Radboud University

Natural Evolution of Algorithms with Parameter Sensitive Performance

Thesis submitted in partial fulfillment of the requirements for the degree of master of science in artificial intelligence

Arne Wijnia

September 1, 2014

Supervisors:
dr. T. Wareham Memorial University of Newfoundland, St.John's, NL Canada
dr. I. van Rooij Donders Centre for Cognition, Radboud University Nijmegen

External examiner:
dr. J. Kwisthout Donders Centre for Cognition, Radboud University Nijmegen

Student Number:
3043762

Abstract

Computational-level theories of cognition often postulate functions that are computationally intractable (e.g., NP-hard or worse). This seems to render such theories computationally and cognitively implausible. One account of how humans may nevertheless compute intractable functions is that they exploit parameters of the input of these functions to compute the functions efficiently under conditions where the values of those parameters are small. Previous work has established the existence of such algorithms for various cognitive functions. However, whether or not these algorithms can evolve in a cognitively plausible manner remains an open question. In this thesis, we describe the first formal investigation of this question relative to the constraint satisfaction model of coherence. In our investigation, we evolved neural networks for computing coherence under this model. Our simulation results show that such evolved networks indeed exploit parameters in the same way as known tractable algorithms for this model.

Acknowledgements

I would like to thank my supervisors Todd Wareham and Iris van Rooij for their contributions to this thesis. It was great meeting in person in both the early and later stages of the project. Such meetings taught me a lot about work and discussions on a scientific level. It was great to see how you have different approaches that complement each other perfectly. I would also like to thank the TCS group (now known as the CCS group) for the interesting meetings, discussions and comments on the early stages of my thesis.

Lastly, I would like to thank everyone else who contributed to this thesis the last half year, knowingly or unknowingly, in many different ways.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

Many of the cognitive capacities that we use in everyday life seem to require no effort. For example, we can see, talk and have social interaction without conscious intervention. These tasks are not as easy as they seem, however, and it is remarkable how humans have evolved to do these things with such big success. Many of these cognitive capacities of humans are studied and captured into models to gain a better understanding of them. Marr was the first to describe cognitive functions and the levels on which they operate [12]. He had three levels in mind: the first one was the computational level. This described the function as a simple input output function. It explains *what* the function does. The second level is the algorithmic level and it describes *how* the inputs are transformed to outputs by an algorithm. The third and final level is the implementation level and explains the hardware behind it: how the algorithm is implemented in neurological structures.

Many cognitive capacities are described on the computational level, but for many it is hard to find suitable algorithms [2, 14, 26, 27, 30, 31]. This is mostly due to the fact that if the size of the input of these models grows (resulting in bigger problem instances), finding the matching output becomes increasingly harder. Because of this, such models are called intractable, which means that for anything but the smallest inputs the resources needed for computing such models are not realistically available [8]. One must thus find ways to work around the complexity of these models. Approximation algorithms and heuristics are a popular means of doing this. However, the validity of such methods is debatable with respect to cognitive science [32]. Furthermore, heuristics should provide near optimal solutions, but some are not nearly as optimal as is believed [11].

Another way to deal with the intractability of these models is by adopting the Fixed-Parameter Tractable (fpt) cognition thesis. This states that cognitive functions are among the functions that are fixed-parameter tractable for one or more input parameters that are small in practice [30]. Limiting certain parameters of the input will make the function solvable by algorithms running in time polynomial for the input and only superpolynomial for a function based on the parameter [4]. For many problems, such fixed-parameter tractable or fp-algorithms have been proven to exist [15]. Thus this approach is a promising way of revising a model in such a way that it becomes tractable.

However, finding the appropriate parameters to restrict and creating the fp-algorithms is not enough. First of all, fp-algorithms are not algorithmic level explanations: they just give a possibility for an algorithm, so one must first show whether or not that is the actual algorithm used by a natural cognitive agent. Secondly, fp-algorithms are not even psychologically plausible per se. It is not yet proven that they are actually possible in biological systems and for so far, they remain artificial constructs. Lastly, such algorithms cannot be assumed to be hand-coded inside the cognitive agent, rather they should evolve over time. This too is not yet proven or investigated.

This is the main focus and primary contribution of this thesis: is it plausible that cognitive agents have evolved algorithms that are fixed parameter tractable? In 2008, a small project was done to investigate the evolution of fp-algorithms [28]. However, the project only focused on a computational model that did not model a cognitive capacity and the results were not very clear. So in this project, we focus specifically on a model that is cognitively relevant and on a biologically plausible evolutionary process.

The model that we will take for this project needs to be a model of a cognitive function. Furthermore, we want it to be widely applicable for many cognitive capacities. For this reason, we chose to use the Coherence model as described in 1998 by Thagard and Verbeurgt [26]. This model is widely applicable to many human inferential abilities, such as perception, reasoning, belief, judgment, proof and explanation [25]. Coherence is so widely adaptable because it models

a capacity that we use to make sense of the world. Every interpretation can be captured as an instance of Coherence: we have hypotheses about what happens and compare those with what really happens. Some of those cohere and others do not. We can use that knowledge to determine which hypotheses are true and which are not true.

To investigate a plausible evolution of an algorithm for the Coherence model, we need to evolve a program that is based upon a biological cognitive structure. This is why we will get solutions for instances of Coherence by evolving Artificial Neural Networks (ANN) [34]. These networks are inspired by biological neural networks. This makes them great as a tool to study the possibility of naturalistic evolution, since they model a natural structure.

Because ANNs are non-standard algorithms, the evaluation of their performance is not straightforward. Standard fp-algorithms only assume that runtime will increase with an increase of certain input parameter sizes. However, this will not necessarily be the case for the performance of the ANNs. Rather than an increase of runtime, we might also see a decrease in solution quality. This will be the second contribution of this thesis: we will describe a new way to evaluate non-standard algorithms, based on the quality of their solutions, rather than on their runtime.

This thesis is organised as follows: I will first give some background on the model that is used throughout this thesis in the section about Coherence (Section 2.1), followed by a more in depth section about computational complexity theory and how it is applied in cognitive science (Section 2.2) and a final background section on artificial neural networks and evolutionary algorithms (Section 2.3). I will then detail the specific research questions (Section 3.1) and explain the notion of parameter sensitive performance as a way to evaluate algorithms (Section 3.2). Other methodology sections give more information on how Coherence can be represented in an ANN and the simulations performed to answer the research questions (Section 3.3 - 3.5). Finally, I will present results of these simulations (Section 4) and discuss the implications of these results (Section 5).

2 Background

In this section, key underlying themes of this thesis are introduced and explained. In the first part (Section 2.1) the model that we will use throughout the thesis (Coherence) will be outlined. Next, background will be given on concepts and techniques from computational complexity theory and how it is used in cognitive science (Section 2.2). The last section will cover the tools used to evolve solutions for Coherence (Section 2.3).

2.1 Coherence

The model of Coherence as constraint satisfaction was proposed in 1998 by Thagard and Verbeurgt [26]. Informally it can be described as follows: one has a set of propositions, or elements, called P. Elements are connected with each other by a set of constraints C. Each constraint is either positive, C^+, or negative, C^-, and connects two elements. One has to divide the elements into a set of accepted or "true assigned" elements and a set of rejected or "false assigned" elements, which results in an assignment A. This must be done in a way that satisfies as many constraints as possible. Positive constraints $(p, q) \in C^+$ are satisfied when both elements have the same assignment and negative constraints $(p, q) \in C^-$ if both have different assignments. More formally, the problem can be stated as follows:

> **Coherence**
> **Input:** A network $N = (P, C)$, where $C = C^+ \cup C^-$ is a set of positive and negative constraints and $C^+ \cap C^- = \emptyset$
> **Output:** A truth assignment $T : P \rightarrow \{true, false\}$ that satisfies a maximum number of constraints in C. (Here a positive constraint $(p, q) \in C^+$ is satisfied iff $T(p) = T(q)$, and a negative constraint $(p, q) \in C^-$ is satisfied iff $T(p) \neq T(q)$.)

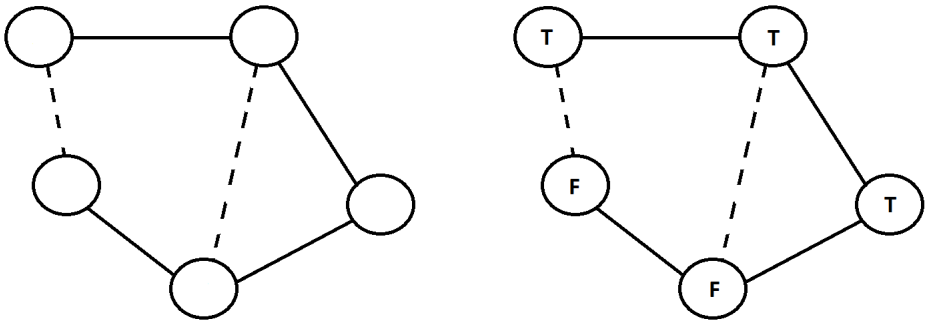
When searching for solutions, it is important to keep in mind that there are several different optimal solutions possible for the same problem instance and that none of those solutions might satisfy every constraint. Take the example given in Figure 1. The solution given in subfigure 1b is an optimal one, even though not every constraint is satisfied and one might find other solutions for this instance that satisify the same amount of constraints. In practice, finding the optimal solution for Coherence instances has proven to be computationally difficult, as the number of possible assignments of the elements grows rapidly with each added element and checking all assignments to determine those that satisfy the maximum possible number of constraints is prohibitively time-consuming. This makes the model intractable, which as described in the introduction, is problematic. In Section 2.2, we will consider how to both prove and handle this intractability.

2.2 Computational complexity theory and cognition

In this section, the basic background to computational complexity theory and how it is linked to this project will be covered.

