and [36] for single robots. Regarding online adaptation, the former paper lists techniques such as distributed population-based algorithms and cultural evolution using imitation-based algorithms. The main aim of [36] is to show the *(1+1)-restart-online algorithm*, which is a variation of an online and on-board Evolution Strategy adapted to autonomous robots. This algorithm solves or mitigates problems such as premature convergence caused by the singleton population (by changing the mutation step size on the fly), the penalization for bad behaviour that is actually caused by the actions of the previous solution (by using a recovery

time, during which the robot is not evaluated), misleading evaluations caused by unlucky conditions (by reevaluating the current best with a certain probability) and getting stuck in local optima (by restarting the algorithm with similar or different parameters whenever the search is stalled) [36].

The current work is based on the results obtained from [8] and uses it as a starting point: it deals with adaptation by rewiring the inputs of the BN with the robot's sensors, thus achieving *phenotypic plasticity*, since it's possible to obtain different phenotypes by using the same genes.

2.2 Coupling between BN and robot

The BN is used here as the center of the control software of the robot. To do this, we choose some nodes of the network and set them as input nodes, then the same is done for output nodes. The value of each input node is then overwritten at each time interval with the value of the robot's sensor readings (usually by applying a threshold to binarize the values), while the output nodes are read and their values are used as input for the robot's actuators. The inputs can be processed in various ways before being read by the network and so can the output values, before being passed to the actuators (as seen in figure 2.1).

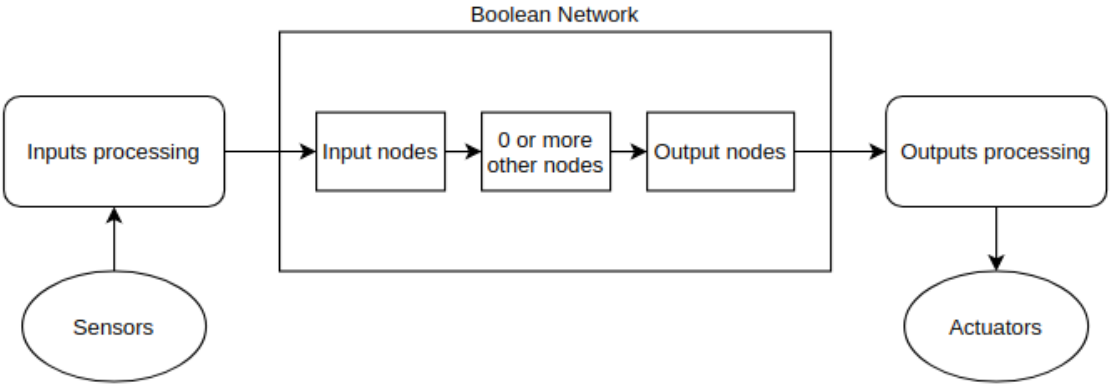


FIGURE 2.1: The coupling between the BN and the robot, as seen in [18].

For this work, the proximity sensor readings are aggregated before passing them to the BN: this was needed to test small networks with only 25 nodes or less using multiple sensors (24) all around the robot. More precisely, 6 groups of 4 adjacent sensors each (as shown in figure 2.2) are defined and then for each group the maximum value is calculated. The binary output values is then converted by multiplying them by the constant `SPEED`.

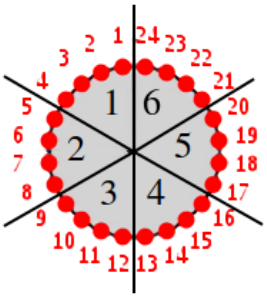


FIGURE 2.2: The six groups of the proximity sensors are here depicted. This is enough to detect obstacles and differentiate between front, back, left, right, etc.

The Range-and-bearing sensor is also used to create 6 virtual binary sensors, all around the robot: each one is enabled if another robot is sending some data near the main robot and is in the direction of that sensor.

2.3 Automatic design methodology

The design problem is treated here as an optimization problem: the goal is to find the parameters which give the highest evaluation. A search algorithm is used to generate new sets of parameters to test, by looking at how the previous solutions performed in the simulation, as reported by the evaluation function [18]. An overview of this is visible in figure 2.3.

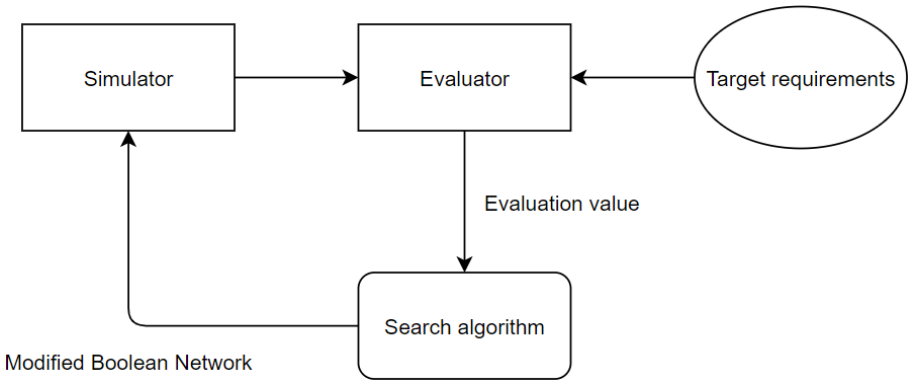


FIGURE 2.3: The design methodology implemented, as seen in [18].

The resulting robot controllers are only tested in simulation: this is enough for the purposes of these experiments but not if the goal is to achieve satisfactory results in the real world. This is because of the *reality-gap*, which is the discrepancy between simulation and reality. Since the robot modifies its network configuration while operating in the environment, this is an *online* design methodology.

The search algorithm used here is described by the following steps:

1. the current best evaluation is set to zero;
2. a new network configuration is randomly generated;
3. if the evaluation function of the new configuration is greater than the previous best, then the new configuration becomes the best;
4. a new configuration is created by applying the chosen adaptation process on the current best configuration;
5. restart from step 3 if the ending conditions are not met.

This algorithm keeps and modifies the current best configuration to do *exploitation*, otherwise it would be a simple random search, which does only *exploration*. The balance of the two is important to quickly identify regions in the search space with high quality solutions and not to waste too much time in already explored or low quality regions [4]. The specific adaptation processes used in step 4 are listed in the next section (2.4).

2.4 Adaptation processes

An adaptation process is a strategy used by the previously described (section 2.3) search algorithm to generate new network configurations from the current one, trying to increase the maximum evaluation reached and therefore adapting the robot to a specific environment and task. In the following sections the different "families" (classes) of such processes are explained and examined. These processes are small and simple, requiring little computing power, so that they can be done in very small hardware. Similar strategies could be used in the future in the fields of *Synthetic Biology* and *Biorobotics*: the vision is to have a micro and hybrid robot, also made of biological components, trying to accomplish a mission in environments where human exploration is not possible such as oceans, human and animal bodies, etc [49].

2.4.1 Network editing

This family of adaptation processes try to modify the network itself to achieve the goal. This means that both the nodes' boolean functions and the network topology can be subject to change.

Boolean function bit flips

A certain number of bits in the truth table of the boolean functions will be flipped, changing a 0 to a 1 or vice versa. This can modify the type of dynamics observed, making them more ordered or disordered.

Input changes to nodes

Some nodes are selected and for each of them one of their inputs is chosen and overwritten, by selecting another node randomly. This new node's input can't be the node itself, though,

if the network has been created to avoid self loops. These changes keep constant the K parameter of the network, in fact, after the editing attempt, each node still has the same number of inputs as before.

2.4.2 Editing of the BN - robot coupling

This family of adaptation processes modify the way that the network interacts with the robot instead of modifying the BN itself. They are thus less dependent on the type of network used and they also provide a way to achieve phenotypic plasticity, as described below.

Network's inputs rewiring

A certain amount of the BN's inputs is changed, by designating different nodes for this role. The total number of input nodes does not change. The output nodes cannot also be selected as input nodes. Since this adaptation process can achieve behaviours such as obstacle avoidance, the robot can express a phenotype among many that is suited for one specific environment, while being characterised by only one genotype (the BN's nodes and connections) [8].

Network's outputs rewiring

This is the same as the previous case but applying it to output nodes instead of input nodes. The network's input nodes cannot also become output nodes.

2.4.3 Other adaptation processes

Random BN generation

To better understand the results of the other adaptation processes, it's important to have a baseline. This is done by creating a new RBN each time, instead of making small changes. The network-robot coupling is also randomized each time. Adaptation processes that give worse or similar results compared to this one should be considered as poorly performing, if the task is hard enough. As the network gets bigger the search space does too, so it should be more and more difficult to find good solutions using this strategy, since it does no exploitation.

Incremental adaptation scheme

This adaptation scheme adds nodes to the network. More precisely, each edit attempt can be done in one of three ways and the choice between them happens by drawing a uniformly random value, giving the same chance to each:

- rewire of a BN's input;
- rewire of a BN's output;
- insertion of a node into the BN.

The first two are the same as described before (in subsection 2.4.2). When adding a node, called H, this scheme randomly chooses another node already in the BN, called Z. The boolean function of H is copied from Z, while the number of inputs of H is K and the number of outputs should follow Poisson's distribution, since that's what happens in RBNs with a constant K value. To obtain a value from that distribution, *Knuth's algorithm* is used. Finally, the inputs and outputs of the new node are chosen randomly between the other nodes. Since the output nodes of H already have K inputs, one of those inputs gets rewritten with H. The following is Knuth's algorithm [28, 44]:

```
init:
  Let L ← e^−λ, k ← 0 and p ← 1.
do:
  k ← k + 1.
  Generate uniform random number u in [0,1] and let p ← p × u.
while p > L.
return k - 1.
```

2.5 Final remarks

The robot uses a BN as the main control mechanism, with the goal of maximising a task-dependent evaluation function. The proximity sensors are aggregated into 6 virtual sensors, which is the same amount of virtual sensors created for the RAB system. Using a robot simulator, the robot itself calculates the evaluation of each solution created by the employed search algorithm, which tries to find the solution with the maximum evaluation. The search algorithm simply remembers the best solution found yet and modifies it to try to improve it, by using one of the many possible online adaptation processes. These simple processes (which can be used in miniaturized hardware) can edit the BN itself or the wiring between the BN and the robot's input and output. Different and more complex search algorithms can also be used.

The next chapter shows how the experiments are conducted and how each parameter is used.

Chapter 3

Experimental Setting

This chapter shows a description for each task, along with the details and the precise settings used in the experiments. Section 3.1 describes the simulator used and its settings, section 3.2 shows the chosen robot and how it is used, section 3.3 contains information about the arena and obstacles used in the experiments, section 3.4 lists all the goal tasks that the robot has to learn by online adaptation, section 3.5 displays the chosen programming languages, libraries and other tools, while section 3.6 deals with the parameters of BNs and the values used for them. Finally, section 3.7 describes how the different adaptation processes are set.

3.1 Simulator

The simulator used is **ARGoS** [42]: it is a general-purpose and multi-physics robot simulator. It can simulate large-scale swarms of robots and it can be customized by adding plug-ins. Its requirements are high accuracy, flexibility and efficiency [53]. This choice was also made because of the ease of use and the know-how already available for this particular simulator.

This is how the simulator is set:

- each simulation with networks of 25 nodes lasts 3600 seconds while networks of 100 nodes require 4 times as much (therefore 144 seconds for each node, in general), so that there is adequate time to explore the bigger search space;
- the speed in ticks per second is 10;
- the noise (parameter `noise_level`) of the proximity sensors and wheel actuators is enabled and set to 0.03 for both in almost all the experiments (except for the first ones). This is usually needed to dampen the detrimental effects of the reality-gap problem and it also makes the task harder for the robot, but it's worth noting that a high quality controller can exploit the noise to achieve better results;
- `random_seed` is set to a constant value, so that the robot always starts in the same position;
- the range of the Range-and-bearing system is set to 1m and no noise or packet loss functionality is enabled for it.

3.2 Robot

The chosen robot is *Foot-bot*, which is already available in Argos (and visible in figure 3.1). It's a differential drive robot capable of moving on the ground; the features used here are the 24 proximity sensors evenly placed along its circumference, the two motorised wheels and the Range-and-bearing system. It also has light sensors, multiple RGB LEDs, a gripper, wheel sensors and ground sensors.

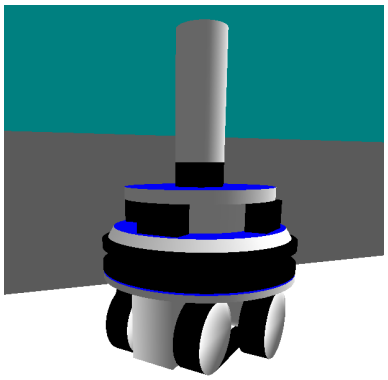


FIGURE 3.1: The robot used in the experiments, as seen in the Argos UI.

The velocity[1] of the wheels that are activated by the BN is 15. The values obtained from the proximity sensors are between 0 and 1 and the threshold used to convert them to boolean values is 0.1, meaning that values above it become `true`, while the others become `false`.

3.3 Environment and arena

The arena is a 4m by 4m square, it's delimited by walls and has a central 1.25*1.25 meters squared obstacle (an immovable box). Since the walls' thickness is 0.1 meters, the area inside the walls is $3.9m * 3.9m = 15.21m^2$ and the area of free space is $15.21m^2 - 1.25m * 1.25m = 13.6475m^2$. This means that the area of the central box is less than a ninth of the area of free space. The arena is shown in figure 3.2.

[1] If both wheels have their velocity set to 15, then the robot will move forward at 15 cm/s.[43]

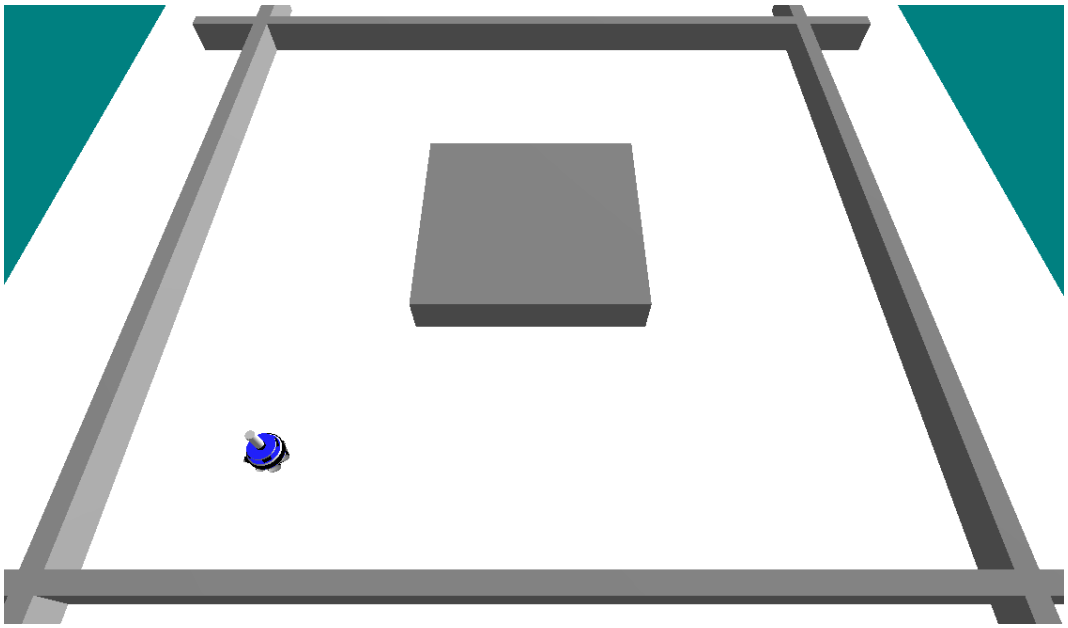


FIGURE 3.2: The arena used in almost all the experiments.

To test the generalization capabilities of the solutions, the box in the middle is swapped with a number of randomly placed smaller obstacles. The robots have never seen this arena during training but they should be able to complete the goal task anyhow. This arena is shown in figure 3.3.