2.2.1 Computational complexity theory

Computational complexity theory is used to assess the computational difficulty of problems based on the resources required [8]. A computational problem is a specification of a set of inputs and the output associated with those inputs. An algorithm for such a problem specifies how, given any input of that problem, to compute its corresponding output. The complexity of a problem



(a) A coherence network (b) A solution

Figure 1: Coherence instance example. This figure shows a Coherence instance with 5 elements, 2 negative constraints (the dashed lines) and 4 positive constraints (the solid lines) in part a. In part b, a solution is shown, where each element has been assigned either true or false. Note that this is an optimal solution, even though not every constraint is satisfied.

is based on the increase of needed time and resources to find the output as the size of the input increases. The time needed for an algorithm to transform the input to the output in the worst case is described by the *Big-Oh* notation. This is always a function of the input and it thus describes how the time needed to find the output is correlated with the size of the input. Some problems can be solved in polynomial time, which means that there exist algorithms for them that run in polynomial time, or in $O(n^x)$, where n is the input size and x is a constant. The class of decision problems (problems where the question is whether a solution is better than a given constant and the answer is either "yes" or "no") which can be solved in polynomial time is called P. Not all problems have a solution that can be found in polynomial time. There is another class of decision problems called NP. For each problem in this class, it is possible to check in polynomial time where a given candidate solution is in fact a valid solution to the problem. Many problems are in NP, including the decision version of Coherence.[1] Note that each decision problem in P is also in NP, but it is assumed that not every problem in NP is also in P, in other words, it is assumed that $P \neq NP$ [7].

The classes P and NP can be used to prove why certain problems are not tractable. Such problems are called NP-hard. This means that for a problem of this class, every other problem in NP can be reduced to that problem. Such a reduction from problem A to problem B is valid if there is an algorithm transforming an instance x of A to an instance y of B such that the algorithm runs in polynomial time and the answer to x is "yes" if and only if the answer to y is "yes". Furthermore, a problem is NP-complete if it is both in NP and it is also NP-hard. Problems that are NP-hard cannot be in P, unless $P = NP$ (because every problem in NP is reducible to a NP-hard problem and this would mean that every problem in NP is reducible to a P problem and thus $P = NP$) and this means that Coherence, which is NP-complete, cannot be solved by any polynomial time algorithm unless $P = NP$. Because of this, complexity models that are NP-hard, like Coherence, are called *intractable*.

[1]For Coherence, given a candidate truth assignment, one has only to count the number of constraints satisfied by that assignment, which can be done in polynomial time.

		$2^k \cdot n$		
n	2^n	$k = 1$	$k = 5$	$k = 10$
1	2	2	32	1024
5	32	10	160	5120
10	1024	20	320	10240
25	33554432	50	800	25600
50	1.1×10^{15}	100	1600	51200
100	1.3×10^{30}	200	3200	102400
1000	1.1×10^{301}	2000	32000	1024000

Table 1: Polynomial, exponential and parameterized running times.

2.2.2 Computational complexity in cognitive science

As explained in the introduction, many cognitive capacities are described on the computational level, but for many it is hard to find suitable algorithms. Take for example Coherence again: although this model seems able to explain many cognitive capacities, the model itself is NP-complete, meaning that no polynomial time algorithms exist that can find the optimal solution for all instances unless $P = NP$ [26]. Without a polynomial time algorithm, it can only find solutions for very small instances in reasonable time. This is an obvious problem, because it is believed that humans can also solve large Coherence networks [26].

There are several ways to overcome the problem of intractability. Instead of always going for the optimal solution, one can use heuristics or approximation algorithms to find solutions that are suitable enough, because these solutions for example only differ from the optimum by a maximum percentage [1]. However, the quality of solutions produced by such algorithms is often poor [11, 32]. Another way to overcome this complexity problem is by changing the current model in such a way that it still models the cognitive function, but has now possible algorithms that can solve problem instances in a feasable time, despite the input sizes [4]. A popular revision is through *parameterization of the input*, where certain assumptions are made about the input in such a way that the runtime is polynomial for the input and superpolynomial only for certain parameters of the input. By restricting these parameters, the algorithm can find solutions in reasonable time, no matter how big the input gets and the problem is now *fixed-parameter tractable* relative to these parameters. To illustrate this, Table 1 shows how running time increases with input size for polynomial, exponential and parameterized running time algorithms.

For Coherence, several fp-algorithms have been found as well. It is known that restriction of the parameters s (number of satisfiable constraints), u (number of unsatisfiable constraints) and $|C^-|$ (number of negative constraints) makes finding a solution for any instance easy [29,30]. For example, if the number of negative constraints in an instance is low, it is easy to find the optimal solution for the instance, regardless of its size [29]. A description of this algorithm can be found in Appendix A.

The biggest challenge for fp-algorithms is finding the parameters to restrict, because the number of possible parameters to restrict is huge as it can be any aspect of the input. In the case of Coherence, possible parameters include the number of constraints, cycles, or isolated elements. Suitable parameterizations of the input for cognitive functions thus involve finding the appropriate parameters to restrict. The biggest trick for parameterization of the input is thus finding parameters that are the possible cause of the intractability, testing whether or not a restriction of these parameters really helps to improve algorithm performance and last but

not least also test if it is plausible that such parameters are small in everyday life inputs (and humans can thus only encounter instances where these parameters are restricted).

However, classical fp-tractability only assesses algorithms based on their increase in running time. This is for this project not sufficient, because we will use artificial neural networks that will try to find optimal solutions in a fixed number of generations. If the task is hard, this evolution process will likely not slow down, but simply find a less perfect solution in the same number of generations. It is thus very improbable that instances with many negative constraints will cause the network evolution to slow down in each generation, or will take longer for an evolved network to be processed. Rather, they will decrease the networks solution quality. More on an alternative way to assess the performance of the artificial neural networks will be given in Section 3.2.

2.3 Artificial Neural Networks and Evolutionary Algorithms

Artificial Neural Networks (ANN, but also simply called Neural Networks (NN) or just networks) are computer programs inspired by the human brain where a set of input neurons receives certain outside signals. This technique has evolved from simple one layer systems [13], to feedforward Multilayer Perceptrons (MLP) [18] to Recurrent Neural Networks (RNN) [10]. The basis of ANNs is this: input neurons have weighted connections with other neurons. In the end, the connections link to output neurons that give an output corresponding to the input. By updating the weights and the connections between neurons, ANNs can find increasingly better input-output mappings.

An important decision to make when using ANNs is whether one wants to use feedforward or recurrent ANNs. Feedforward networks have no cycles and thus no connections from a layer to a previous layer. Recurrent networks have such connections. There are several differences in performance of these networks and their capabilities. In this project, recurrent networks will be used, because it is known that these networks can in principle solve NP-complete problems like Coherence, whereas feedforward networks cannot [19, 20].

In more recent years ANNs have become more relevant because the computing power of computers is rapidly increasing. This allows for larger networks and thus the search for solutions of more complex problems. Although there is still a long way to go before networks with sizes that mirror the human brain can be trained, ANNs remain highly studied tools of machine learning, in part because of their growth potential in the future. Furthermore, this means that they can be applied to increasingly difficult (cognitively relevant) problems.

In this project we are interested in natural evolution of algorithms for a cognitive capacity. ANNs are great for this, because they are inspired by a natural system that has evolved over hundreds of millions of years: the brain. To further emphasise the natural evolution, the ANN will be updated with techniques from Evolutionary Algorithms (EA) [5, 6]. Such algorithms are based on elements of natural evolution, like mutation, cross-over and fitness. The terms used in this section are terms one can find in biology as well. The programs that are evolved are called the genotypes and are built up from smaller pieces called the genomes. The genotype represents genetic code that will ultimately determine what the end result looks like. This actual end result is called the phenotype. Sometimes the genotypes map directly to phenotypes, which is for example the case with genetic programming where the outcome is a program. In other techniques, like the evolution of ANNs, the networks produce a result that looks quite different from the networks themselves.

Even though many forms of EA exist, the basis is always the same. It is a stepwise process listed below.

1. Generate set of initial genotypes. A set of genotypes is first generated. This is called the population and each of those genotypes is an individual within the population.

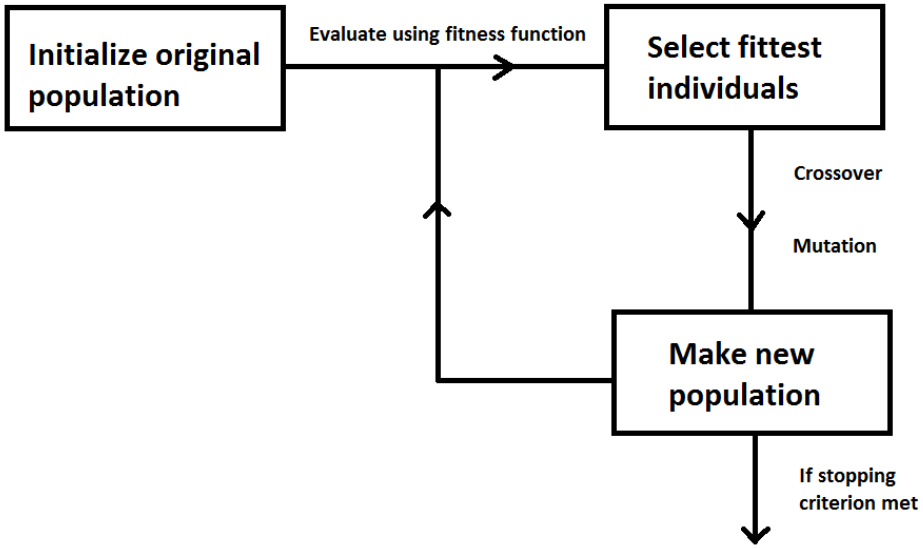


Figure 2: Process of an evolutionary algorithm. This figure gives an overview of an evolutionary algorithm.

2. Evaluate individual genotypes via a fitness function. Each individual is evaluated using a fitness function. The fitness function is thus one of the most important elements of the EA, because it determines which individuals are good and which are not. It is up to the programmer to decide what makes an individual good, but it is usually a combination of things. These include the genotype's ability to solve the issue at hand correctly, the time it needs and the size of the solution. In the fitness function, one must often make a trade off between very good solutions and fast solutions. After an individual is evaluated, the EA generates a new population.

3. Generate new population via mutation and recombination with respect to fit individuals. The new population is based on the best performing individuals of the previous population. Each of these best individuals can be slightly changed using crossover or mutation. In crossover, a genome of one genotype is swapped with that of another genotype, creating two new genotypes. In mutation, a genome of a genotype is randomly changed. All of these new genotypes and possibly some old ones from the previous population form a new population.

4. Stopping criterion. The result of the processes in steps (2) and (3) above is called a generation and as many generations can be created as one needs. A stopping criterion is used to end the process at a certain point. This can happen after a fixed number of generations, or when a genotype is found that encodes a good enough solution.

The general evolutionary process is illustrated in Figure 2. The evolution process can be influenced by many factors, such as the value of parameters like mutation rate, crossover or population sizes, or the choice of fitness function or stopping criterion. It is problem dependent to determine the optimal settings for these parameters.

As was mentioned at the beginning of this subsection, many forms of EAs exist. For this project, genetic algorithms are used. They evolve programs that produce a specific output that is a direct solution to a problem instance. If we link this to the artificial neural networks, these networks form the individuals of the population. Each neural network will produce an output and the ones that produce the best output will be used to generate a new population. In the end, only the best ANNs remain, specifically evolved and trained to generate solutions for instances of Coherence. How this is exactly implemented will be covered in Section 3.3.

3 Methods

In this section, the methods used in this thesis are described, starting with a further refinement of the research questions (Section 3.1). Parameter Sensitive Performance is then introduced as a new way to evaluate algorithms (Section 3.2). The latter part of this section will cover the implementation of Coherence in the ANN (Section 3.3), the evolution of these networks (Section 3.4) and the instances used in simulations (Section 3.5).

3.1 Research questions

The main question from the introduction can be summarized as follows:

> Is is plausible that cognitive agents have evolved algorithms that are fixed-parameter tractable?

Given the concepts introduced in the background we can make this research question more concrete. In the background we described that we will use the model of Coherence in this project and we will use ANNs to evolve solutions for Coherence instances. Furthermore, fp-algorithms of Coherence were described, where input parameters are exploited. This leads to the more specific research question that will be used in this project:

> Can we evolve artificial neural networks for Coherence instances in such a way that they exploit input parameters which are known to render Coherence fixed-parameter tractable?

To answer the main question, we have to separate it in two subquestions. We decided to look at both the $|C^-|$ parameter and the u parameter[2]:

Q1. Does the $|C^-|$-parameter have influence on network performance, either in training or testing?

Q2. Does the u-parameter have influence on network performance, either in training or testing?

3.2 Parameter Sensitive Performance

In this project, algorithms were evolved for instances of Coherence. The main goal is to investigate whether or not these algorithms exploit the same parameters which are known to render the problem fp-tractable. This would mean that the algorithms perform better on instances where these parameters are small in size. However, this is not easy to assess, because the algorithms used are neural networks and the outcome of the neural network is not an algorithm that can be checked for its runtime properties. Rather, the outcome will be a truth assignment over the elements. Because of this, the performance of the neural networks has to be assessed directly, instead of assessing the output. This means that we will feed many instances to trained networks and assess the performance of the network on these various instances, expecting different results when the networks are tested on instances with high parameter sizes (for which fp-algorithms are known) compared to tests with instances with low parameter sizes.

There are three effects that can occur when the neural networks are tested on instances with larger parameter sizes:

[2]The alert reader will notice that we did not investigate parameter s for which Coherence is also known to be fp-tractable. While such an investigation would indeed be of interest, we were not able to do it because of lack of time, and hence leave it for future work.

1. Increase in running time. As with traditional fixed-parameter tractability, the algorithms could have need of more running time to compute solutions for instances with large parameter sizes.

2. Increase in algorithm size and complexity. The neural network might evolve more nodes, or more connections.

3. Decrease of solution quality. It might also mean that the quality of solutions decreases. In the example of Coherence instances, it might mean that the algorithm finds solutions which satisfy less constraints than the optimal solution. If the time that the algorithm gets to solve the problem is fixed, one would expect that the solution time decreases: the algorithm has to find a solution to a more difficult instance in the same time and will thus likely get poorer solutions.

Since the number of generations that the neural network has is fixed it is very likely that the solution quality will decrease with an increased size of parameters. After all, if the computation time is not likely to increase with an increased size of input parameters for which fp-algorithms are known, the network must deal with this increase in a different way. Since it still uses the same number of generations to deal with instances that are now more difficult to solve, the only way it can do so is by producing results of lower quality. The more difficult the instance, the longer it takes to find a solution of comparable quality to those instances that are easier. Because the process is stopped after a fixed number of generations, it will simply not have found an equally good solution by then. Because of this, we will evaluate how the solution quality varies with certain parameter sizes.

At this point we introduce a new evaluation criterion for algorithms, based on solution quality rather than running time. It is largely similar to fixed-parameter tractability: if certain parameters remain small, the solution quality will not drastically decrease and runtime will not drastically increase as a function of the input size. The only difference now is the emphasis on the solution quality. The term that we will use for this new evaluation is: parameter sensitive performance.

Parameter Sensitive Performance (PSP)
An algorithm exhibits parameter sensitive performance relative to parameter p if as the value of p increases, the runtime drastically increases and/or the solution quality drastically decreases.

Regarding Q1 and Q2, we can now phrase hypotheses based on the notion of parameter sensitive performance as explained here. For Q1, it is likely that many negative constraints in a test instance will lead to poor performance of the network:

HYPOTHESIS:
Networks trained on instances with few negative constraints cannot perform well when tested on instances with many negative constraints, because they have not evolved to handle these. In general, it is expected that networks test better on instances with few negative constraints, regardless of how they are trained.

For Q2, similar results are expected:

HYPOTHESIS:
Networks test better on instances with few unsatisfiable constraints, regardless of how they are trained.

3.3 Implementing Coherence instances in neural networks

To answer the research questions Q1 and Q2, an artificial neural network needed to be created. These networks have several layers. The first layer is a set of input nodes I. A second layer consists of output nodes O. Extra layers are possible and these contain hidden nodes H. The amount of nodes and their connection with one another is called the topology of the network. Since a Coherence instance can uniquely be defined by the layout of its constraints, it made sense to have the input neurons map to constraints. The output should be a truth assignment over all elements, which is why we decided to have each output neuron map to an element.

An example of an implementation of a Coherence instance using the instance of Figure 1a can be found in Figure 3. On the left of the figure are the input nodes. These represent the constraints, in such a way that each constraint maps directly to a specific input node.

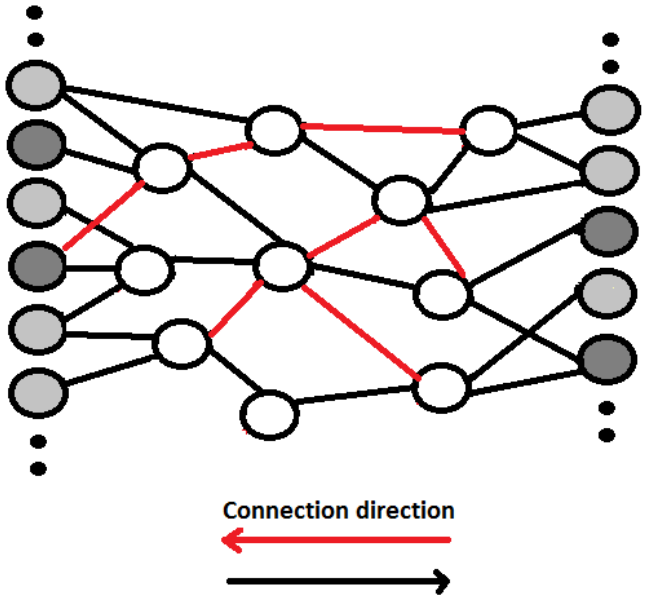


Figure 3: Coherence implemented in a neural network. The original instance found in Figure 1a has six constraints. These are the input nodes. The ones that are darker gray are negative constraints with value -1 and the lighter ones are positive constraints with value 1. On the right, the output nodes represent the elements. The lighter elements are assigned true and the darker ones false. Note that there are many more input and output nodes (represented by the black dots on either side of the neurons), but these are not relevant for this instance and are thus given arbitrary values. In the middle, the white nodes are hidden neurons. The red lines represent the recurrent connections of the network and the black lines feedforward connections.

To allow the network to handle Coherence instances of varying sizes a clever solution needed to be found for the amount of input nodes. If one trains a network with 10 input nodes, then that network will only be able to handle Coherence instances with these amount of constraints. This very much limits the possibilities, because each differently sized Coherence instance would need its own neural network. To overcome this problem, the amount of input nodes was set to a large amount, so it could handle many differently sized Coherence instances. For those that have fewer than the maximum number of constraints, the input nodes not used for constraints were given arbitrary values. In Figure 3 this is illustrated by the black dots, which represent input neurons that are not used by this particular instance.

The output nodes represent the assignment of the elements, so each output node maps directly to an element. Like the input nodes, not every Coherence instance will need to use each output node. This means that although the number of output nodes is fixed, the number of used output nodes is not. On small instances, only a few output nodes were evaluated for their values. On larger instances, many, or all output nodes were evaluated.

As one can see, the only information fed to the network is the number of constraints and the only output is a value for each element being either true or false. The hidden nodes of the network (in Figure 3 shown as the two white nodes in the middle) are evolved by the evolutionary algorithm and are independent of the instance.

3.4 Evolving artificial neural networks for Coherence instances

There are several ways in which one can update the weights of an ANN and backpropagation is one of the more popular ones [18]. However, such methods end up in local minima often and can only update weights and not network topology [9]. Hence, the network topology has to be set manually, which can be tricky. Evolutionary algorithms are therefore a popular means of evolving both weights and topologies, because they do not necessarily end up in local minima [33]. For this project we wanted to give the network as many options as possible, so we allowed evolution of both topology and weights. This also resembles a more natural process compared to standard learning rules and manually setting topologies.

An effective way of evolving both weights and topology is the NeuroEvolution of Augmenting Topologies (NEAT) framework, developed by Stanley and Miikkulainen [24]. NEAT is specifically designed to overcome the main problems of standard topology evolving techniques. The following subsections will highlight relevant features of NEAT. In Section 3.4.2, the in this project used extension of NEAT, HyperNEAT, will be described.

3.4.1 The NEAT framework

The main focus of NEAT is to not only evolve the weights of a neural network, but also to evolve the network topology itself. The bases for such topology evolving networks can be found in TWEANNs (Topology and Weight Evolving Artificial Neural Networks) which already existed for several decades. However, the TWEANNs had certain shortcomings. In this section I will shortly discuss the different problems of TWEANNs and how NEAT is able to solve these problems.

Competing convention problem The competing conventions problem, also known as the permutation problem [17], is one of the main problems in neuro evolution. The problem entitles that the way in which a solution to a weight optimization problem of a neural network is found is not unique. This can be problematic, because if the encoding for genomes that represent the same solution is not the same, then crossover tends to lead to damaged offspring. For example,

if the encodings [A,B,C] and [C,B,A] would be used for a crossover, [C,B,C] could be the result. This result would have lost one third of the information that was available in the parents.