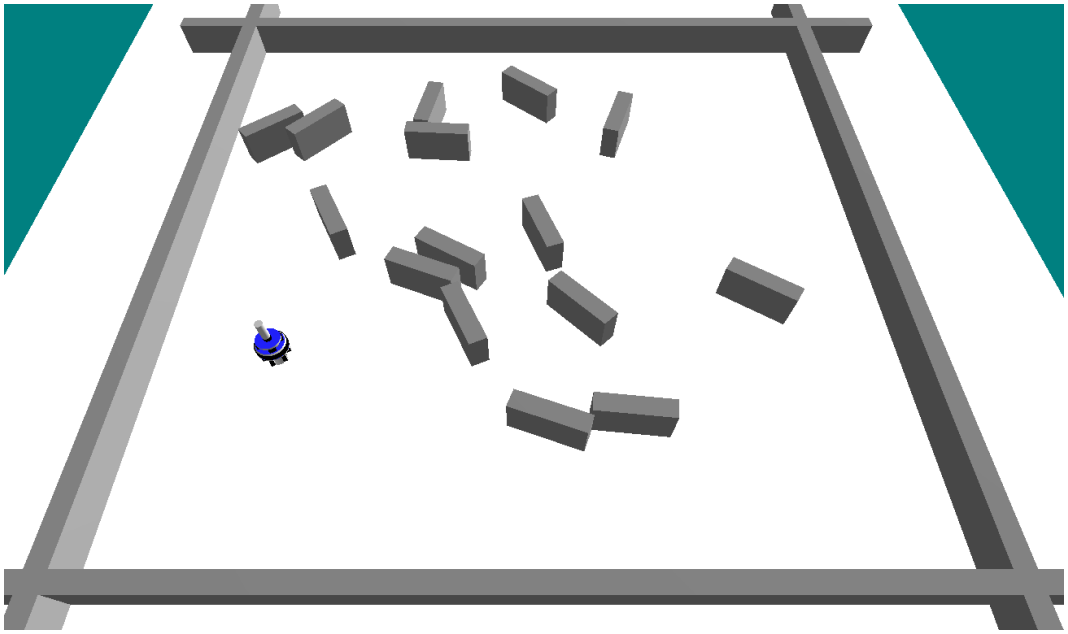


FIGURE 3.3: The arena used to check the generalization capabilities of the solutions.

3.4 Tasks to learn and evaluation functions

All the tasks listed here have an evaluation function that can be computed by the robot itself, since the function uses only the information that the robot has. This is a positive property, since it enables the robot to autonomously adapt and work in an unknown terrain, without the need for an external evaluator that calculates the current performance level.

3.4.1 Obstacle avoidance

The goal is to get the robot to circle around the box as quickly as possible[2], avoiding all collisions. It's a simple task but it should be hard enough to study the characteristics of the adaptation processes and how they compare to each other. It that's not the case, more advanced tasks are needed. The output of the evaluation function is accumulated along the execution steps and then normalised, obtaining a value in the range [0-100]. The evaluation function is defined as follows [38]:

$$F = (1 - pmax)(1 - \sqrt{|bl - br|})(bl + br)/2$$

Here `pmax` is the maximum value returned by the proximity sensors whereas `bl` and `br` are the binarised values used to control the left and right motor, respectively. The first factor rewards movement far away from any obstacle, the second one punishes movements that are not straight and finally the third factor rewards high speed. Since bl and br can only be 0 or 1, when the robot is turning the evaluation function returns zero. This means that scores of 100 or near that value are not possible, since the robot has to turn multiple times to be able to move forward for a long time. More precisely, with the previously described arena the highest evaluation function value observed is 92 without sensor and actuator noise and 90 with noise enabled.

3.4.2 Robot following behaviour

This task requires to stay close to another robot that moves erratically, without bumping into it and while also doing obstacle avoidance. This resemples a simplified version of *Prey Capture*, which is a special case of a class of problems known as *Pursuit and Evasion* [21]. In Pursuit and Evasion there are a predator and a prey in an environment. The predator moves to try to capture the prey while the prey attempts to escape and flee from the predator. These kind of scenarios are interesting for the study of adaptive behavior because they happen frequently in natural ecosystems. Also, their objective is relatively simple but they require complex sensory-motor coordination with respect to the environment and another moving entity [34]. This task does not require the use of memory to reach satisfactory results, but the agent can detect the prey only within a certain distance, so if the prey moves out of the sensor's range it can be helpful to remember where the prey was and predict where it will be in the future, to quickly catch it. The hope here is to drive the system towards the emergence of new strategies.

[2]The trail left does not need to make an actual circle.

The maximum speed of the "prey" is equal to the one of the main robot, so that a good solution can continue to follow the prey without losing it. This makes the problem more manageable. Its movement is programmed so that it's not a simple Brownian motion, by setting the speed of the wheels to new random values in the [0, 15] range every 40 steps. The mechanism used here to detect the prey is the Range-and-bearing (RAB) system: this system allows robots to perform localized communication by broadcasting data in a limited area, but only if they are in direct line of sight. This system returns for each received message the data itself, the angle relative to the x axis of the receiving robot, the angle relative to the robot's xy plane and the distance from the message's source [43]. The "prey" robot continuously broadcasts the value "1"; by using the angle of the received messages, 6 virtual boolean sensors are created to be used on the main robot, so that the BN can roughly know the direction of the prey. If a new message is perceived, then one of the 6 sensors activates, based on the angle of the sender.

The evaluation function is defined as follows:

$$F = F1 * 1/3 + F2 * (2/3)$$

$$F2 = \begin{cases} 1 & \text{if } 5cm <= robot\_distance <= 100cm \\ 0 & \text{otherwise} \end{cases}$$

Here F1 is the evaluation function for obstacle avoidance, as previously described, while F2 is the "follow" component. The final evaluation function was defined by trial and error. Initially the if condition only stated "if $robot\_distance < 100cm$" and the weights of F1 and F2 were both set to 0.5. F2 has to be favored because otherwise it would be too easy to lose track of the prey; this means that the robot following behaviour is usually harder than obstacle avoidance; since the automatic design methodology is lazy it prefers the easiest route to high evaluation values and therefore, without proper weights it would simply ignore the F2 component. To better achieve both F1 and F2 the if condition also has to be modified as shown, by adding a requirement to keep a minimum distance from the prey robot.

By using just one prey robot, it's possible that the predator and prey are very far from each other. In this case, the predator can only learn to accomplish the obstacle avoidance task. This behaviour can then lead the predator to the prey, so that it can also learn to follow it. The problem is that small enhancements in the network that would improve the chasing behaviour get discarded whenever the two robots are far away from each other, thus wasting simulation cycles. In an attempt to reduce this problem, in some experiments another prey is added. The two preys should stay in different areas of the arena; to achieve this a simple implementation is chosen, by using the RAB system to detect each other, with its range set to 2 meters:

```
function step()
  if(far_from_other_prey()) {
    move_randomly()
  } else {
    robot_angle <- get_other_prey_angle() //using 2nd bit of RAB
    if(is_on_left(robot_angle)) go_right() else go_left() //escape
  }
```

The resulting behaviour is a simple form of surveillance of an area. Obstacle avoidance could also be added, but it's not considered necessary for these experiments. It's possible to add even more preys but it would probably be problematic: the main robot could simply learn to move around the arena while doing obstacle avoidance, since it would be close to at least one prey from time to time, thus obtaining a high evaluation value without actually following any robot.

3.5 Scripting environment

3.5.1 Programming languages

The programming language used is **Lua** 5.3: this is because in ARGoS the only supported languages are C++ and Lua. Lua is a lightweight dynamic-typed embeddable language. Languages that compile to Lua were also considered but ultimately were discarded.

Lua is the chosen language because it's higher level than C++ and because it's garbage-collected, so that the researcher can focus more on the problems at hand, without having to deal with memory management, memory leaks, undefined behaviour (this can still happen but to a lesser degree), segmentation faults, etc. The major downsides of Lua are the lack of type safety and a reduction in execution speed. Also indexing generally starts at index 1 instead of 0, the standard library is relatively small and the language lacks multiple features found in many other languages, such as classes, lambdas, stream APIs, etc. Furthermore, the ARGoS APIs that can do things such as modifying the arena during the experiment and read information regarding the arena and the other robots are available only in C++. The most important data structure available in Lua is the table, which can be used both as an array and as a hashed map (dictionary).

3.5.2 Libraries

The following libraries have been used to enhance the language or add new features:

- **Inspect**: creates a human-readable representation of Lua tables. This is used to encode the BNs found during the adaptation process, so that they can be saved in a file on a single line;
- **Lambda**: gives terse lambda definitions (using strings), to make the language less verbose. For example `l("(a,b) -> a+b")` is the same as `function(a, b) return a+b end`. Such functions can even use global variables but can't use other variables outside of their scope, meaning that they are not *closures*. It's also worth noting that using globals inside these lambdas should be discouraged, since they are written as strings and IDEs do not change strings when renaming variables or functions, so bugs can happen. It is therefore best to use them only when a pure[3] and simple function is needed, like `_ + _`[4];

[3] Meaning no side effects, as in functional programming.

[4] This is equal to `(a,b) -> a+b`.

- **Lua-Collections**: a collection class based on Laravel collections. This lets the programmer use *stream APIs* like `map`, `filter`, etc. It has been modified to also support other methods inspired by the Scala's stdlib like `reduce`, `foldLeft`, `foldRight`, `zipWithIndex`, etc. This enables the developer to use less variables and less loops, making the code more functional;
- **Middleclass**: a simple OOP library, to create classes and objects and better organize the project;
- **Pprint**: a pretty printing library, mainly used to debug.

Libraries to add the features of *For-comprehension*[5] and *Pattern matching* have also been evaluated but were ultimately not used.

3.5.3 Other tools and features

The *IntelliJ IDEA* IDE was used with *EmmyLua* plugin, adding the compatibility with Lua source code and enabling the type safety inspections provided by the plugin. This plugin reads the type information that the programmer wrote in the comments and creates visual warnings when the program fails to type check, adding a simple form of type safety.

A utility library was also written to add multiple features not found in the standard library. The most important are shown here:

- throw an error whenever some code uses an undefined global variable, instead of failing silently, by editing the metatable of the `_G` table;
- deep copy functionality for tables;
- range function that returns an array containing the specified range of values (for example "from 2 to 10 with step 2"). This is usually used as a starting point with the stream API, for example

  ```
  collect(range(1, 10)):random(5):all()
  ```

  returns 5 numbers between 1 and 10, with no repetition;
- functional "if" function (that is unfortunately not lazy, so it always calculates both branches) which evaluates to the value of the chosen branch;
- functions to create arrays or matrices by simply specifying the size and the "generator" function that will be called to create the value for each cell. This is useful as a replacement for the missing `fill` function (which creates an array with the specified number of the same specified value) and to create more complex arrays in a terse format. For example, an identity 5*5 matrix can be simply created with the following code:

  ```
  create_matrix(5, 5, l((row, column) -> my_if(row==column, 1, 0)))
  ```

The code regarding the BNs is written in a simple but highly customizable class, to simplify its usage. Finally, most of the code is tested using a straightforward form of unit testing, by

[5] The PenLight library has this feature.

using the "assert" keyword. This is done to avoid regressions and bugs in the code caused by making changes.

3.5.4 Automated simulation execution

A **Bash** script is used to automate the execution of multiple runs. The results are written to text files, so that they can be analyzed in the future. For each run, the script saves the best network, its evaluation value as well as all the history of such values and the history of states of the first and final BNs. Since different runs require to change some parameters of the BNs (for example the bias for the creation of the truth tables), this script modifies the code of the simulation itself to update such parameters between runs. Each configuration with the same parameters is tested 50 times.

To quickly get the results, multiple ARGoS instances can run at the same time, in parallel, to use all the CPU cores of the machine. There are also other ways to obtain parallelization, but this one needs no changes in the simulation code: it can be done simply by copying the project's folder multiple times, then by making changes to each one's parameters as needed (so that each instance performs a different simulation) and finally running one ARGoS process for each copied folder, each one in a separate Linux shell.

The final box plots and XY charts are created using the Scala language with the XChart library, by reading the previously created files.

3.5.5 Code optimization

The following techniques were used to optimize the code and ultimately to obtain the results more quickly:

- a profiler, to check which parts of the code need more tuning. In particular Lua-profiler is the chosen library;
- avoiding globals: globals are stored in a table (hash map) [45] and accessing one of its elements is quite slower than accessing a local variable [33]. To solve this, global functions can be localized when deemed necessary, in the following way:

```
local min = math.min
```

- avoiding the `ipairs` function when possible, since it is slower than a simple for loop that uses the length operator for the stopping condition [33];
- avoiding the `table.insert` [33] and `table.remove` functions, because of their speed. Instead, insertion can be achieved by simply setting the table's element at the specified position (or key) and removal can be done by setting the specified position in the table to nil, if there is no need to also shift the elements;
- avoiding frequent table (or function) creation, by caching when possible. This helps to reduce the amount of work to do and it also creates less *garbage*, so that there is less need for the action of the garbage collector;

- changing the whole implementation to reduce the algorithmic cost of a specific function, in terms of *Big O Notation*.

3.5.6 Best practices

A series of coding best practices is used to raise the quality level of the project, to avoid bugs, facilitate development, maintenance and enhancement of the code, for anyone who may use it in the future. The following are the adopted best practices:

- general best practices such as descriptive names, avoiding acronyms, writing Unit Tests, reducing dependencies, Don't Repeat Yourself principle, etc;
- write functional code when possible, reducing mutable state and side effects to avoid possible bugs. In fact, the ease of *equational reasoning* enabled by *referential transparency* makes code editing easier and less bug prone;
- avoid mixed tables: a mixed table is a table that contains both hashed values and numerically indexed values, which can be the cause of misuse and mistakes, in particular when iterating over them; an example of such tables is the following:

```
local mixed_table = {1, foo = "bar" };
```

- pay attention when using the # length operator, since in Lua the length of a table is only defined if the table is a sequence [32].

The code quality level was not the first priority, though, since the main goal of this work is to do research, not to build a lasting software product.

3.6 Boolean Network parameters

The BNs used in this work are all Classic Random Boolean Networks. The parameters used when creating a RBN are the following:

- **K**: the number of input arcs for each node. This is set to 3;
- **P**: the probability ("bias") that each bit in the truth tables is true. The values tested are 0.1, 0.21, 0.5, 0.79 (it's the critical value with K=3) and 0.9. If the *Dual encoding*[6] was used, the critical value would be 0.21;
- **N**: the number of nodes. The values 25 and 100 are the most used, but bigger networks are recommended to reduce the risk of creating networks behaving like Braitenberg vehicles[11]. Such robots would work very well at achieving such a simple task, obtaining high evaluation function values but preventing us from learning about the differences of the adaptation processes tested here;
- **self_loops**: specifies whether the network topology can have nodes connected with themselves. This is set to false;

[6]With Dual encoding, an input value of 1 would set the input node to 0 and viceversa; the outputs of the network would also be inverted before passing them to the actuators.

- **override_output_nodes_bias**: specifies whether the bias of the output nodes should become 0.5 or be left as it is[7]. This functionality is enabled, to avoid behaviours that move too little when using low bias values and behaviours that keep the actuators too active with high bias values.

Also, the nodes used as inputs are 6, which is equal to the amount of sensor zones defined in the previous chapter (section 2.2); since the robot used only has two wheels, the nodes used as outputs are two.

3.7 Adaptation process parameters

Each adaptation process was set to apply the minimum amount of change to the BN. This means that the amount of network input rewires is only one at each editing attempt and the same goes for network output rewires, function bit flips, etc. The only exception to this is the random editing process, which changes the whole network each time to provide a baseline for the other results. This decision keeps the adaptation process simple and as a result the different adaptation strategies can also be more easily compared. Also, each network is tested for 1200 steps before it gets edited; using 10 ticks each second results in $3600 * 10/1200 = 30$ network edits for BNs with 25 nodes and 120 edits for BNs with 100 nodes.

3.8 Final remarks

The simulator used is ARGoS. Bigger networks have more time to adapt, because of the larger search space. Noise for sensors and actuators is used to make the task harder. The chosen robot is Foot-bot. The arena is squared, with an immovable box in the center. The evaluation functions of the tasks can all be computed by the robot, so that it can autonomously adapt. The tasks to learn are obstacle avoidance and robot following. The programming language used is Lua, with multiple third-party libraries and tools. Bash and Scala scripts are used to automate the simulation executions and to create plots and charts. Multiple techniques are employed to optimize the code and increase its quality level. The BNs have nodes with three inputs, no self loops and their output nodes have their bias set to 0.5.

The next chapter will exhibit and analyze the results of the experiments, while also showing the obtained plots and charts.

[7] More precisely, if this parameter is set to true then a randomly extracted half of the bits in the truth tables of the output nodes become true, while the other bits of those tables become false.