To generalize, if one has n hidden neurons, one will have $n!$ different solutions with the same functionality. Therefore it is important to come up with a representation that facilitates crossover and at the same time does not allow multiple representations of the same solution. The representation as used by NEAT is based on the principles of historical markers. These markers are used to trace the origin of genomes.

Speciation In order to create innovation in TWEANNs, mutation is used to add new structures to the existing network. While the innovation is intended to boost the performance of the network, more often it initially decreases the fitness of the network. For example the addition of new connections can reduce the fitness before the weights get the chance to optimize. The chance that a new node or connection immediately expresses useful functionality is very small. Therefore, some generations are required in order to give the mutation the chance to turn the network into a useful solution. The initial loss of fitness however, makes it less likely for the mutation to survive long enough in the population in order to optimize. This shows that it is very useful to be able to protect innovative structures within a network, to give it a chance to bloom into useful functionality.

NEAT uses a different approach in order to protect innovation based on biological principles. In nature, different species consist of different structures, which compete in different niches. Therefore the innovation within each structure is being protected by its niche. The same principle can be applied to neuro evolution. If networks of different structures only have to compete within their niche they get the chance to optimize their innovation in order to be able to compete with the rest of the population. The problem that remains is, how to know if a certain network belongs to a certain niche? This is where the historical markers, which where created in order to solve the competing conventions problem, come into play. These historical markers make it easy to categorize the different networks into different species. In this way, NEAT has the ability to protect innovation with the help of speciation.

The initial population The way in which many TWEANN systems generate their first population is by simply collecting different random topologies. This is done in order to make sure that the diversity of topologies in the system is large from the start. But this approach also causes the inclusion of useless networks which do not have a path from the input nodes to the output nodes. This is because the connections and nodes in these random topologies are not subjected to any evaluation before they are included in the starting population. To prevent the inclusion of networks without a connecting path between the input and the output neurons, NEAT evolves networks from minimal solutions. This way the search space is kept to a minimum, which reduces the computational load placed on the system.

3.4.2 The HyperNEAT framework

Neural structures like the human brain have millions of neurons with trillions of connections. Techniques like NEAT are unable to evolve structures of this size. That is why Hypercube-based NeuroEvolution of Augmenting Topologies (HyperNEAT) was introduced by Stanley et al. [23].

The biggest difference between the networks trained with NEAT and the large biological neural structures like our brain is the fact there are many motifs and similarities in our brain, like cortical brain parts, that have the same kind of structure [21]. Furthermore, the organisation that the brain has often reflects natural geometry. This means that features of the outside, physical world are reflected in brain structures such as for example the retinotopic organisation

Property	Value
num generations	10
population size	250
neuron mutation rate	0.1
connection mutation rate	0.8
weight mutation rate	0.1
survival rate	0.3
crossover proportion	0.4
recurrent	best guess
recurrent cycles	1

Table 2: Properties used in the evolution process.

of the visual cortex [3]. On the contrary, networks evolved with NEAT have less organisation and regularity. HyperNEAT uses connective Compositional Pattern Producing Networks (connective CPPNs) to evolve large networks that have regularities [22]. This is the reason why we will evolve networks with HyperNEAT in this project: it resembles the natural system better and the networks are capable of handling large inputs. For example, Coherence instances with more than 20 elements can be given as an input for a HyperNEAT network, whereas standard NEAT hits its limit already at networks of size 7.

Both NEAT and HyperNEAT work with a properties file, where many properties for the evolution process can be set. These include population sizes, initial population properties, evolution properties and properties specific for HyperNEAT. Some of the most important properties and their values used in this project are listed in Table 2. For an explanation of these properties and their details the interested reader can refer to [6, 23].

3.4.3 Fitness function for evolving Coherence as neural network

An important part of evolutionary algorithms is the fitness function. The fitness function determines which individuals of the population are best and will be used in the next generation. In this project, individuals are best if they have a good solution quality. So the more constraints that are satisfied of any given instance, the better a network is. Besides solution quality, there are other very common fitness function elements, two of which and the reasons why they are not used are highlighted below.

1. The size of the network. Evolving topology can lead to huge networks containing many nodes and connections very quickly. This problem, called bloat, is also very common in genetic programming. Such larger networks do not have necessarily better fitness and are harder to interpret [16]. A small penalty for larger networks in the fitness function is common practice to prevent bloat of networks. However, NEAT has the property of evolving simple networks automatically, because it starts with the most simple networks. This means that the size of the network need not be a part of the fitness function.

2. Convergence time. Early tests (see Section 4.1) indicated that the networks already converged towards a solution within a few generations, so convergence time was not a necessary part of the fitness function. Besides, we do not aim for a specific solution quality, but rather a quality within a fixed number of generations. So convergence time is irrelevant.

Given the above, the fitness function used in our project need only measure solution quality. Solution quality will be measured by the amount of constraints satisfied, C_{sat}, in respect to the maximum number of possible constraints that can be satisfied, C_{maxsat}, as shown in Equation 1.

$$fitness = \frac{C_{sat}}{C_{maxsat}} \tag{1}$$

3.4.4 Implementation details

Based on the previous sections about EA, NEAT, HyperNEAT and the fitness function, the evolutionary algorithm for this project can be created. In Algorithm 1 an overview is given of this algorithm's most important part: the evaluation of individuals. A problem instance is resembled by its constraints and is stored in a constraint matrix. At the start of the program, a new instance is retrieved and changed to a network input. With this network input, the neural network determines a corresponding output. The number of satisfied constraints in the output is determined and the fitness is calculated by comparing this number to the overall number of satisfiable constraints. Thanks to the HyperNEAT package used (see http://eplex.cs.ucf.edu/hyperNEATpage/ for information on this package), these steps are the only ones that needed to be implemented. Besides a specification of input, output and a fitness function, HyperNEAT can do everything itself, based on the values of the evolution properties (such as those from Table 2) set by the user.

Algorithm 1 HyperNEAT evaluation of a network

```
satisfiedconstraints = ∅
for all instances ∈ instanceset do
  networkinput = constraintmatrices.get(instance)
  networkoutput = substrate.next(networkinput)
  if networkinput[ij] = 1 && networkoutput[i] = networkoutput [j] then
    satisfiedconstraints += 1
  else if networkinput[ij] = -1 && networkoutput[i] = 0 or 1 && networkoutput[j] = 1 or 0
  then
    satisfiedconstraints += 1
  end if
end for
fitness = satisfiedconstraints/maxsatisfiedconstraints
return fitness
```

3.5 Coherence instances used in simulation studies

The Coherence instances used in the simulations needed to vary both in topology and in the sizes of the parameters $|C^-|$ and u. However, a suitable pool of instances ready to be used was not yet available. This meant that for the purpose of this thesis, instances needed to be generated.

The biggest issue was that evolutionary algorithms need to have the optimal solutions for used instances of Coherence to compare with the solutions produced by the evolved neural networks in order to evaluate the fitness of these networks (see Section 3.4.3). This means that for each generated Coherence instance, the optimal solution had to be found. As Coherence is NP-complete, only solutions to instances with a maximum of 20 nodes could be found by a

standard exhaustive algorithm. Implementation of an fp-algorithm, like the one proposed by van Rooij [29] and discussed in Appendix A, was of no use, because the instances needed in the simulations should also have the option of large parameter sizes.

The instances generated have the following sizes: 8, 12, 16 or 20 nodes. The amount of constraints is 50% of the total number of possible constraints, randomly placed between the elements. Furthermore, each of these instances has a varying number of negative constraints ($|C^-|$) and satisfiable constraints (s), which will be highlighted in the result section because they are simulation specific. Of each specific combination of settings, 15 instances were generated. The settings are summarized in Table 3.

Instance property	Value
Number of elements	8, 12, 16 ,20
Percentage of possible constraints	50
Percentage of negative constraints	10, 20, 50, 80
Number of unsatisfiable constraints	10, 20, 45, 70
Number of instances per setting combination	15

Table 3: Generated instance properties.

4 Results

In this section, the simulations performed to answer Q1 and Q2 are described, including their results (Section 4.2 and Section 4.3). We start off however with some simulations that were used to determine characteristics of the network. The results of these simulations determined which simulations would be useful to answer Q1 and Q2.

4.1 Determining characteristics of network training and operation

In these simulations, we determined characteristics of network training and operation. Before simulations specifically linked to the research questions could be conducted, it was important to know how the networks would evolve, how many runs would be needed to get a representative result, what general parameter settings were needed and what factors other than $|C^-|$ and u could influence the performance. The simulations were divided into the following four groups, each of which investigated a different characteristic of network training and operation:

1. Computation time. As stated previously, the network was given a fixed number of generations to find a solution during training. During testing, the instance was fed through the network just once. This means that computation time would likely not vary, regardless of what kind of instances would be fed to the network. However, this was not a certainty, because bigger instances, or instances with larger parameter values of $|C^-|$ and u could still take longer to evolve, because each generation would take longer. These few simple simulations were used to determine whether or not computation time was a factor worth evaluating our networks on, rather than just the quality of their produced solutions.

   The results in Table 4 show the time one training generation took for several combinations of size or $|C^-|$. These results showed that regardless of instance size, or of parameter $|C^-|$ size, the time to train a network did not increase.

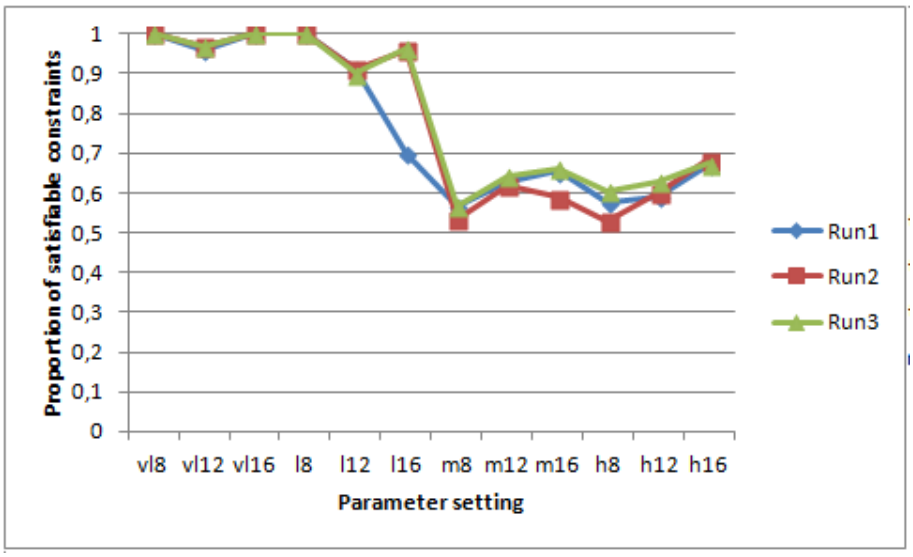
Elements	C^- %	Time (s)	Elements	C^- %	Time (s)
8	10	942	8	50	942
12	10	943	12	50	943
16	10	943	16	50	944
20	10	944	20	50	944
8	20	944	8	80	945
12	20	944	12	80	946
16	20	943	16	80	947
20	20	943	20	80	947

Table 4: Computation time simulations results. Time for 1 generation for training on instances of various sizes and percentages of C^- constraints. Time is in seconds.

2. The stability of the evolution process. These simulations were used to determine the stability of the evolution process, by evolving several networks on the same kind of instances and checking for differences. If the evolution process is stable and consistent, networks trained on similar instances should perform similarly when tested on similar instances. Such a result would mean that few runs would be needed in following simulations, because there is little variability between them in the first place. As is shown by the results in Figure 4, networks trained on instances with the same parameters and networks tested

on instances with the same parameters showed almost no variability between runs. This means that each run produced a network that got the same kind of results on the same kind of instances. In the figures, the different parameter settings are presented as follows: vl, l, m, h (very low, low, medium, and high, respectively) mean 10, 20, 50, 80 percent negative constraints respectively. The number that follows represents the number of elements of the instance used.

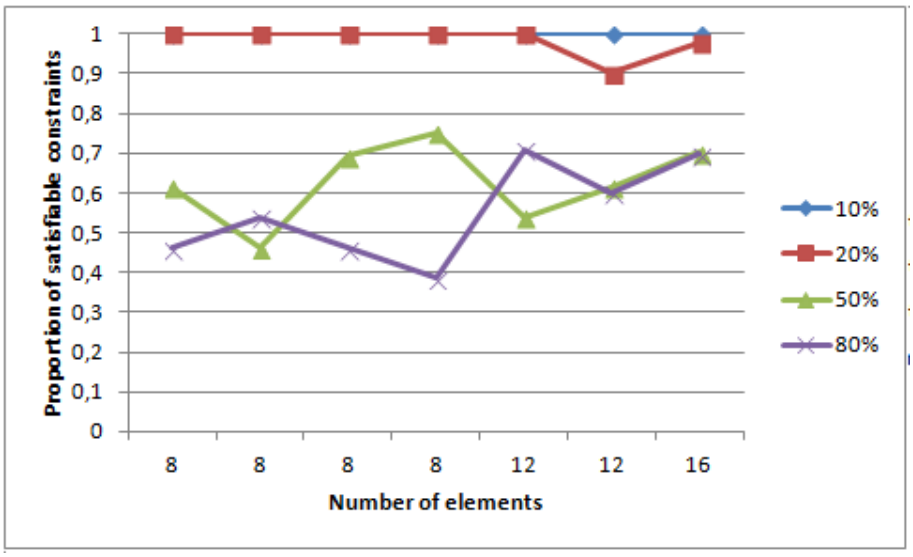
(a) Train results evolution consistency.



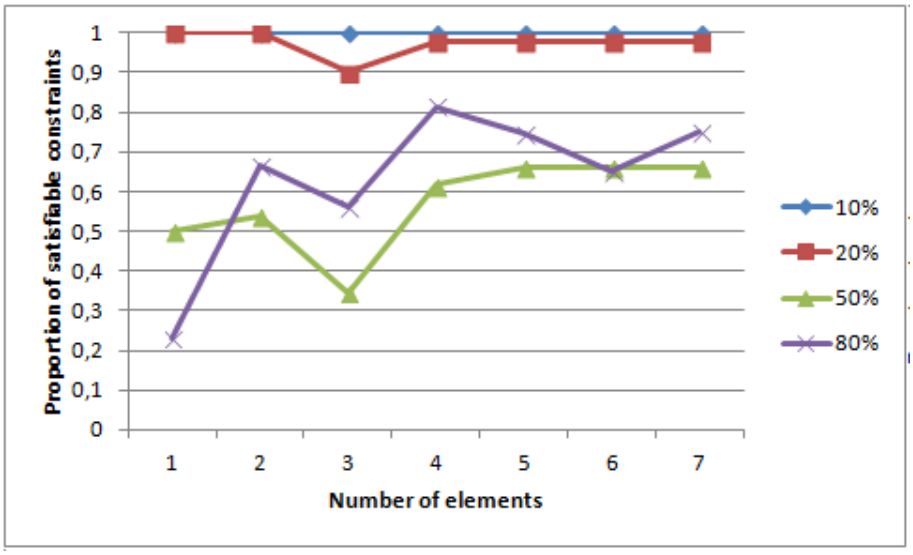
(b) Test results evolution consistency.

Figure 4: Evolution consistency tests. These networks were trained and tested on the same instances. There are three evolutionary runs and as can be seen, there is almost no difference between the runs (for each setting, they all score roughly the same fitness value). So the graph shows that the evolution process will find similar results when trained and tested on similar instances. On the x-axis are several parameter settings, where the letters represent C^- percentage (vl = 10, l = 20, m = 50, h = 80) and the numbers the elements.

3. Instance size. The third group of simulations was used to determine the influence of instance size (number of elements) on network training and testing. These were important simulations, because an fp-algorithm (or a network with such characteristics) should not show major performance differences when trained or tested on differently sized instances. In these simulations, instances with 8, 12 and 16 elements were used and with 10, 20, 50 and 80 percent negative constraints. The results in Figure 5a show that networks trained on instances with few elements could easily handle instances with more elements. When trained specifically on larger instances, the network had poorer solution quality when it was tested on smaller instances. However, this was, as shown by Figure 5b an almost equal quality when compared with larger instances.



(a) Results of training on small instances (8 elements) and testing on instances with various sizes.



(b) Results of training on large instances (16 elements) and testing on instances with various sizes.

Figure 5: Results of networks trained and tested on instances with various sizes. These networks were trained on instances with a fixed number of elements. The results show the performance once the networks are tested on instances with 8, 12 or 16 elements. This was done for four different negative constraint percentages, which are shown in the legend on the right of the figures.

4. Recurrency. As mentioned before, recurrency is an important part when one is designing an ANN. If the network has the option of recurrency, it means that a node in a layer can have a connection back to a previous layer, thus creating cycles. In HyperNEAT, one has the option to set the number of times a cycle is run. This means that the number of

recurrent connections cannot be set and is up to the evolution. However, each recurrent cycle can be looped for a fixed number of times.

These simulations were conducted to investigate the influence of recurrency on network performance, in this case the quality of the solutions. These tests were important because as stated before, Coherence can only be effectively solved by recurrent networks (see Section 2.3). This meant that the solution quality would likely increase with more recurrent cycles. The number of allowed cycles in the simulations is either 1, 5 or 10. As Figure 6 shows, solution quality is almost the same, whether 1, 5 or 10 cycles were used.

The results of these simulations showed that an increased number of recurrent cycles had no effect on solution quality. The networks were trained on instances with high percentages of negative constraints, because the performance of such trained networks was relatively low. It would thus be the best candidate to improve when more recurrency was allowed, because it could use the most improvement out of all trained networks.

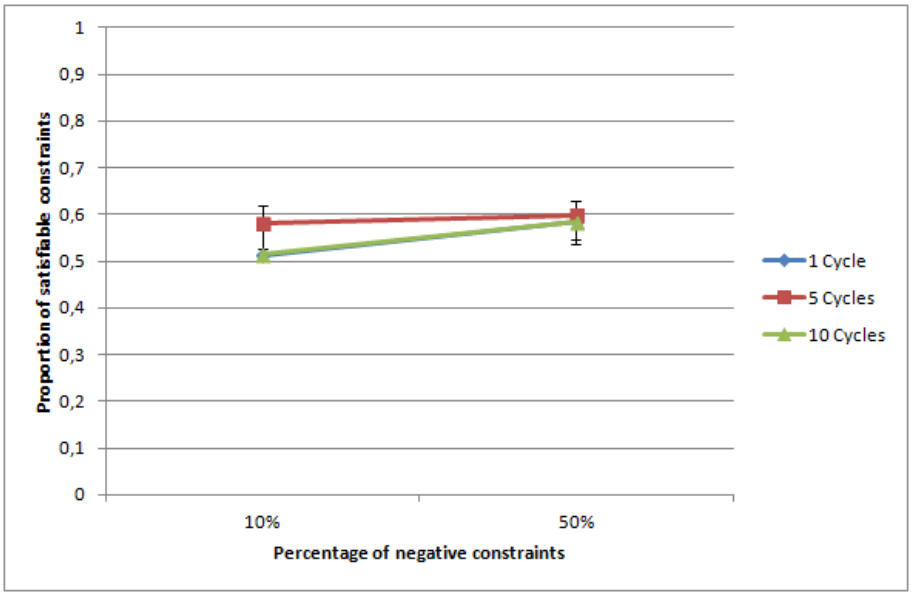


Figure 6: The results of recurrency simulations. These results show that more cycles through recurrent connections does not improve solution quality. The networks were tested on instances with low percentage negative constraints and medium percentages (10 and 50 percent) and in both cases the results are nearly identical. These are the averages over three runs, with error bars indicating the extremes of the averages.