Chapter 4

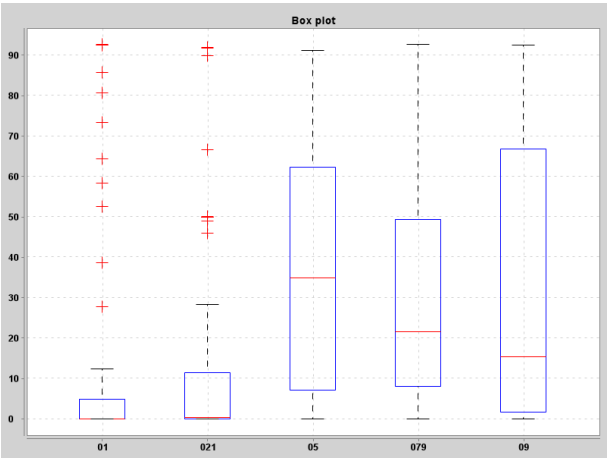
Results

In this chapter the results of each experiment are discussed, checking whether the expected behaviour appeared and if there is any unexpected phenomenon. In that case, one or more possible explanations are proposed. The simulation batches are listed in chronological order: section 4.1 refers to the experiments using obstacle avoidance as the goal task, 4.2 conveys how well the adapted robot can generalize while doing obstacle avoidance, 4.3 introduces the robot following task and finally 4.4 shows the robot's ability to generalize when following another robot. For each configuration, 50 RBNs for each bias are generated. This means that each box in the box plots sums up the results of 50 different runs.

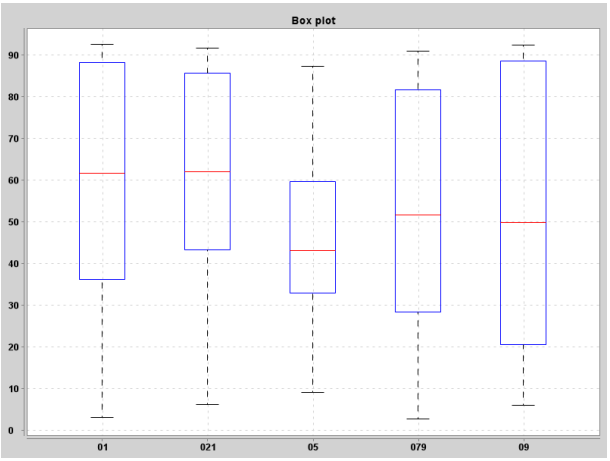
4.1 Obstacle avoidance

4.1.1 Networks with 25 nodes

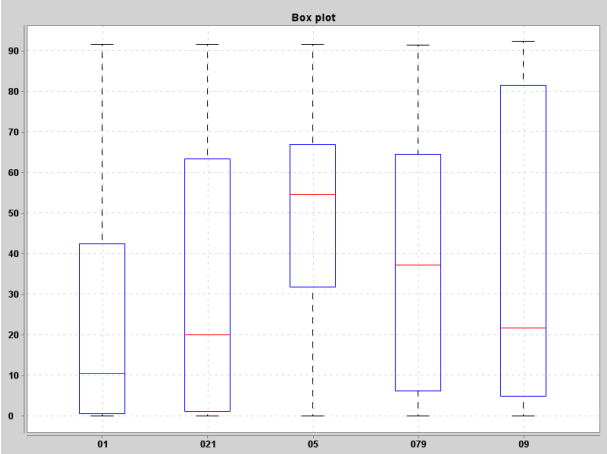
The simulations were run for 3600 seconds, so there were 3600*10/1200 = 30 editing attempts for each network.



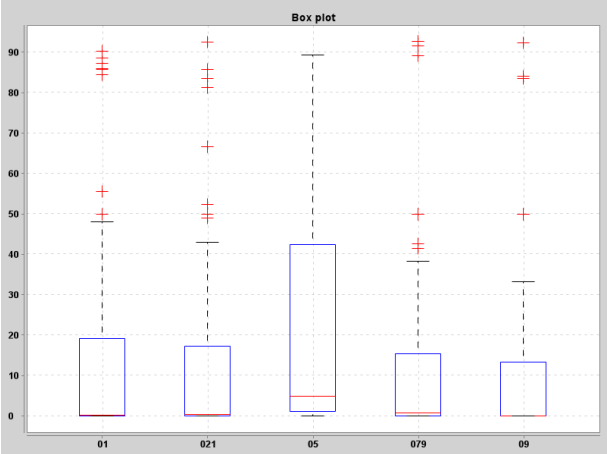
(1) Evaluation box plot for "Input rewire".



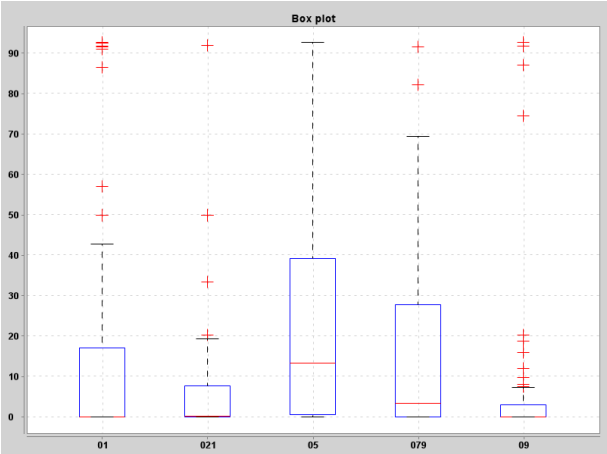
(2) Evaluation box plot for "New network each time".



(3) Evaluation box plot for "Output rewire".



(4) Evaluation box plot for "Boolean function bit flip".



(5) Evaluation box plot for "Node input swap".

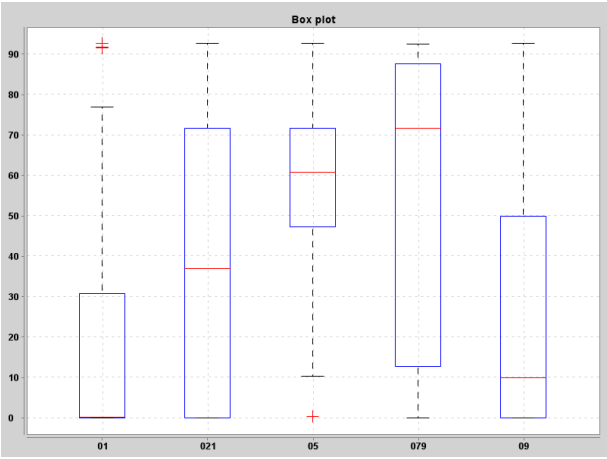
The "New network each time" experiment in figure (2) gives the best results of all the experiments. The reason for this is probably that the search space is relatively small for networks of this size and the task is not very difficult (the behaviour "if front-right obstacle then go left, else go straight" is enough to get a high evaluation value), so the probability of randomly finding a good solution is high enough that random search becomes viable. It's also interesting that in this experiment the chaotic networks perform worse than the other ones.

The "Output Rewire" experiment in figure (3) also achieves decent results, particularly with bias set to 0.5. The "Input Rewire" experiment in figure (1) shows a similar but lower distribution. The chaotic networks perform better than the other ones in all the experiments of this batch, except for the random search. This is interesting because usually the critical networks (bias 0.79) are the ones with the best results. In subsection 4.1.3 we'll see that this outcome might change when noise is added to sensors and actuators.

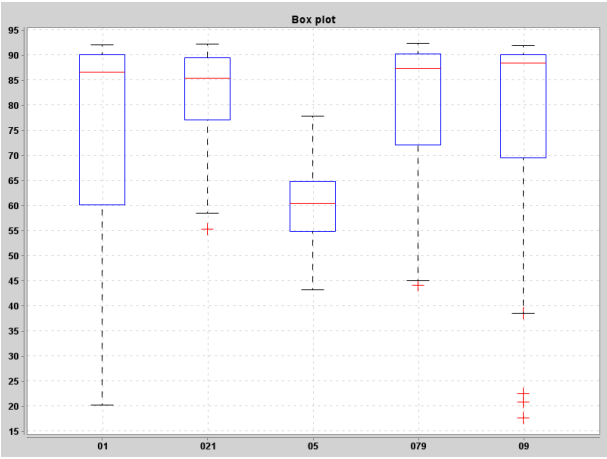
Finally, the "Boolean function bit flip" and "Node Input Swap" in figures (4) and (5) experiments gave the worst results here, because they make really small changes to the BN and with only 30 attempts they usually can't reach good solutions.

4.1.2 Networks with 100 nodes

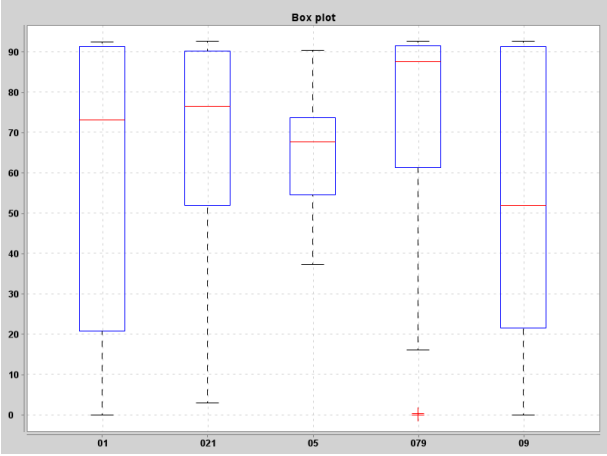
Since the nodes are 4 times as much as before, now the simulations last 14400 seconds instead of 3600, which means that there are 120 editing attempts, 4 times as much.



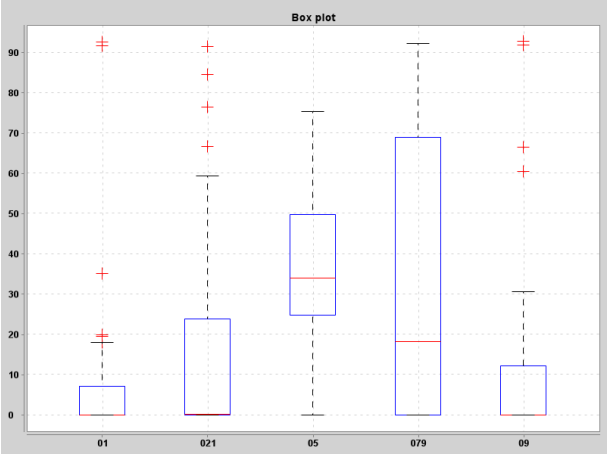
(6) Evaluation box plot for "Input rewire".



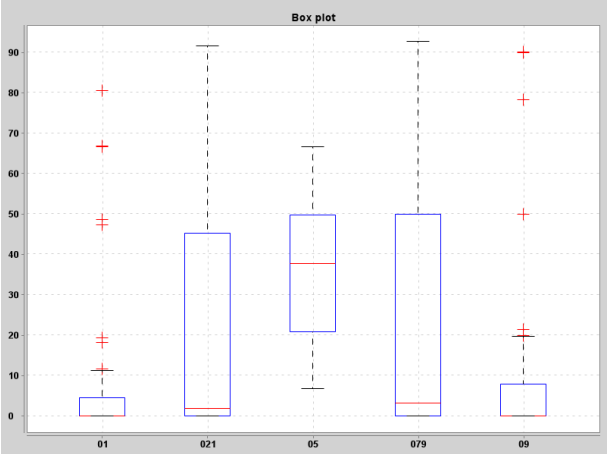
(7) Evaluation box plot for "New network each time".



(8) Evaluation box plot for "Output rewire".



(9) Evaluation box plot for "Boolean function bit flip".



(10) Evaluation box plot for "Node input swap".

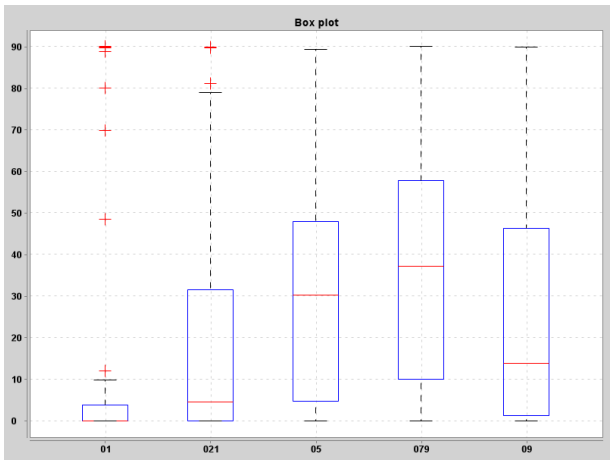
The increase in network size made a visible improvement across the board. The "New network each time" experiment shown in (7) still gives the best results, even better than using 25 nodes, as seen in figure (2); in fact while the search space is now bigger the task is still the same and with 4 times as much time to search, finding at least one good solution becomes more likely. It's also worth noting that in this experiment the chaotic networks are the worst performing ones. The "Output Rewire" and "Input Rewire" experiments visible in figures (8) and (6) also improved and the benefits of critical networks start to show, since they are now the ones performing better. The "Boolean function bit flip" and "Node Input Swap" experiments shown in figures (9) and (10) still give the worst results but they saw an increase in evaluation value reached, too. In these last two experiments the chaotic networks seem to perform better than the others.

4.1.3 Enabling noise for sensors and actuators

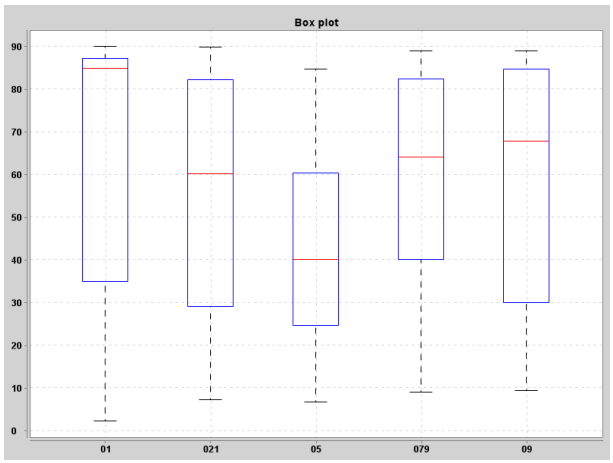
As previously mentioned, without noise the superiority of critical networks is hard to see, so (proximity) sensor and actuator noise is enabled. This makes the task slightly more complex: now if the robot moves along a wall it can actually end up too close to it, so it has to correct its route; it's also possible that a sensor activates for an instant even without an actual obstacle near the robot, so the network has to react correctly even in this case.

Networks with 25 nodes

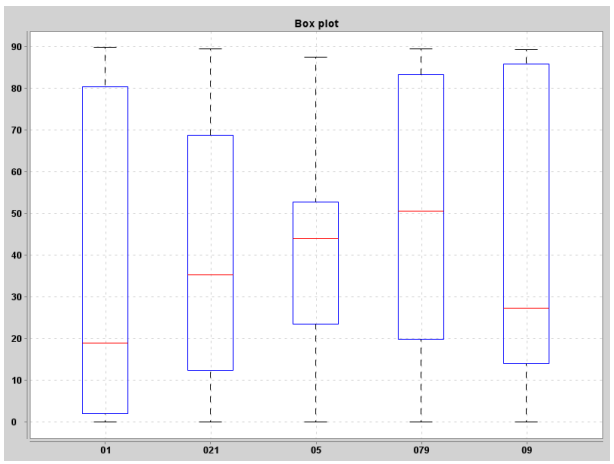
Here 25 nodes were used, so the experiments last 3600 seconds. From now on the experiment's length is shown only when it differs from the usual rule (144 seconds for each BN node).



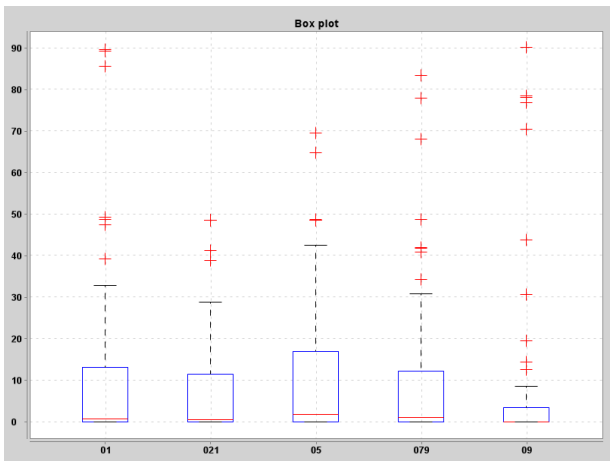
(11) Evaluation box plot for "Input rewire".