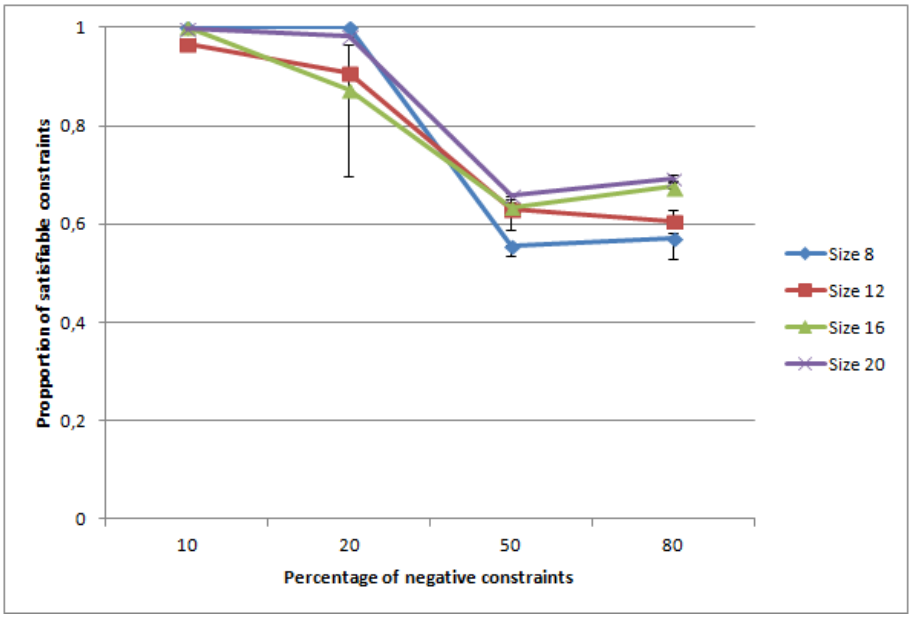
The simulations in the following subsections about $|C^-|$ and u were directly influenced by the results of these simulations. The number of runs, the sizes of the networks and the number of recurrent cycles used in the following subsections were derived from those results.

4.2 Q1: Simulations for the influence of parameter $|C^-|$

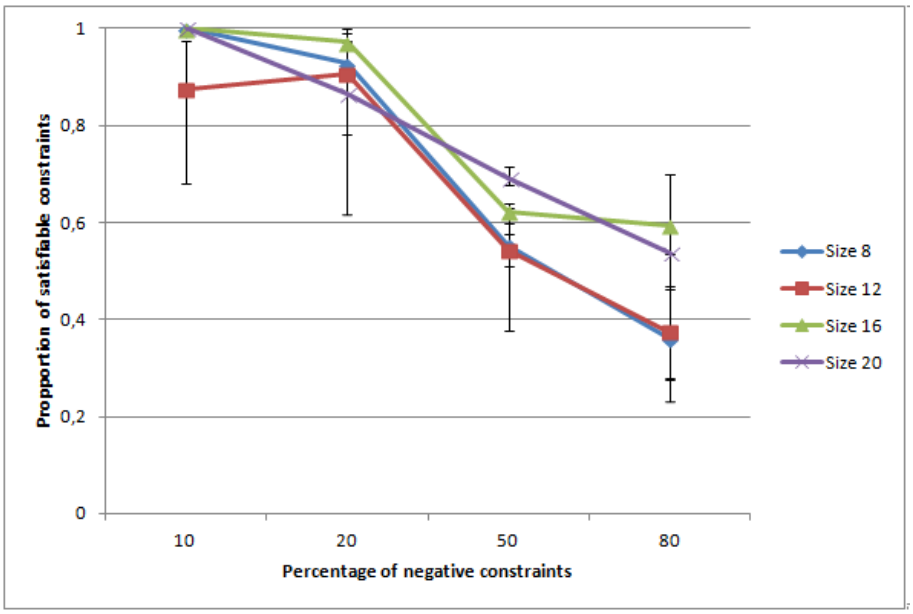
In this subsection, the simulations that were performed to investigate the influence of the parameter $|C^-|$ and their results are shown. Instances with 8, 12, 16 and 20 elements were used and with 10, 20, 50 and 80 percent negative constraints and networks were trained in various ways.

Each simulation was done three times, to account for variability in the evolution process. The results of these simulations can be found in Figure 7. Note that these results are the averages over 3 runs. Error bars display the maximum and minimum values of the runs.

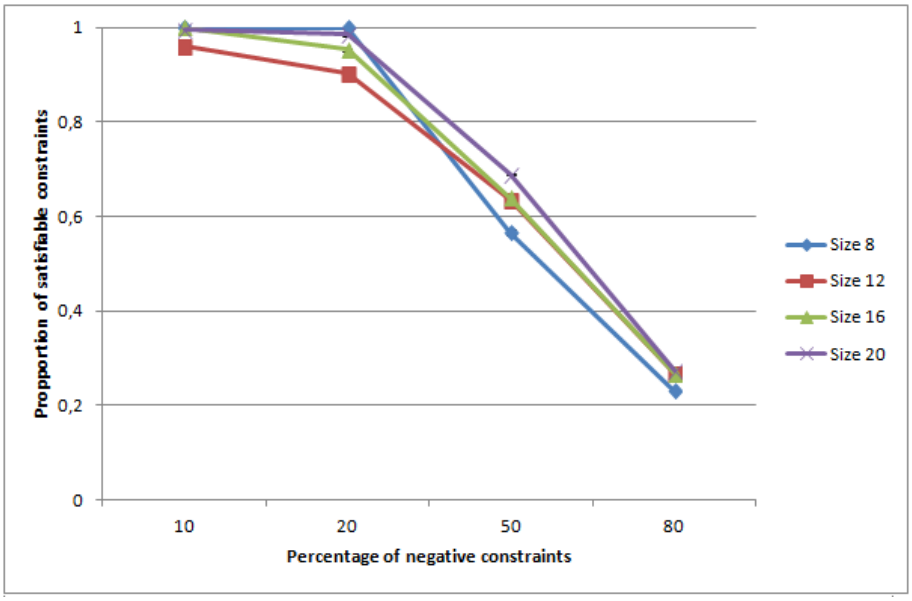
The results showed that size of the instances was not a factor in network performance, but that the number of negative constraints certainly was. Regardless of how the networks were trained (either on the same instances, random ones, with various kinds of C^- percentages, or with whatever size), they all tested better or at least equal on instances with a small percentage of negative constraints, compared to instances with larger percentages negative constraints.



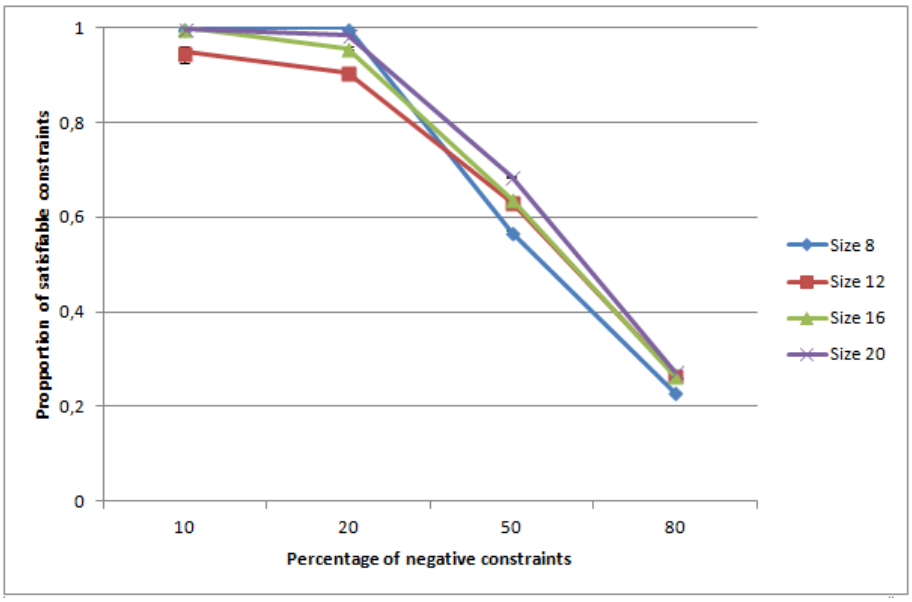
(a) Results of networks trained and tested on instances with the same parameter settings. The error bars show the maximum and minimum value of the three runs.



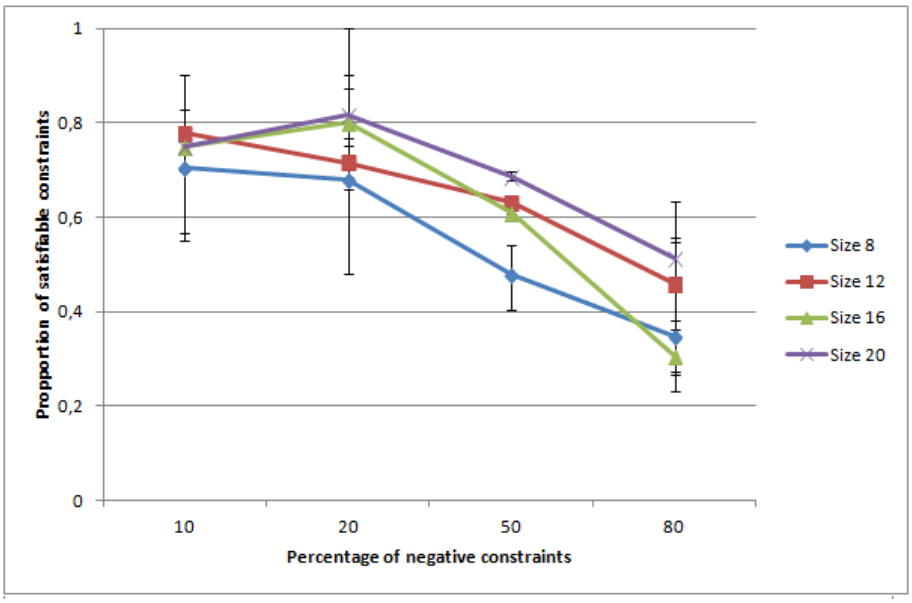
(b) Results of networks trained on instances with random percentages of negative constraints (7 train instances were used and each had a random percentage of negative constraints, being either 10, 20, 50 or 80 percent). The error bars show the maximum and minimum value of the three runs.



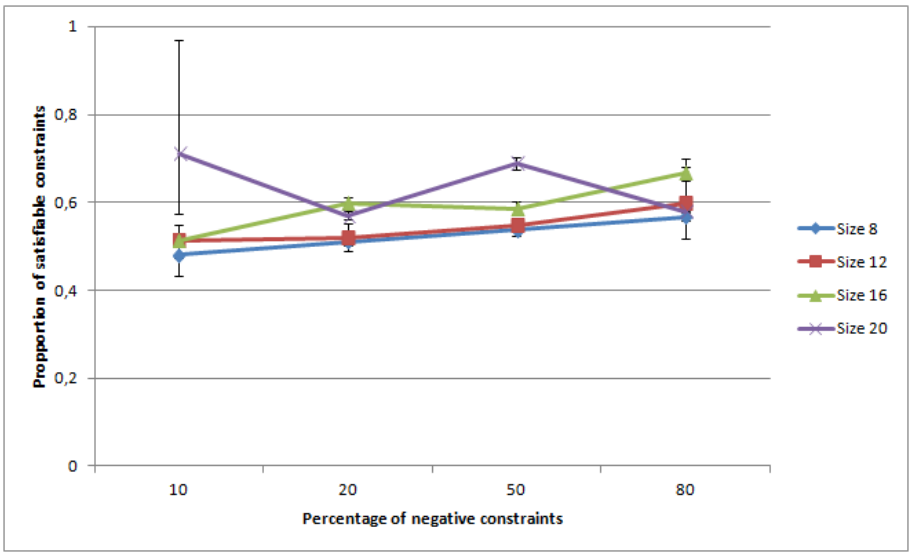
(c) Results of networks trained on instances with 10% negative constraints. The error bars show the maximum and minimum value of the three runs.



(d) Results of networks trained on instances with 20% negative constraints. The error bars show the maximum and minimum value of the three runs.



(e) Results of networks trained on instances with 50% negative constraints. The error bars show the maximum and minimum value of the three runs.

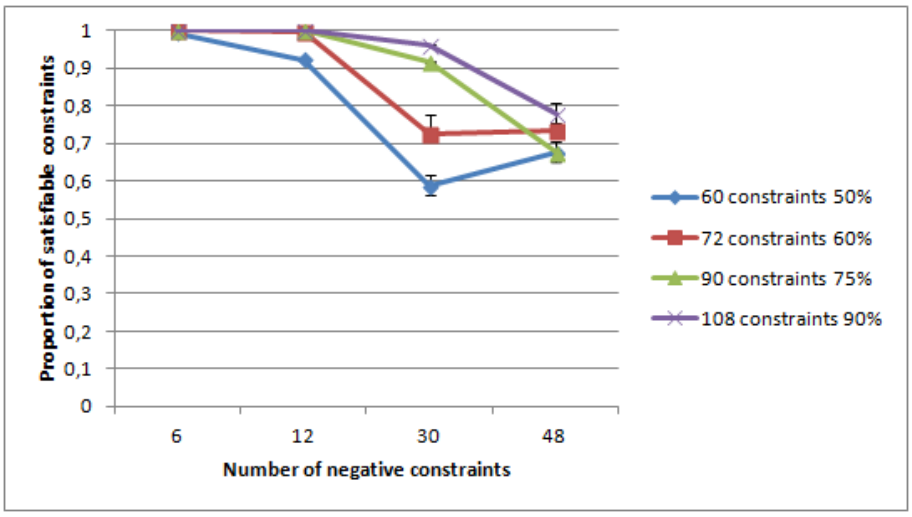


(f) Results of networks trained on instances with 80% negative constraints. The error bars show the maximum and minimum value of the three runs.

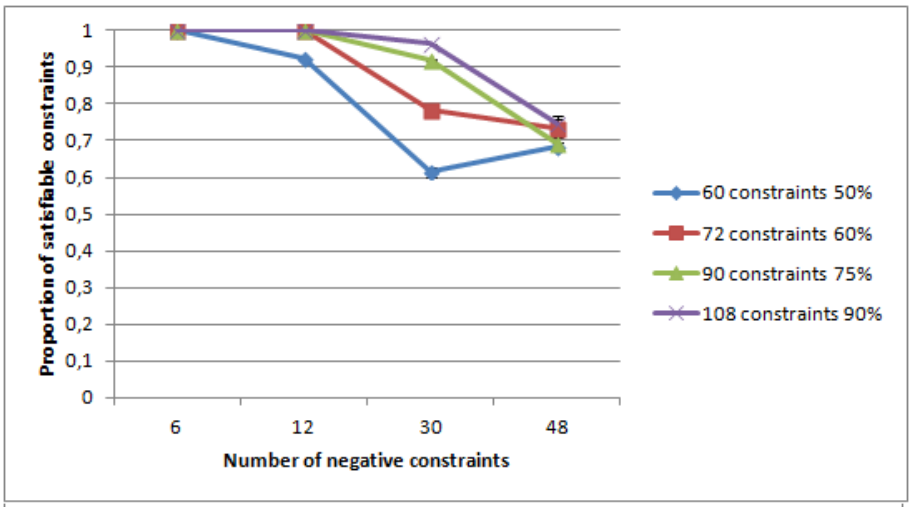
Figure 7: Simulation results for the influence of $|C^-|$, with various instance sizes. These networks were trained in various ways. After testing them, they all performed best on instances with fewer negative constraints. Note that there is almost no difference between solution qualities of differently sized instances.

We subsequently investigated whether or not the results of the previous simulations were purely due to $|C^-|$, or had another cause. By increasing $|C^-|$, one also decreases $|C^+|$. To test whether or not the effect was really due to an increase of $|C^-|$ and not actually a decrease of $|C^+|$, we increased the number of $|C^+|$ constraints in these simulations while keeping the number of $|C^-|$ constraints stable. We expected that the number of positive constraints in an instance does not effect network performance, regardless of instance size, or number of negative constraints. All these simulations were done with instances with 16 elements, 6, 12, 30 and 48 negative constraints and 50, 60, 75 and 90 percent of overall constraints (because increasing the overall percentage of constraints will increase the $|C^+|$ if $|C^-|$ is fixed). Once again, three runs were used. Figure 8 shows the results. Note that these results are the averages over 3 runs. Error bars display the maximum and minimum of the runs.

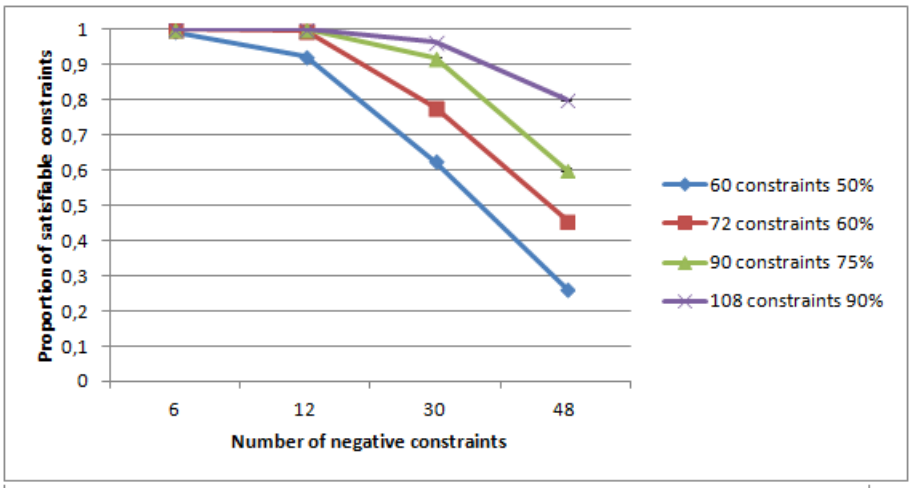
The results of these simulations showed that the previous results were not because of the fact that the number of positive constraints was decreased. In these simulations, the number of positive constraints in instances is increased, but the same effect of the previous simulations can still be seen. As a matter of fact, more overall constraints seems to improve solution quality. The effect of the $|C^-|$ parameter is thus relative rather than absolute: the lower the percentage of negative constraints, the better the performance.



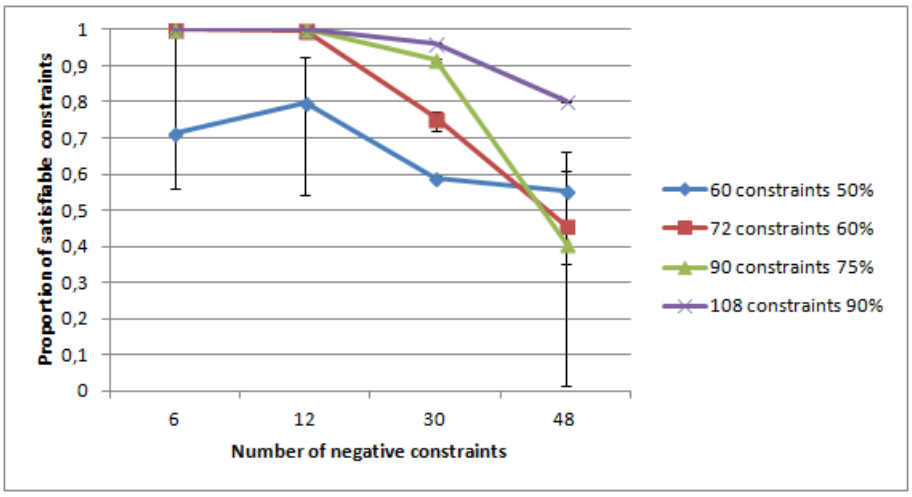
(a) Results of networks trained and tested on instances with the same parameter settings. The error bars show the maximum and minimum value of the three runs.



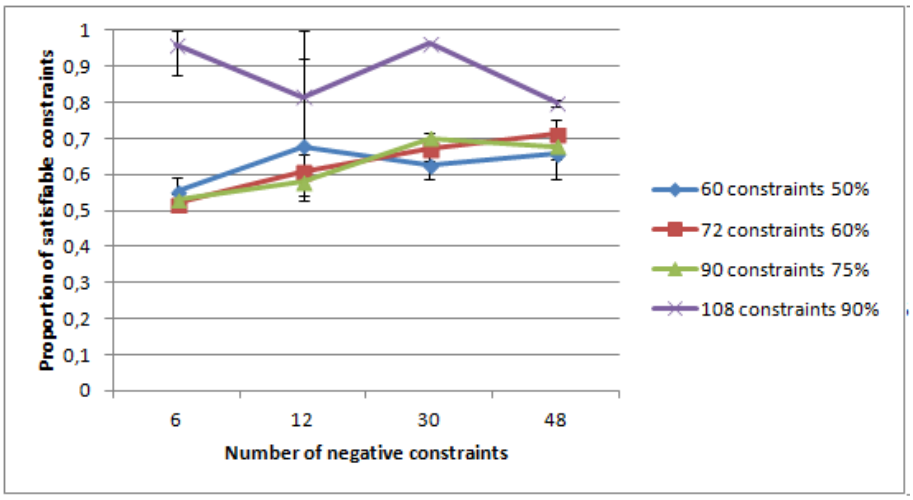
(b) Results of networks trained with instances with random percentages of negative constraints (7 train instances were used and each had a random amount of negative constraints, being either 6, 12, 30 or 48). The error bars show the maximum and minimum value of the three runs.