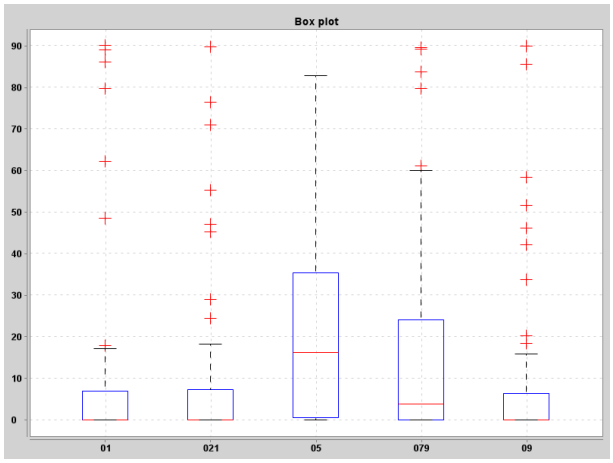
(12) Evaluation box plot for "New network each time".



(13) Evaluation box plot for "Output rewire".



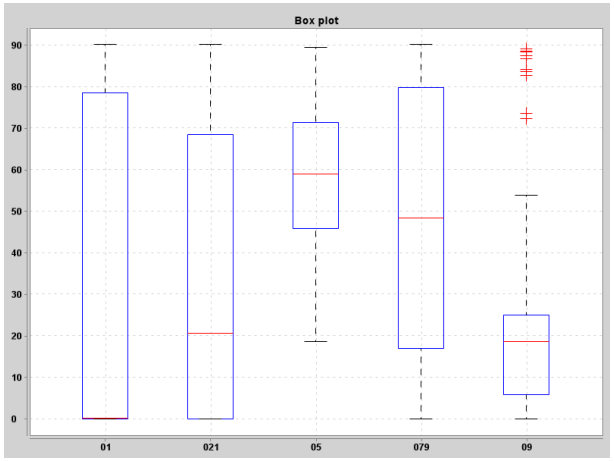
(14) Evaluation box plot for "Boolean function bit flip".



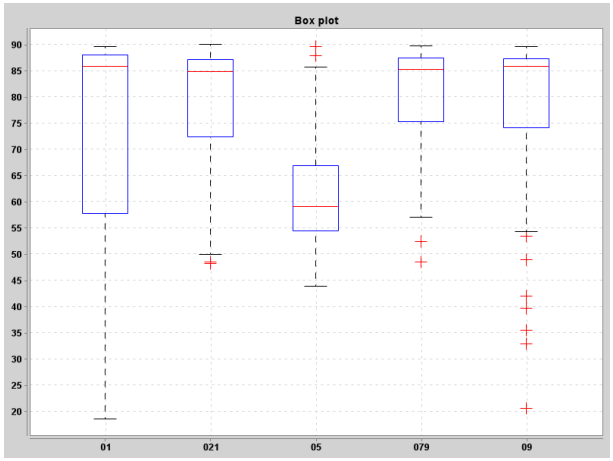
(15) Evaluation box plot for "Node input swap".

The most important change here is that enabling the noise let the critical networks show their capabilities in both the "Output Rewire" and "Input Rewire" experiments, visible in figures (13) and (11), since the bias value of 0.79 now gives the best results for both the experiments. The "New network each time" experiment shown in figure (12) gives even better results than before (subsection 4.1.1, except for bias 0.21 and 0.5), while "Boolean function bit flip" and "Node Input Swap" suffer from the same problem as before, as seen in figures (14) and (15).

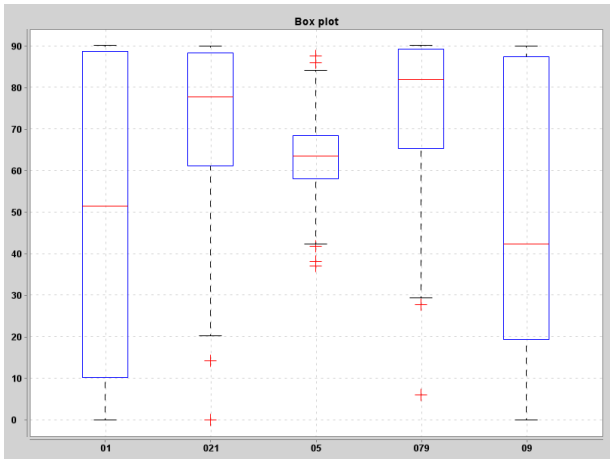
Networks with 100 nodes



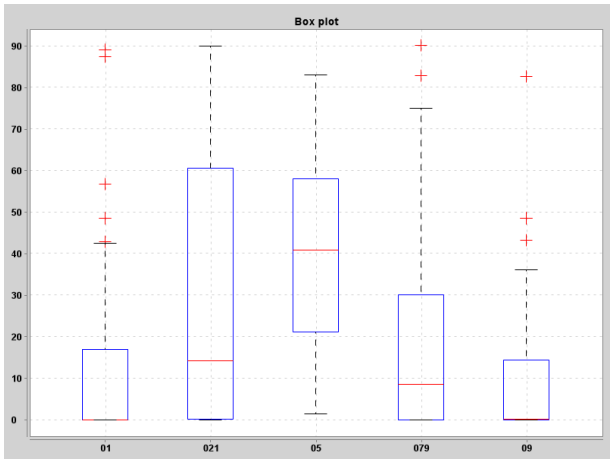
(16) Evaluation box plot for "Input rewire".



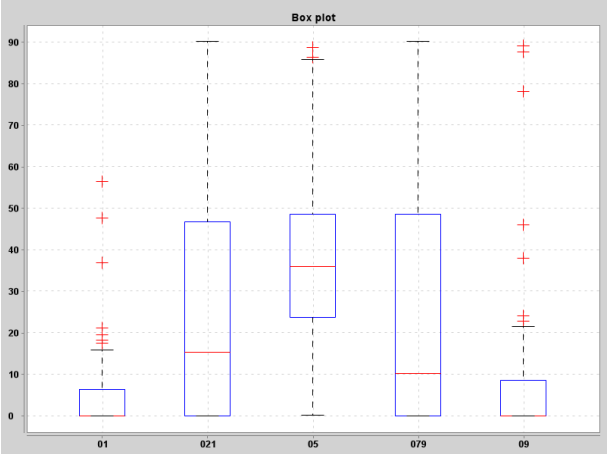
(17) Evaluation box plot for "New network each time".



(18) Evaluation box plot for "Output rewire".



(19) Evaluation box plot for "Boolean function bit flip".



(20) Evaluation box plot for "Node input swap".

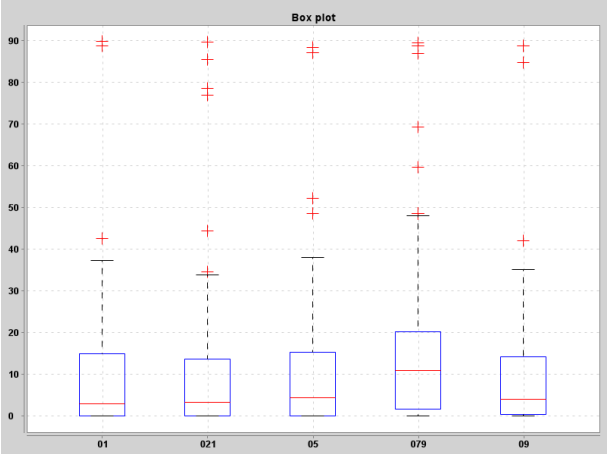
The addition of noise doesn't drastically change the results for BNs with 100 nodes when compared to the results in subsection 4.1.2 (without noise), except that now critical BNs in "Input rewire" seem to perform worse than chaotic ones. The results of these experiments are shown in figures from (16) to (20).

4.1.4 Incremental adaptation scheme

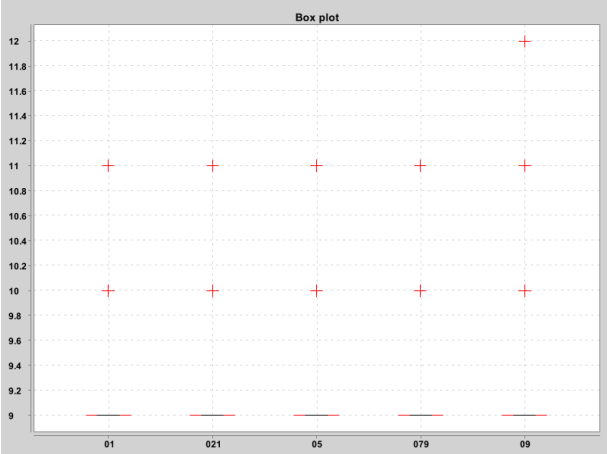
This scheme tries to improve the evaluation value by changing the input wirings, the output wirings or by adding a node to the BN. By plotting an X-Y graph with the network's size on the X axis and the evaluation on the Y axis we can see if there is a correlation between the two. It's also interesting to see whether the added nodes are kept or discarded by the adaptation process.

Networks starting from 9 nodes

Here BNs with 9 nodes are the starting point, with the experiment length set to 2160 seconds, which correspond to 18 editing attempts.



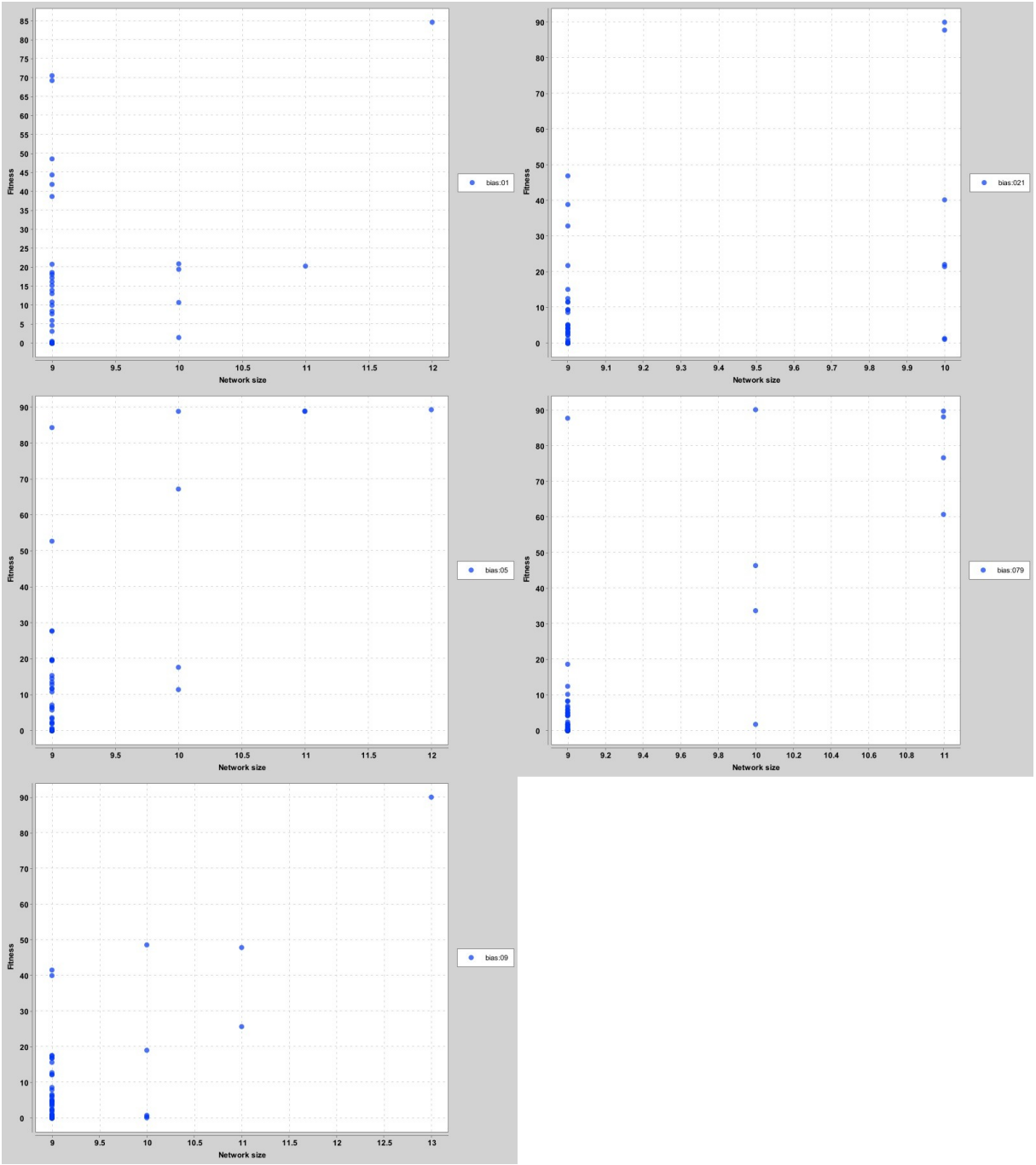
(21) Evaluation box plot for this experiment.



(22) Box plot of the BNs' final size for this experiment .

Critical networks appear here to be slightly better but they all give very poor results, as seen in figure (21). At this point, the cause of this seems to be the very low amount of initial nodes and editing attempts; the following experiment (in 4.1.4) will shed more light into this conjecture.

Given that the average amount of nodes added (if they all get accepted) is $18/3 = 6$, by looking at the BNs' size box plot in figure (22) it's clear that many added nodes were discarded. The hypothesis here is that this is usually not an useful editing attempt. This is even more apparent in the following experiments (in this subsection), by increasing the experiment's length and the starting network's size. This does not mean that there is no correlation between network size and evaluation value, but rather that the addition of a single node to an existing network (without further tuning) is usually detrimental.

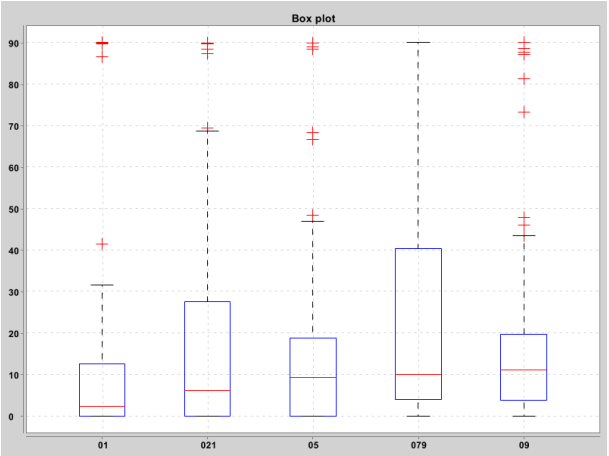


(23) X-Y Graphs for this experiment, showing how the evaluation value changes with network size for each bias value.

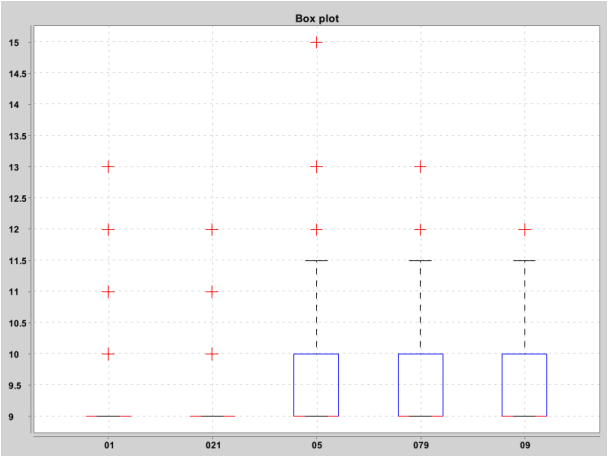
It's hard to get valuable information from the X-Y graph in figure (23) obtained by these results, because the majority of data is in the first value of X, since the networks grew very little. This is why the following experiment is the same as the current one but with three times the length.

Networks starting from 9 nodes with more allotted time

With three times more seconds (6480), the hope in this experiment is to see the networks grow more, to get an insight into how BNs with different sizes behave and how their evaluations compare with each other.

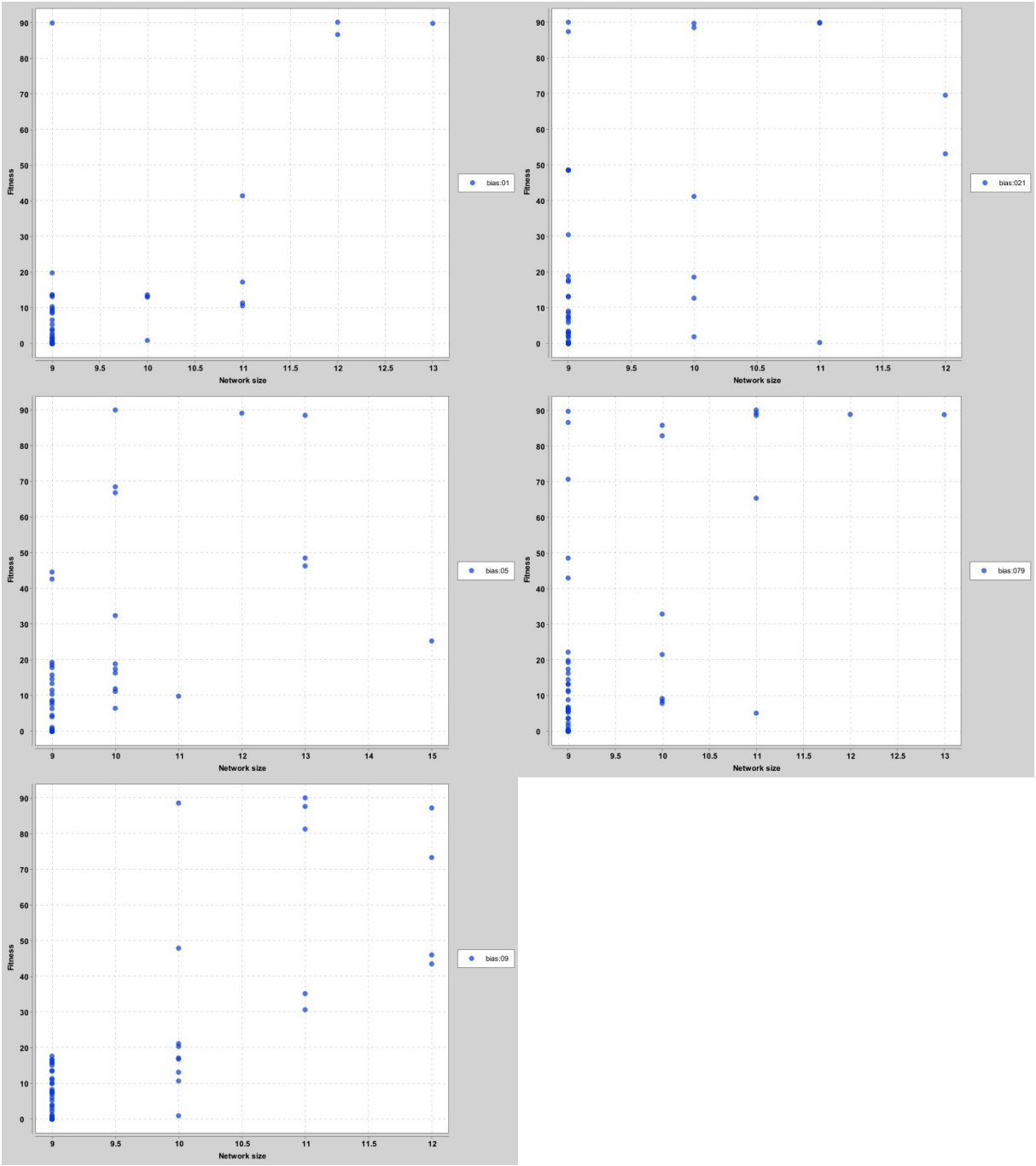


(24) Evaluation box plot for this experiment.



(25) Box plot of the BNs' final size for this experiment .

The evaluations here are still very low, as shown in figure (24), which means that the problem is not the low amount of editing attempts. The evaluation value of critical networks also has a higher variance. Figure (25) shows that the networks still discard many of the added nodes, since these new nodes usually do not increase the evaluation immediately, thus keeping the networks smaller than the ideal size for this task. This explanation seems plausible when considering the results of networks starting with 25 nodes (in the following experiment, same subsection), which obtain higher evaluations than the ones starting from 9.

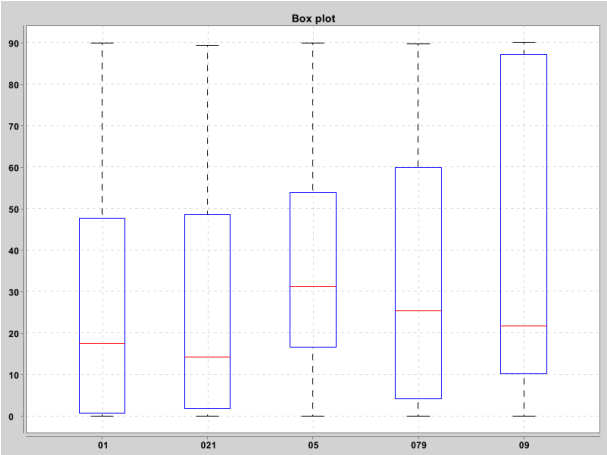


(26) X-Y Graphs for this experiment, showing how the evaluation value changes with network size for each bias value.

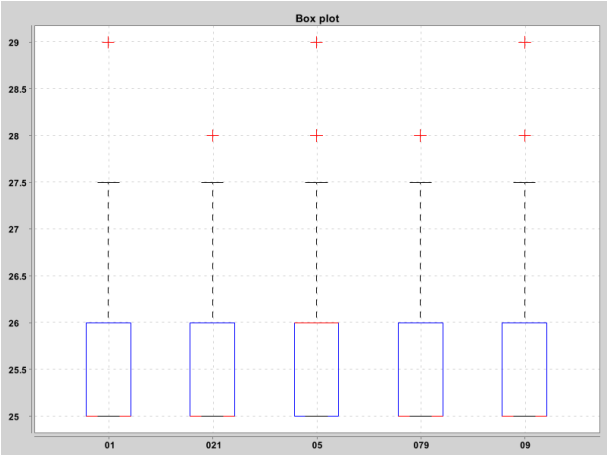
On the X-Y graph in figure (26) we can see that the lower bound of the evaluation value increases with the node count and that the biggest networks also have very high values. Therefore, there seems to be a positive correlation between size and evaluation.

Networks starting from 25 nodes

Now the starting networks are quite bigger than before and should be able to reach higher evaluation values. The experiment's length is 3600 seconds.

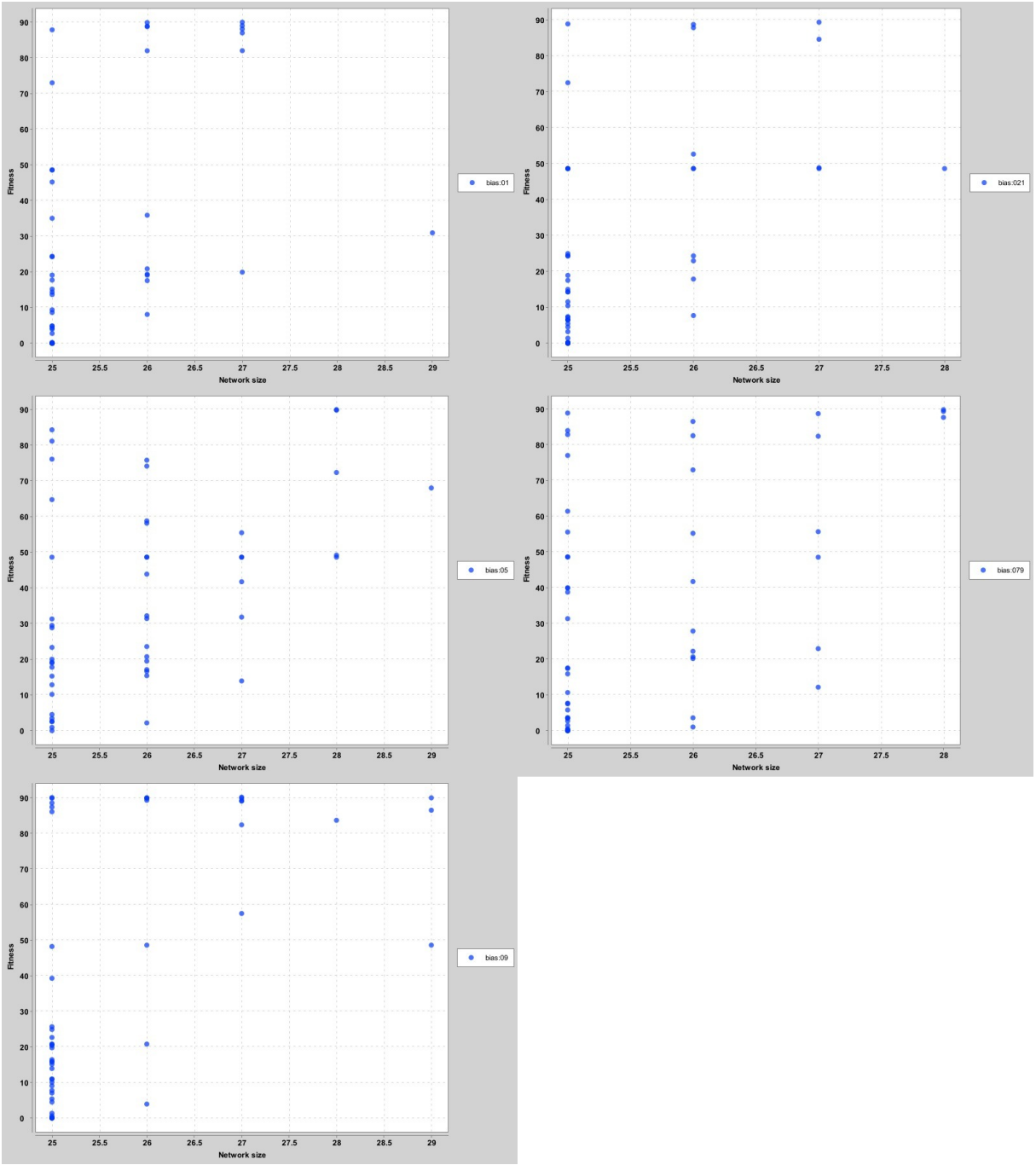


(27) Evaluation box plot for this experiment.



(28) Box plot of the BNs' final size for this experiment .

Figure (27) reveals that the evaluation values are higher than the previous set of experiments shown in figure (21) and (24), as expected, reinforcing the idea that the small initial network's size is a problem, when starting from 9 nodes. The networks also continue to discard most of the new nodes, which causes the growth process to be very slow. The previously described hypothesis for this phenomenon still holds. The BN's growth is visible in figure (28). In this experiment, the X-Y graph in figure (29) still shows a positive correlation between size and evaluation value.



(29) X-Y Graphs for this experiment, showing how the evaluation value changes with network size for each bias. value.

4.2 Generalization check for obstacle avoidance

Some networks with high evaluation values (>= 90) were tested in a different arena, as shown in section 3.3, with multiple small obstacles instead of the big box in the center. Both the BN ensembles with 25 nodes and 100 nodes are able to generalize, but sometimes they get stuck. The robots also show the ability to react to obstacles from only one side and to

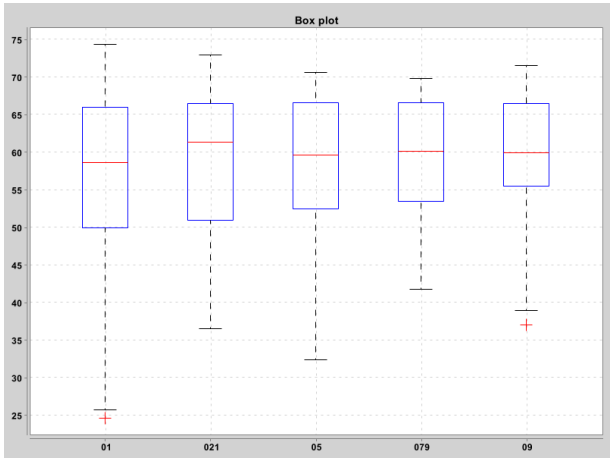
rotate in only one direction: this is enough for the arena where they were trained but this new arena causes the robot to sometimes crash into obstacles. The robot is still usually able to get relatively high evaluation values (>70), when it doesn't get stuck.

4.3 Robot following task

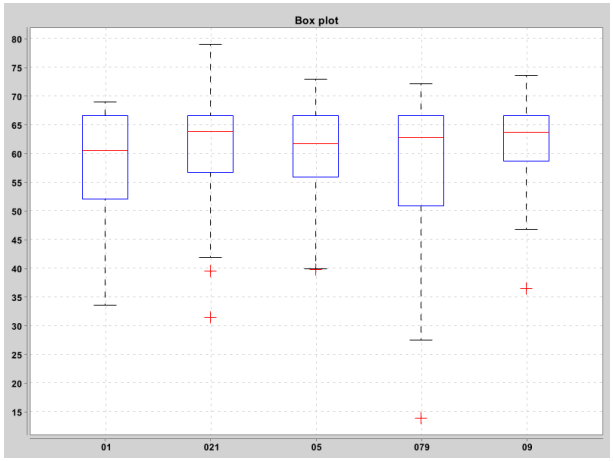
The BNs have 25 nodes (except in section 4.3.8) and the simulations were run for 7200 seconds, which is equal to 60 editing attempts. In the case of incremental learning, the first half of the attempts is done with only the obstacle avoidance part of the evaluation function, adding the second component only in the second half.

4.3.1 1/3 of obstacle avoidance and 2/3 of robot following, only 1 prey

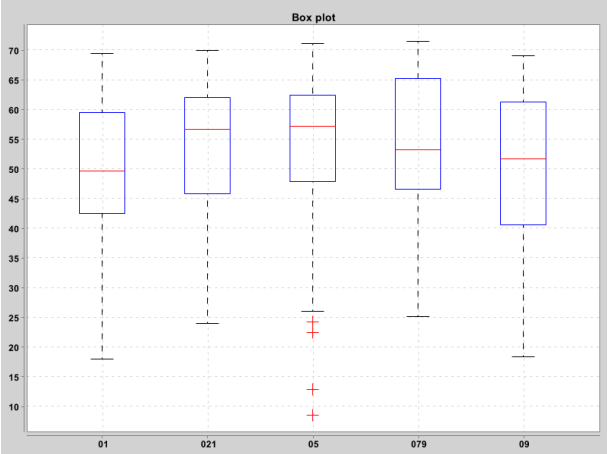
Previous attempts (not shown here) with the weights set to 0.5 and 0.5 showed that the robot could only learn obstacle avoidance. Therefore the weights are set to give more importance to the robot following behaviour.



(30) Evaluation box plot using Input rewire, no Incremental Learning.



(31) Evaluation box plot creating a RBN each time, without Incremental Learning.



(32) Evaluation box plot using Input rewire, with Incremental Learning.

In the results shown in figures from (30) to (32) we can see that very few networks manage to get evaluation values higher than 70. The random editing approach is the only one that achieves values close to 80; the evaluations close to 80 are very rare even in the box plot where they occur and the median values are generally close to 60, except for the experiment with Incremental Learning, which achieves slightly lower values. Overall, "Random Network Each Time" seems to perform best, followed by "Input Rewire" and with Incremental Learning giving the worst results.

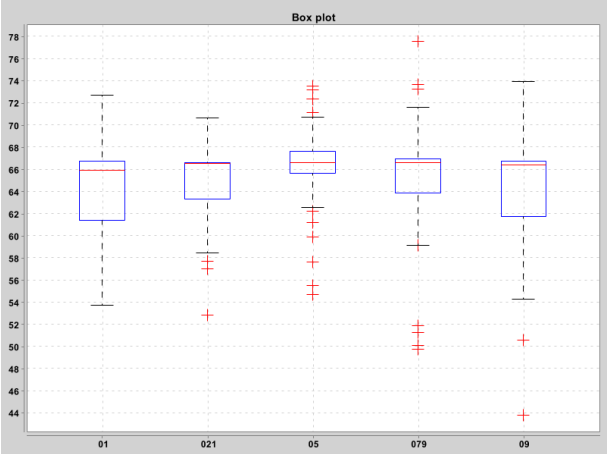
By doing just the robot following task very well it's possible to get a value of 66. Given how many networks get similar values the hypothesis is that the networks simply learned to follow the prey, without obstacle avoidance. By looking at the best networks it's clear that they managed to learn one of two behaviours:

- a "follow and pin to corner" behaviour: this is better than the previous results (using 0.5 as weight for F1 and F2) but the network has not adapted yet to reach both goals, since the robot stays too close to the prey, without doing proper obstacle avoidance;
- a "follow and wait" behaviour: the robot follows the prey and stops when it senses it in its proximity. This is a good result, but there is still room for improvement, in particular regarding obstacle avoidance.

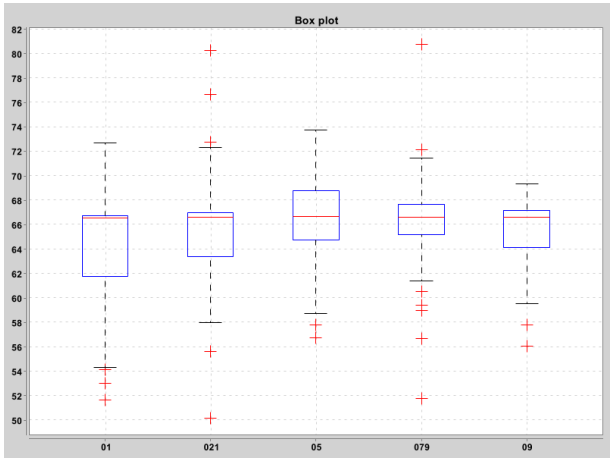
These behaviours are both closer to the expected one than those achieved with both the evaluation function's weights set to 0.5.

4.3.2 Adding one more prey robot

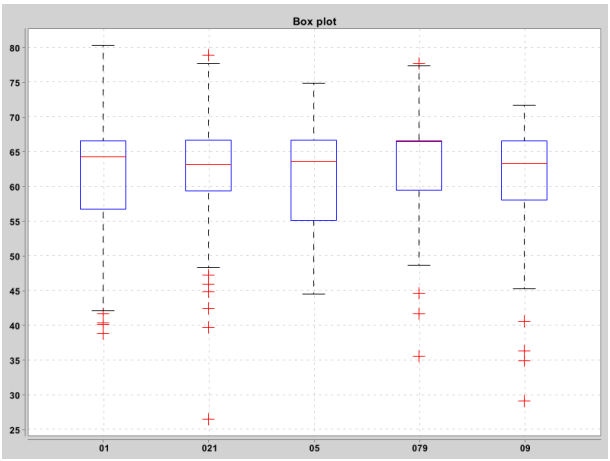
This new set of experiments is done to increase the likelihood of meeting a prey robot, so that there is less time lost searching for it without improving the performance, as explained at the end of 3.4.2. The expectation is to see better behaviours than the ones in the previous experiment (4.3.1).



(33) Evaluation box plot using Input rewire, no Incremental Learning.



(34) Evaluation box plot creating a RBN each time, without Incremental Learning.

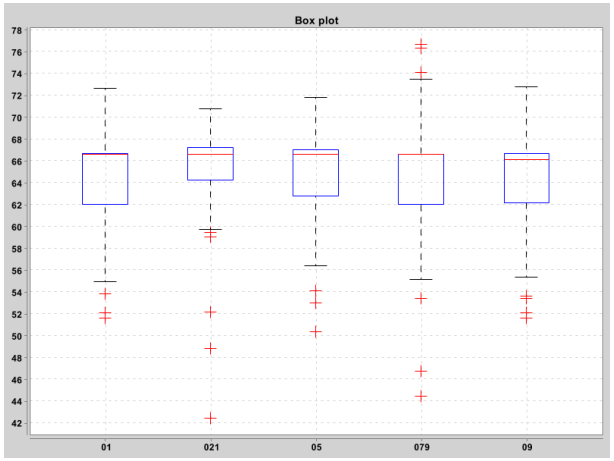


(35) Evaluation box plot using Input rewire, with Incremental Learning.

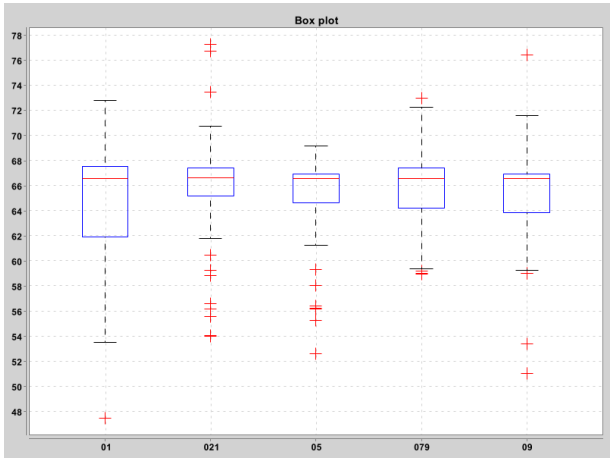
Figures from (33) to (35) show that there is less variance overall. Moreover, the outliers with the best evaluations here developed some lazy behaviours: they simply move in small circles and do some basic obstacle avoidance when needed. This could be because the probability of one of the preys going near the main robot is now too high. Networks that have high evaluations but are not outliers show different behaviours: for example, one of them navigates the arena while doing obstacle avoidance correctly, ignoring the preys almost all the time and rarely following one for a limited time. This is caused by the added robot, since it's now possible to navigate the arena and get high evaluations by simply being close to one prey from time to time. These are unsatisfactory results which seem to be caused by the insertion of a second prey. The next set of experiments (4.3.3) also uses two preys, so more information can be gained from it.

4.3.3 Setting a minimum distance to the prey following component

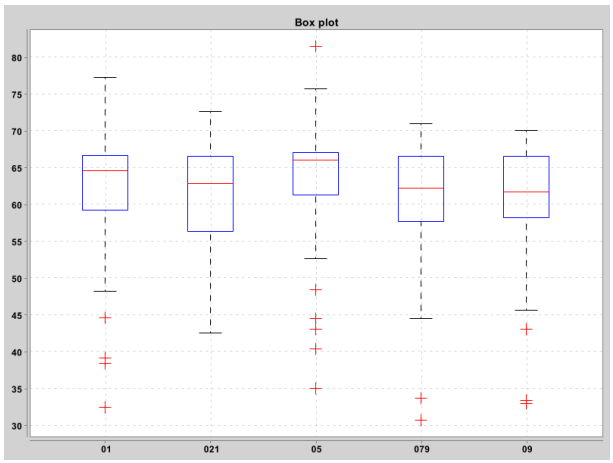
Many of the best results previously achieved (with only one prey) focused on the chasing behaviour while doing small amounts of obstacle avoidance. To solve this, the evaluation function was modified so that the Robot Following component is 0 whenever the robot is too close to the prey. This set of experiments also uses two preys instead of one.



(36) Evaluation box plot using Input rewire, no Incremental Learning.



(37) Evaluation box plot creating a RBN each time, without Incremental Learning.



(38) Evaluation box plot using Input rewire, with Incremental Learning.

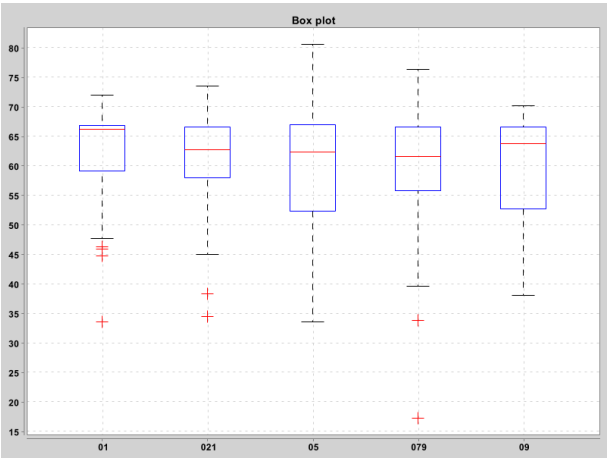
The best performing outliers here show behaviours similar to the previous set of experiments. Therefore, it's possible that the underlying cause of this is still present. Networks with high evaluations which are not outliers showed the first behaviour that resembles the expected one: this robot followed one of the two preys very well, while also doing a certain amount of obstacle avoidance. The only major problem with it is that whenever it loses the prey it simply continues to navigate the arena, since with two preys it's now easy to find

one. This means that the predator doesn't need memory to remember the direction of the recently lost prey, thus making the problem easier and solvable by simpler, reactive robots. This was not intended, since it would be better to have a problem that requires memory.

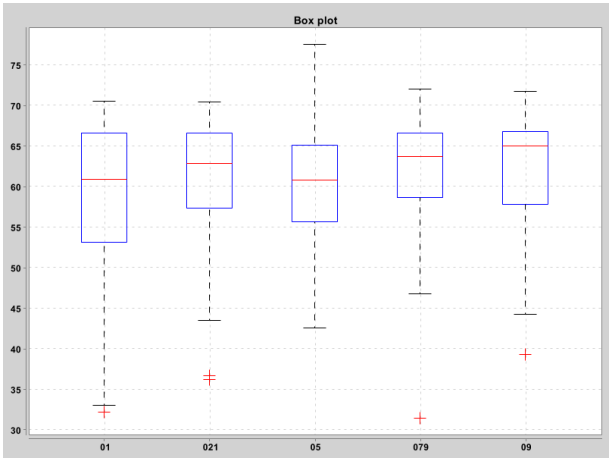
When looking at the results in figures from (36) to (38), it's surprising that using incremental learning produces worse results: this may be because the first task to learn was the simpler one (obstacle avoidance), but more testing would be needed to confirm this theory. Finally, we can see that random search finds solutions that are not as performing as in the previous task (simple obstacle avoidance, in 4.1.3, figure (12), in particular with 0.1 as bias), while also having twice the time to adapt, which means that the current task requires more complexity than a Braitenberg vehicle.

4.3.4 Removal of the added prey

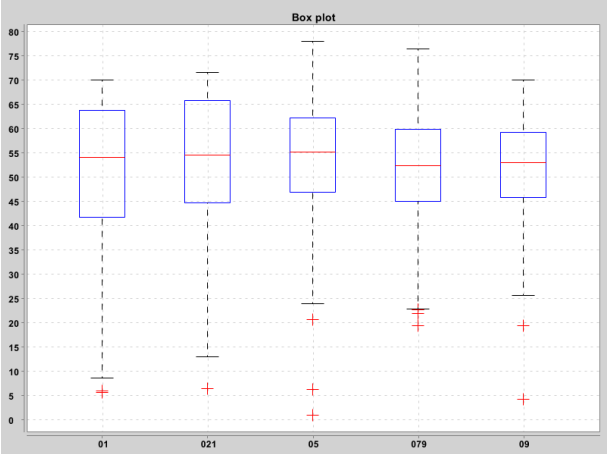
To reduce the amount of lazy behaving solutions, one of the two preys was removed. This group of experiments is otherwise the same as the previous one (4.3.3).



(39) Evaluation box plot using Input rewire, no Incremental Learning.



(40) Evaluation box plot creating a RBN each time, without Incremental Learning.



(41) Evaluation box plot using Input rewire, with Incremental Learning.

We can see in figures from (39) to (41) that the variance is now higher than the previous set of experiments, in general. This is because it's now very important not to lose the one prey available, since the risk is to get only a third of the maximum evaluation possible or less, by doing only obstacle avoidance.

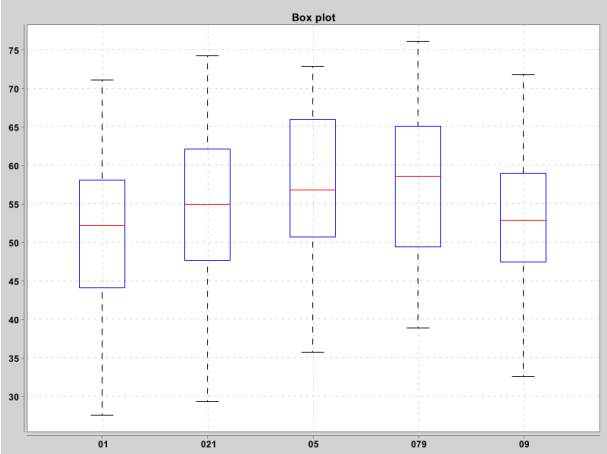
The best networks still show very lazy behaviours, but now they mostly stay in the same place: they do this by making small circles or by moving from side to side of the same corridor. This is probably because the prey can stay in the same place for a long time. Such a behaviour is plausible, since the minimum speed of the prey is set to 0, therefore there are multiple ways for it to stay in the same place for a long time:

- by moving towards a wall that it's already touching;
- by rotating on itself;
- by having both the wheels velocities set to small values.

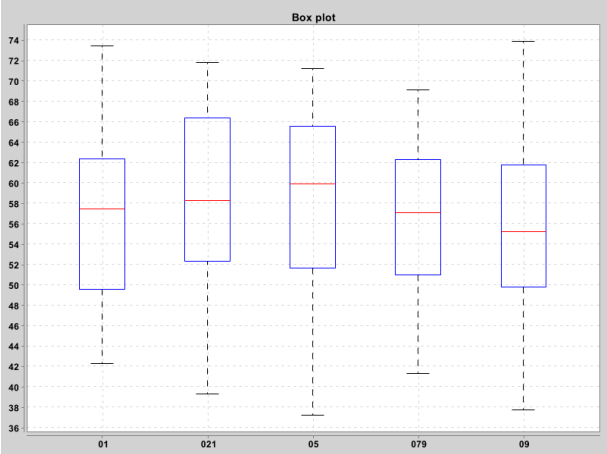
Leaving out the BNs with the highest evaluations, the others with high evaluations still show adequate behaviours, by following the prey.

4.3.5 Making the prey less static

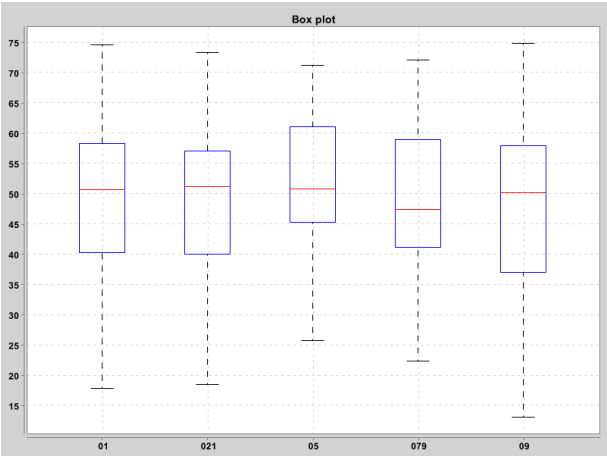
To reduce the amount of lazy and static behaviours in the main robot, the prey should better navigate the arena, without staying for too much time in the same area. This could be achieved by implementing a simple controller that navigates the arena with obstacle avoidance or by using one of the boolean networks previously adapted to achieve this behaviour or, finally, by simply setting the minimum wheels' speed to a value higher than zero. For simplicity, the latter was chosen.



(42) Evaluation box plot using Input rewire, no Incremental Learning.



(43) Evaluation box plot creating a RBN each time, without Incremental Learning.



(44) Evaluation box plot using Input rewire, with Incremental Learning.

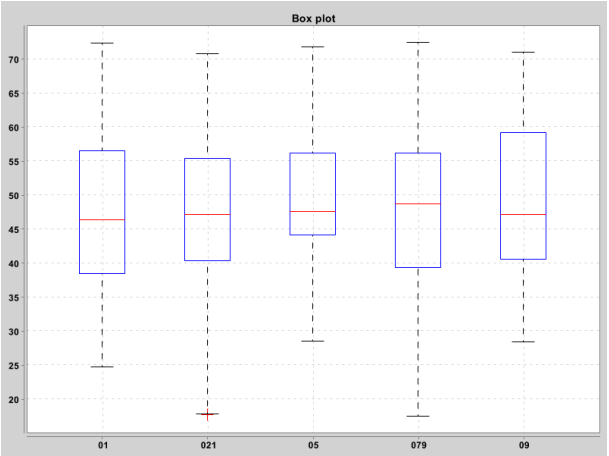
This simple change made some improvements, visible both in the box plots and by watching the robot move. In fact, the three best networks found here all have behaviours that are neither lazy nor static (but some lazy networks can still be found): they follow the prey, with some of them also doing limited amounts of obstacle avoidance. Thus, the learning context now appears to be correct. Furthermore, we can see in figure (42) that when using Input Rewire with no incremental learning the critical networks have a higher evaluation that the others. This was not the case in the previous experiments for this task (section 4.3.4). Since the predator adapts its behaviour to reliably catch the prey, further improvements on the prey's behaviour could guide the adaptation process into more complex solutions, for example by implementing a "flee" behaviour when the prey detects the predator.

Results from the two other sets of experiments can be seen in figures (43) and (44). Networks adapted using Input Rewire and no Incremental Learning seem to achieve better results,

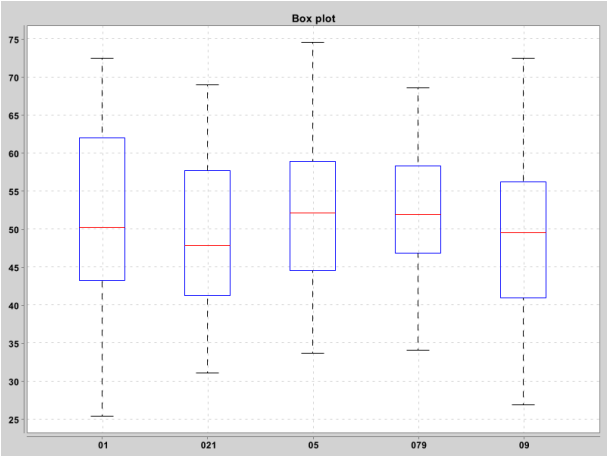
when compared with the randomly created ones, since the best networks using Input Rewire show less lazy behaviours. The reason for this could be that by creating BNs randomly the results are "overdesigned" to the evaluation function; the fact that this happens less when using Input Rewire would then be explained by the reduced amount of change that this adaptation process applies at each editing attempt. Overdesigned networks might show a wider reality-gap; this could be tested by using the adapted networks on a real robot.

4.3.6 Reducing the maximum RAB distance

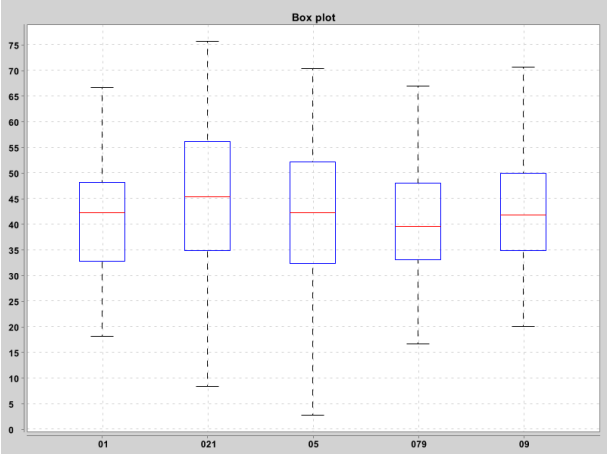
Since some adapted BNs still showed lazy and static behaviours (in particular when creating new RBNs at each step), it could be beneficial to change the maximum distance of the Range-and-bearing system from 1m to 80cm, to guide the adaptation towards more dynamic solutions. In fact, reducing this distance means that the robot has to closely follow the prey, without "resting" in the same position for too much time.



(45) Evaluation box plot using Input rewire, no Incremental Learning.



(46) Evaluation box plot creating a RBN each time, without Incremental Learning.

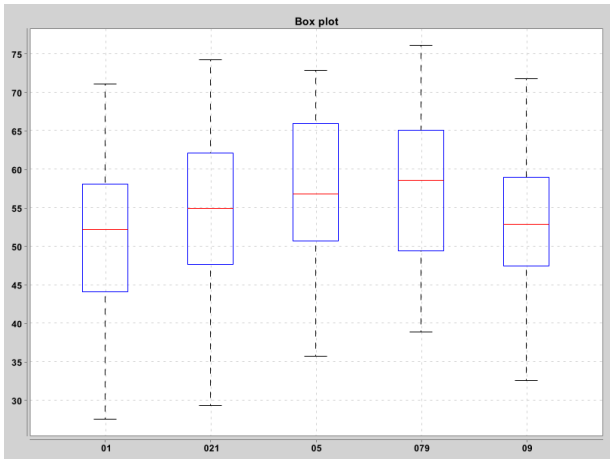


(47) Evaluation box plot using Input rewire, with Incremental Learning.

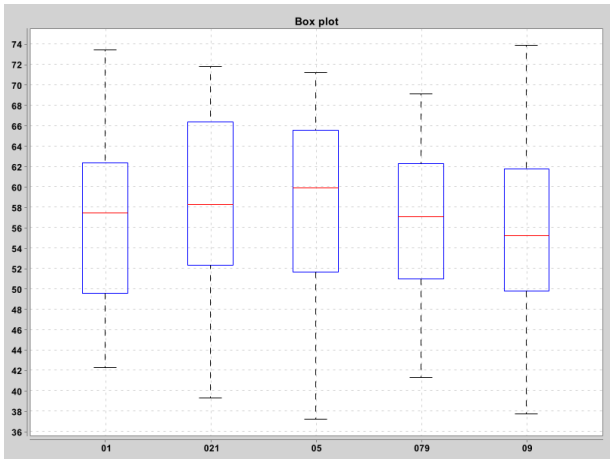
The results of the RAB distance reduction are visible in figures from (45) to (47). The robot behaviours did not show meaningful clues that could point to an improvement on the previous results (in 4.3.5). In fact, when focusing on the five best performing configurations we see that one of them is not able to follow the prey and some of the others show difficulties when trying to find it or to avoid losing it.

4.3.7 Review of the adaptation processes

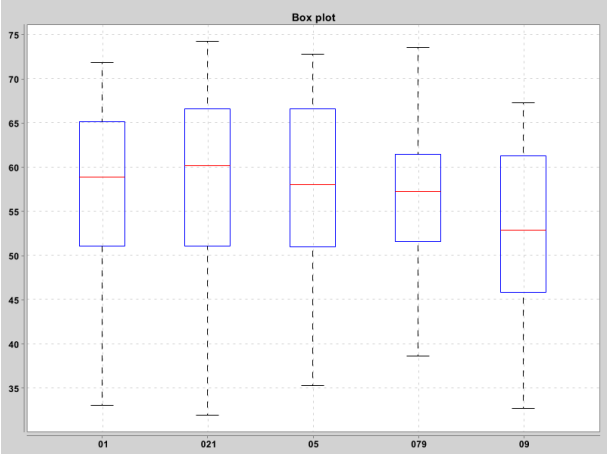
Since the results from the experiments in the previous section (4.3.6) did not achieve clearly better results, the RAB range is set again to 1m. Experiments with this configuration are already present in section 4.3.5, but only a subset of the adaptation processes used on this work are applied there; hence, the results of all the processes described in this work are shown here (even the ones already shown, in figures (48) and (49)), to give a complete overview of these processes on the robot chasing task.



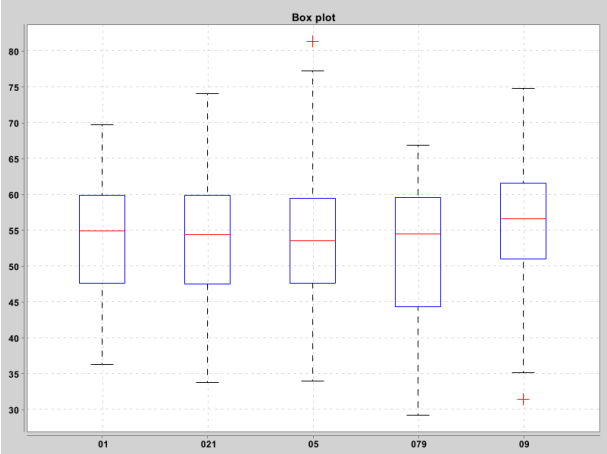
(48) Evaluation box plot for "Input rewire".



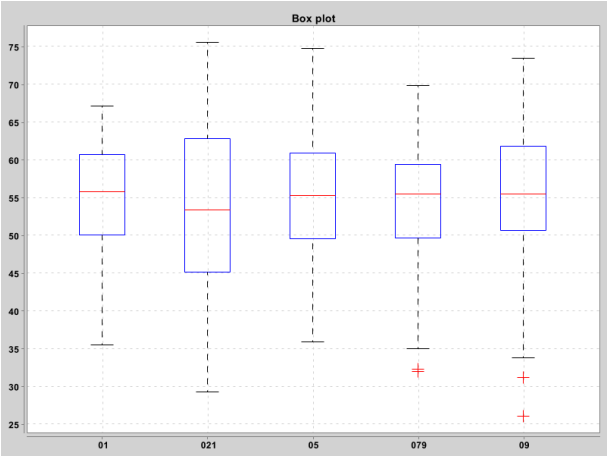
(49) Evaluation box plot for "New network each time".



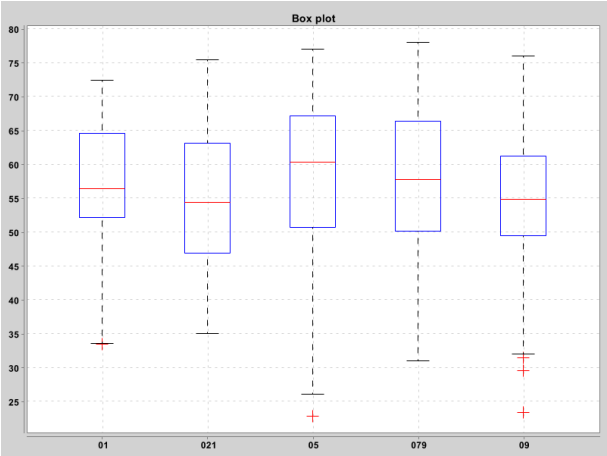
(50) Evaluation box plot for "Output rewire".



(51) Evaluation box plot for "Boolean function bit flip".



(52) Evaluation box plot for "Node input swap".

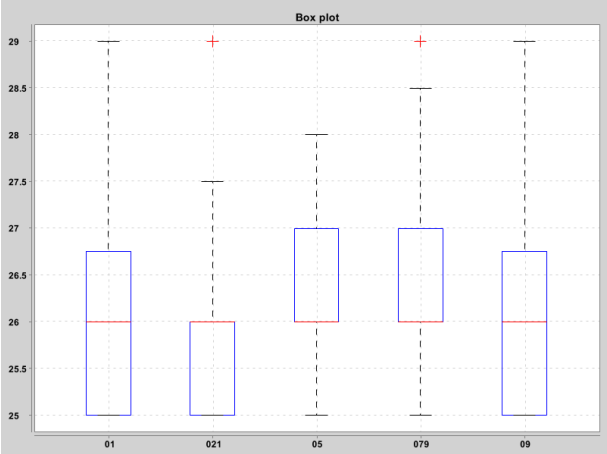


(53) Evaluation box plot for "Incremental adaptation".

In the figures from (48) to (53) we see that all the median evaluations are in the [50, 60] range, with the worst performers being the Bit Flip and Node Input Swap, since they make very small changes at each editing attempt. The critical BNs are clearly better than the others only in the Input Rewire experiment, with Output Rewire showing an almost mirrored distribution, when compared with Input Rewire; these results should be studied further.

BN growth with the Incremental adaptation process

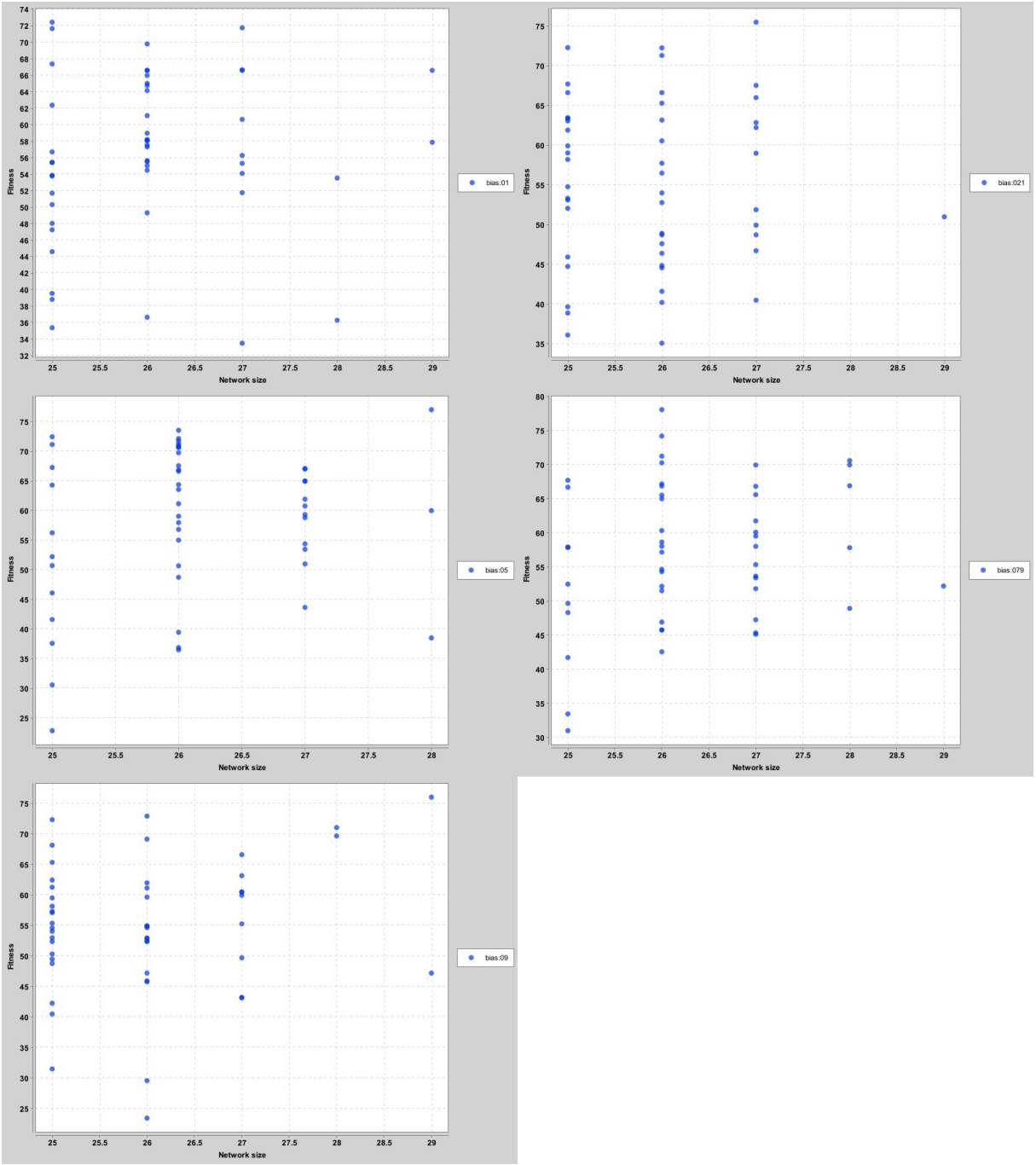
The results of this adaptation process are already shown in this subsection, in figure (53); here, though, the focus is on how the BNs grow using this process.



(54) Box plot for this experiment of the BNs' final size .

In figure (55) the positive correlation between network size and evaluation is visible once again. The networks now discard a few less new nodes and therefore grow slightly more than the previous task (in 4.1.4), as seen in figure (54): the reason for this could be the increased difficulty of this task, compared to obstacle avoidance, but the growth is still far from abundant.

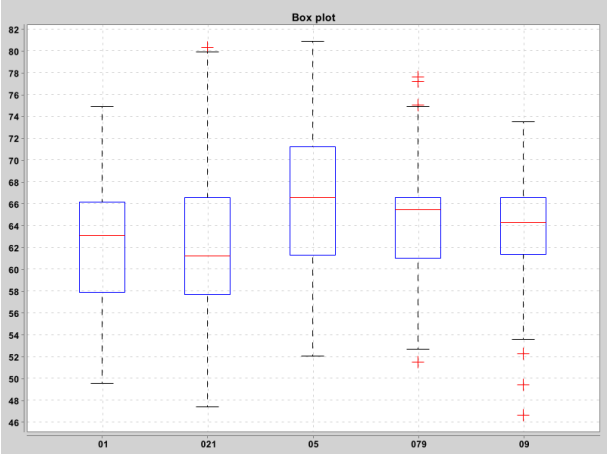
In future research, experiments can be done to test whether it can be helpful to increase the amount of steps used to test a single configuration, to evaluate with more precision the chasing capabilities of the robot.



(55) X-Y Graphs of the BN growth using the "Incremental Adaptation" process.

4.3.8 Networks with 100 nodes

Big BNs are also tested with the robot following task, with the simulation length set to 28800 seconds (4 times as much as the BNs with 25 nodes). For simplicity, only the "Input Rewire" experiment was done.



(56) Evaluation box plot for big BNs and Input rewire, no Incremental Learning.

Here the median evaluation increase is visible, as seen in (56). Also, the critical BNs now seem to struggle against the chaotic ones.

4.4 Generalization check for robot following

The same generalization check in 4.2 is also done for the robot following task. Excluding the lazy solutions and the rare problematic executions where the prey or the main robot get stuck, the configurations with high evaluations (>= 70) showed that they usually can generalize, since they can follow the prey even in this unknown environment while also doing some obstacle avoidance, achieving evaluations above 60.

4.5 Final remarks

4.5.1 Obstacle avoidance

Obstacle avoidance without any noise and without complex obstacles is a task simple enough that satisfactory solutions can be found by simple random search. Increasing the BN's size to 100 results in visible improvements in this task. Enabling noise to sensors and actuators adds complexity to the task and allows the critical networks to show their advantages. The "Node Input Swap" and "Boolean Function Bit Flip" adaptation processes make very small changes, so they usually can't find good solutions in low amounts of time. When adding a single node to the network (without further tweaking) as part of the adaptation process, many added nodes get discarded, even if there is a positive correlation between network size and performance. In fact, this correlation seems to exist. High performance configurations can generalize, since they can achieve the goal even in an arena filled with different obstacles, which were never seen before by the robot.

4.5.2 Robot following

Robot chasing seems to be harder than obstacle avoidance. In fact, if equal weights for the two factors are used in the evaluation function then the robot only adapts to do obstacle avoidance; thus, the weights must favor robot chasing. While adapting, the robot can find a lazy behaviour that does not perform the robot following task as intended; to avoid this, it's important to structure correctly the evaluation function, the arena and all the robot's parameters so that these behaviours are rare. Adding one more prey robot does not help in this case and neither does setting the maximum RAB distance below 1m, but using a minimum distance in the prey chasing component of the evaluation function is at least not detrimental and seems to be helpful. The combined task of obstacle avoidance and prey following requires more complexity than a Braitenberg vehicle. Behaviours that are both static and lazy can be discouraged by making the prey's movement more dynamic. It's possible that by randomly creating BNs more overdesigned networks are formed. BNs with 100 nodes perform better than smaller ones even in this task. Also, critical networks do not consistently show their superiority when doing prey chasing; this could be the subject of further research. Configurations with high performance on this task can also generalize.

4.5.3 Random adaptation process

Multiple remarks can be made on the "New network each time" process:

- it generally achieves adequate results, which means that the RBN's dynamics are rich enough to obtain an effective form of control;
- chaotic networks seem less flexible: this indirectly confirms results in Cell Biology [25];
- the reasons for the high evaluations are probably the moderate BN size and the fact that the goal tasks used do not change over time, therefore there is no need for gradual changes. This last reason is very important when designing robots, since a constantly changing goal task may require an adaptation process that is not completely random;
- in this work, the cost of the editing attempt is not dependent on the amount of edits. The lack of this dependency is also a likely reason for the high evaluations. A similar relationship may exist in both natural and artificial systems;
- finally, it could be useful to proceed with an in-depth study on the properties of critical RBN ensembles.

The next chapter contains a short description about what future works and research can do to better understand and tackle the problems encountered in this work.

Chapter 5

Future Works

To increase the knowledge and improve the current state of the art on this topic, future work should concentrate on multiple aspects:

- improve the current models that mimic the behaviour of biological Gene Regulatory Networks;
- concentrate on online adaptation techniques by trying different ones, conceiving new ones or improving the ones already known;
- design methodologies to simplify the creation of robot controllers that successfully adapt online.

The current work could be a starting point for further research on the topic. The following are some ideas for this purpose:

- a standardized suit of tasks with different features and levels of complexity could be created and used as a "benchmark suit" for online adaptation techniques. The tasks and their evaluation functions should be designed in a way that punishes lazy solutions. This could be useful to quickly compare different controller configurations and adaptation techniques with a high level of confidence in the results;
- the experiments could be re-run with larger networks and with more executions: the former could be useful to check that the findings are still true on big networks, while the latter achieves higher statistical significance;
- as noted in 4.3.5 and 4.3.7, using a flee behaviour in the prey when it's near the predator could be helpful, as well as increasing the steps used to test a single configuration, so experiments to test these hypotheses can be done. Also, any further experiments should initially set all the BN's nodes to 0, to avoid variance due to the BN's initialization [8];
- using the arena with multiple small obstacles during training could drive the adaptation towards more advanced behaviours, since the environment would be harder to navigate and the prey could use the obstacles to escape;
- the critical BNs seem to struggle against the other ones when doing robot following, as seen in 4.3.7, so this phenomenon should be studied in more detail;

- instead of adding a new node and keeping it if it improves the performance, an adaptation process could try to remove a node and accept the change if the evaluation is at least very close to the previous one. Thus, this process would remove the less useful nodes, applying a sort of online "lossy compression" to the network. Using this while also enabling the insertion of new nodes could give insights about the required network size for each task, thus also enabling a better understanding of their difficulty level;
- the simple algorithm used to adapt the current configuration could be swapped with a different algorithm, such as the (1+1)-Restart-Online [36] (explained in 2.1), by editing it so that it uses one of the adaptation processes explained here, instead of the gaussian mutation operator;
- saving the history of the BN's states at the start of the adaptation process and at the end and then comparing the initial results with the final ones could be useful to learn more about how the BN-controlled robot adapts and behaves. Similarly, the initial and final BNs can be saved and compared. This comparisons can be done by using measures from *Information Theory* such as entropy, mutual information, LMC complexity, etc;
- ideally, the online adapting robot should be able to find only the solutions that satisfy a set of guarantees and constraints about its behaviour, as specified by the developer, so that the robot can be used in mission-critical situations. The challenge here is to find a technique for this that is general enough and that has adequate performance with limited hardware;
- a downside of using a BN-controlled robot is that the conversion to and from booleans for the values of sensors and actuators has to be defined explicitly; this is currently done in a fixed and task-dependent way. Finding a generalized and adaptable solution that becomes part of the BN model could enrich it and prove to be useful to reduce the amount of steps necessary to use such networks.

Conclusion

Boolean Networks are a model for Gene Regulatory Networks; GRNs can produce complex behaviours even with a compact description, so they can be effectively used as robot programs [50]. After explaining what GRNs and BNs are, this work shows that the simulated robot employs a BN as the core of its controller, with a simple online and automatic design methodology that uses one of many possible adaptation processes. The vision is to have a micro robot trying to accomplish a mission in environments where human exploration is not possible [49], therefore overly complex techniques cannot be used. Six simple adaptation processes are analyzed, which can edit the network itself or the coupling between the BN and the robot. The experimental setting and the goal tasks of Obstacle Avoidance and Robot Following are also described, along with their evaluation function, so that it's possible to reproduce the experiments using the ARGoS simulator.

The results from the experiments yielded multiple insights and are summarized in 4.5; the most important ones are the following:

- the online adaptation methodology used on the BN-robot can achieve satisfactory behaviours that can also generalize in both tasks;
- enabling the noise on the simulated sensors and actuators helps to show the superiority of critical BNs;
- the adaptation process that randomly creates BNs generally achieves adequate results, but this could simply be caused by the limited size of the networks used and by the fact that the chosen goal tasks do not change over time;
- results do not always show the critical networks as the superior BN ensemble;
- the adaptation processes that edit the coupling between the BN and the robot's input or outputs seem to be the most effective for the selected tasks;
- there appears to be a positive correlation between the BN's size and the robot's minimum performance.

More work and research is needed to further the knowledge about these subjects, since the results shown here are preliminary. In fact, further experiments should be performed to better understand the reasons of the observed phenomena.

Appendix A

Additional Material

- videos of some of the more interesting robot executions in simulation can be found at this [link].

Bibliography

[1] A. Alaghi and J. P. Hayes. Survey of stochastic computing. *ACM Transactions on Embedded Computing Systems*, 12(2s):1–19, May 2013.

[2] M. Aldana. Boolean dynamics of networks with scale-free topology. *Physica D: Nonlinear Phenomena*, 185(1):45–66, Oct. 2003.

[3] M. Aldana, E. Balleza, S. Kauffman, and O. Resendiz. Robustness and evolvability in genetic regulatory networks. *Journal of Theoretical Biology*, 245(3):433–448, Apr. 2007.

[4] C. Blum and A. Roli. Metaheuristics in combinatorial optimization. *ACM Computing Surveys*, 35(3):268–308, Sept. 2003.

[5] M. Braccini. Applications of biological cell models in robotics. *CoRR*, abs/1712.02303, 2017.

[6] M. Braccini. Towards a boolean network-based computational model for cell differentiation and its applications to robotics, 2020.

[7] M. Braccini, S. Montagna, and A. Roli. Self-loops favour diversification and asymmetric transitions between attractors in boolean network models. In *Artificial Life and Evolutionary Computation - 13th Italian Workshop, WIVACE 2018, Parma, Italy*, volume 900 of *Communications in Computer and Information Science*, pages 30–41. Springer, 2018.

[8] M. Braccini, A. Roli, and S. A. Kauffman. Online adaptation in robots as biological development provides phenotypic plasticity, 2020. arXiv: `2006.02367 [cs.RO]`.

[9] M. Braccini, A. Roli, M. Villani, and R. Serra. A comparison between threshold ergodic sets and stochastic simulation of boolean networks for modelling cell differentiation. In *Communications in Computer and Information Science*, pages 116–128. Springer International Publishing, 2018.

[10] M. Braccini, A. Roli, M. Villani, and R. Serra. Automatic design of boolean networks for cell differentiation. In *Advances in Artificial Life, Evolutionary Computation, and Systems Chemistry*, pages 91–102. Springer International Publishing, 2017.

[11] V. Braitenberg. *Vehicles: Experiments in Synthetic Psychology*. MIT Press, 1989.

[12] S. Dealy, S. Kauffman, and J. Socolar. Modeling pathways of differentiation in genetic regulatory networks with boolean networks. *Complexity*, 11(1):52–60, 2005.

[13] B Derrida and Y Pomeau. Random networks of automata: a simple annealed approximation. *Europhysics Letters (EPL)*, 1(2):45–49, 1986.

[14] E. A. Di Paolo. Rhythmic and non-rhythmic attractors in asynchronous random boolean networks. *Biosystems*, 59(3):185–195, Mar. 2001.

[15] G. M. Edelman and J. A. Gally. Degeneracy and complexity in biological systems. *Proceedings of the National Academy of Sciences*, 98(24):13763–13768, Nov. 2001.

[16] C. FH. On protein synthesis. *Symposia of the Society for Experimental Biology*, 12:138–163, 1958.

[17] L. Garattoni and M. Birattari. *Swarm robotics*. In *Wiley Encyclopedia of Electrical and Electronics Engineering*. 2016, pages 1–19.

[18] L. Garattoni, A. Roli, M. Amaducci, C. Pinciroli, and M. Birattari. Boolean network robotics as an intermediate step in the synthesis of finite state machines for robot control. In *Advances in Artificial Life, ECAL 2013*. MIT Press, Sept. 2013.

[19] C. Gershenson. Classification of random boolean networks. *CoRR*, cs.CC/ 0208001, 2002.

[20] C. Gershenson. Guiding the self-organization of random boolean networks. *Theory in Biosciences*, 131(3):181–191, Nov. 2011.

[21] F. Gomez and R. Miikkulainen. Incremental evolution of complex general behavior. *Adaptive Behavior*, 5(3-4):317–342, Jan. 1997.

[22] S. Gupta, S. S. Bisht, R. Kukreti, S. Jain, and S. K. Brahmachari. Boolean network analysis of a neurotransmitter signaling pathway. *Journal of Theoretical Biology*, 244(3):463–469, Feb. 2007.

[23] I. Harvey and T. Bossomaier. Time out of joint: attractors in asynchronous random boolean networks. In *Proceedings of the Fourth European Conference on Artificial Life (ECAL97*, pages 67–75. MIT Press, 1997.

[24] K. Inoue. Logic programming for boolean networks. In *IJCAI*, 2011.

[25] S. Kauffman. The origins of order: self-organization and selection in evolution. 1993. Oxford University Press.

[26] S. Kauffman. A proposal for using the ensemble approach to understand genetic regulatory networks. *Journal of Theoretical Biology*, 230(4):581–590, Oct. 2004.

[27] S. Kauffman, C. Peterson, B. Samuelsson, and C. Troein. Genetic networks with canalyzing boolean rules are always stable. *Proceedings of the National Academy of Sciences*, 101(49):17102–17107, 2004.

[28] D. E. Knuth. *The Art of Computer Programming, 2 (3rd ed.)* Addison Wesley, 1997.

[29] H. Lähdesmäki, S. Hautaniemi, I. Shmulevich, and O. Yli-Harja. Relationships between probabilistic boolean networks and dynamic bayesian networks as models of gene regulatory networks. *Signal Processing*, 86(4):814–834, Apr. 2006.

[30] J. Liang and J. Han. Stochastic boolean networks: an efficient approach to modeling gene regulatory networks. *BMC Systems Biology*, 6(113), 2012.

[31] J. T. Lizier, S. Pritam, and M. Prokopenko. Information dynamics in small-world boolean networks. *Artificial Life*, 17(4):293–314, Oct. 2011.

[32] Lua 5.3 reference manual. `https://www.lua.org/manual/5.3/manual.html#3.4.7`. Accessed: 2020-06-27.

[33] Lua performance - spring. `https://springrts.com/wiki/Lua_Performance`. Accessed: 2020-06-27.

[34] G. Miller and D. Cliff. Co-evolution of pursuit and evasion i: biological and gametheoretic foundations. 1994. Technical Report CSRP 311, School of Cognitive and Computing Sciences, University of Sussex, Brighton, UK.

[35] S. Montagna, M. Braccini, and A. Roli. The impact of self-loops in random boolean network dynamics: A simulation analysis. In *Artificial Life and Evolutionary Computation - 12th Italian Workshop, WIVACE 2017, Venice, Italy*, volume 830 of *Communications in Computer and Information Science*, pages 104–115. Springer, 2017.

[36] J.-M. Montanier and N. Bredeche. Embedded evolutionary robotics: the (1+1)-restart-online adaptation algorithm. In S. Doncieux, N. Bredèche, and J.-B. Mouret, editors, *New Horizons in Evolutionary Robotics*, pages 155–169, Berlin, Heidelberg. Springer Berlin Heidelberg, 2011.

[37] A. Munteanu and R. V. Solé. Neutrality and robustness in evo-devo: emergence of lateral inhibition. *PLoS Computational Biology*, 4(11):e1000226, Nov. 2008. E. J. Crampin, editor.

[38] S. Nolfi and D. Floreano. Evolutionary robotics. The MIT Press, Cambridge, MA, 2000.

[39] M. Nykter, N. D. Price, M. Aldana, S. A. Ramsey, S. A. Kauffman, L. E. Hood, O. Yli-Harja, and I. Shmulevich. Gene expression dynamics in the macrophage exhibit criticality. 2008. Proceedings of the National Academy of Sciences, USA vol. 105, pp. 1897–1900.

[40] C. Oosawa and M. A. Savageau. Effects of alternative connectivity on behavior of randomly constructed boolean networks. *Physica D: Nonlinear Phenomena*, 170(2):143–161, Sept. 2002.

[41] R. Pfeifer and C. Scheier. *Understanding Intelligence*. MIT Press, Cambridge, MA, USA, 1999.

[42] C. Pinciroli, V. Trianni, R. O'Grady, G. Pini, A. Brutschy, M. Brambilla, N. Mathews, E. Ferrante, G. Di Caro, F. Ducatelle, M. Birattari, L. M. Gambardella, and M. Dorigo. ARGoS: a modular, parallel, multi-engine simulator for multi-robot systems. *Swarm Intelligence*, 6(4):271–295, 2012.

[43] Plow 2015 challenge. `http://argos-sim.info/plow2015/`. Accessed: 2020-05-30.

[44] Poisson distribution. `https://en.wikipedia.org/wiki/Poisson_distribution#Generating_Poisson-distributed_random_variables`. Accessed: 2020-07-05.

[45] Programming in lua: 14. `https://www.lua.org/pil/14.html`. Accessed: 2020-06-27.

[46] K. Raza and M. Alam. Recurrent neural network based hybrid model for reconstructing gene regulatory network. *Computational Biology and Chemistry*, 64:322–334, Oct. 2016.

[47] Regulation of gene expression. `https://en.wikipedia.org/wiki/Regulation_of_gene_expression`. Accessed: 2020-06-16.

[48] A. S. Ribeiro, S. A. Kauffman, J. Lloyd-Price, B. Samuelsson, and J. E. S. Socolar. Mutual information in random boolean models of regulatory networks. *Physical Review E*, 77(1), Jan. 2008.

[49] A. Roli and M. Braccini. Attractor landscape: a bridge between robotics and synthetic biology. *Complex Systems*, 27(3):229–248, Oct. 2018.

[50] A. Roli, M. Manfroni, C. Pinciroli, and M. Birattari. On the design of boolean network robots. In *Applications of Evolutionary Computation*, pages 43–52. Springer Berlin Heidelberg, 2011.

[51] I. Shmulevich, E. R. Dougherty, S. Kim, and W. Zhang. Probabilistic boolean networks: a rule-based uncertainty model for gene regulatory networks. *Bioinformatics*, 18(2):261–274, Feb. 2002.

[52] I. Shmulevich and S. A. Kauffman. Activities and sensitivities in boolean network models. *Phys. Rev. Lett.*, 93:048701, 4, 2004.

[53] The argos website. `https://www.argos-sim.info/concepts.php`. Accessed: 2020-05-30.

[54] A. Veliz-Cuba. Reduction of boolean network models. *Journal of Theoretical Biology*, 289:167–172, Nov. 2011.

[55] M. Villani and R. Serra. On the dynamical properties of a model of cell differentiation. *EURASIP Journal on Bioinformatics and Systems Biology*, 2013(4), Feb. 2013.

[56] D. J. Watts and S. H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393:440–442, June 1998.

[57] J. M. Whitacre. Degeneracy: a link between evolvability, robustness and complexity in biological systems. *Theoretical Biology and Medical Modelling*, 7(6), Feb. 2010.