(c) Results of networks trained on instances with 10% negative constraints. The error bars show the maximum and minimum value of the three runs.



(d) Results of networks trained on instances with 50% negative constraints. The error bars show the maximum and minimum value of the three runs.



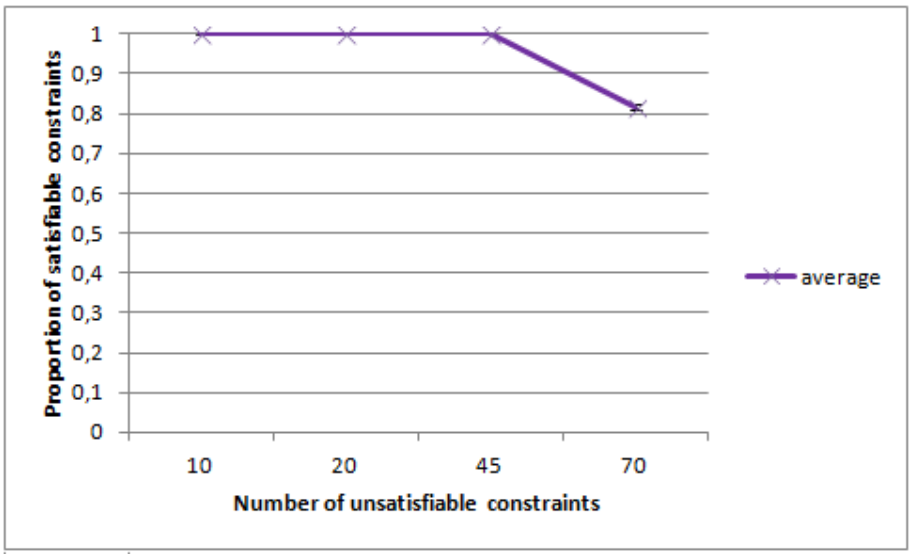
(e) Results of networks trained on instances with 80% negative constraints. The error bars show the maximum and minimum value of the three runs.

Figure 8: Simulation results for the influence of $|C^-|$, with various $|C^+|$. These networks were trained in various ways. After testing them, they mostly performed better on instances with fewer negative constraints and many overall constraints. The effect of the $|C^-|$ parameter is thus relative rather than absolute: the lower the percentage of negative constraints, the better the performance.

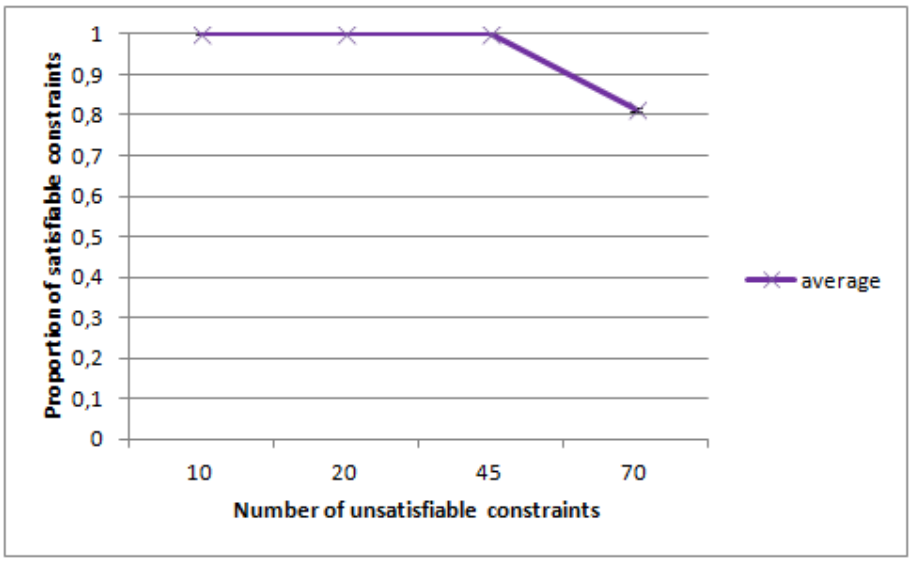
4.3 Q2: Simulations for the influence of parameter u

We also performed simulations for parameter u, to make sure that the effect seen with $|C^-|$ could be generalized to other parameters. Simulations for the u parameter were more tricky, because the instances used were randomly generated. This means that although one can create instances with few negative instances that still have a high u, they do not tend to pop up when one randomly generates instances. Because of this, the instances with high u also had a high $|C^-|$ and the other way around. The simulations were roughly similar to those for the influence of $|C^-|$, with instances varying now only in parameter u size. All had 16 elements. The number of unsatisfiable constraints was varied between 10, 20, 45 and 70. To achieve the higher number of unsatisfiable constraints, all instances were fully connected. The results are shown in Figure 9. Note that these results are the averages over 3 runs. Error bars display the maximum and minimum of the runs.

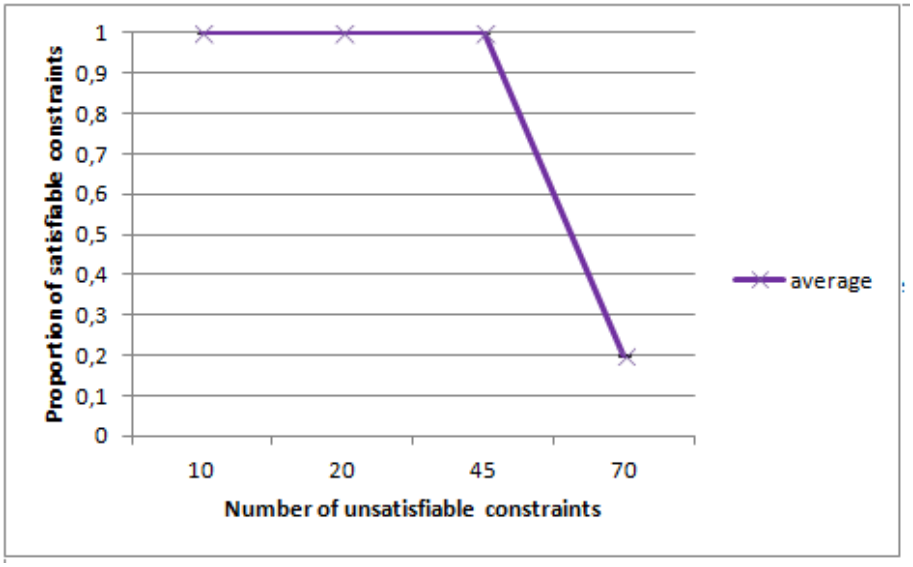
These results showed that in most cases, networks could better handle instances with smaller numbers of unsatisfiable constraints. Only for networks specifically trained on instances with high numbers of unsatisfiable constraints, shown in Figure 9f, was this effect not observable.



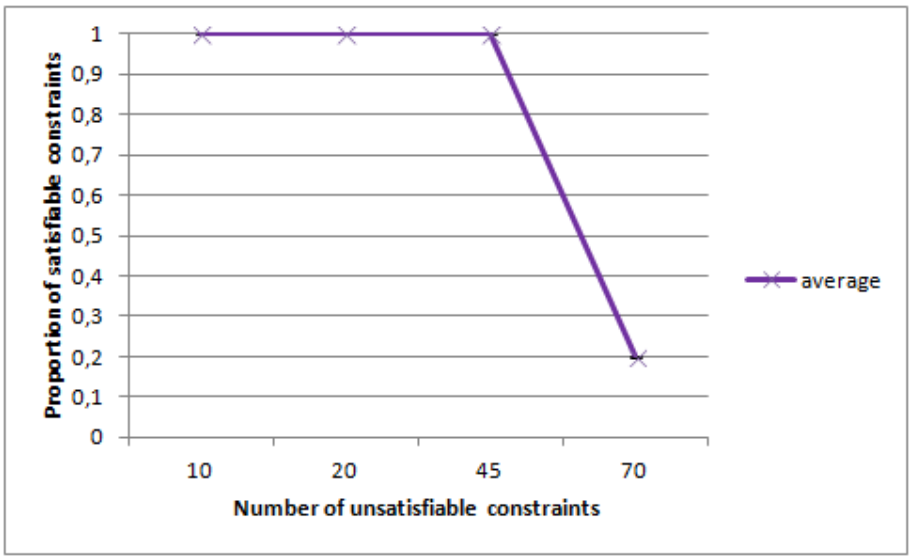
(a) Results of networks trained and tested on instances with the same parameter settings.



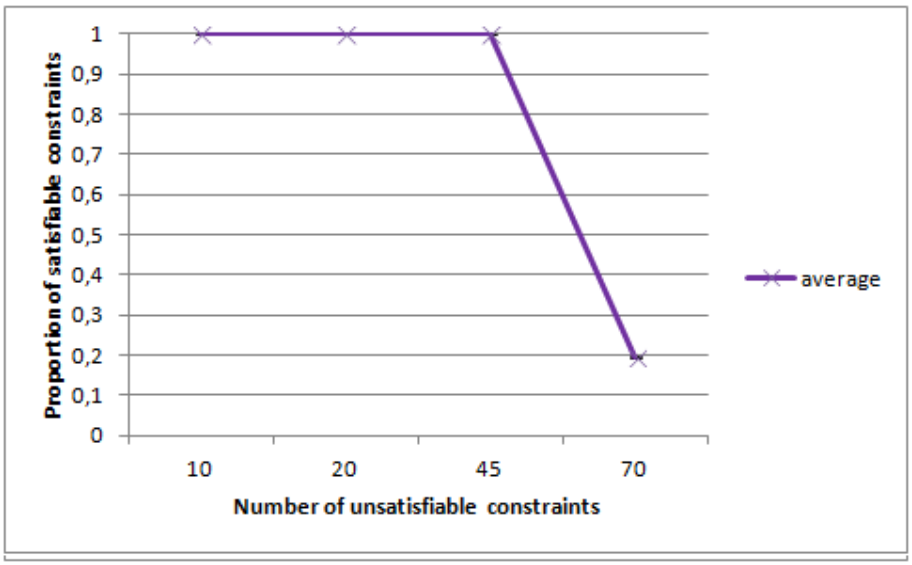
(b) Results of networks trained on instances with random amounts of unsatisfiable constraints.



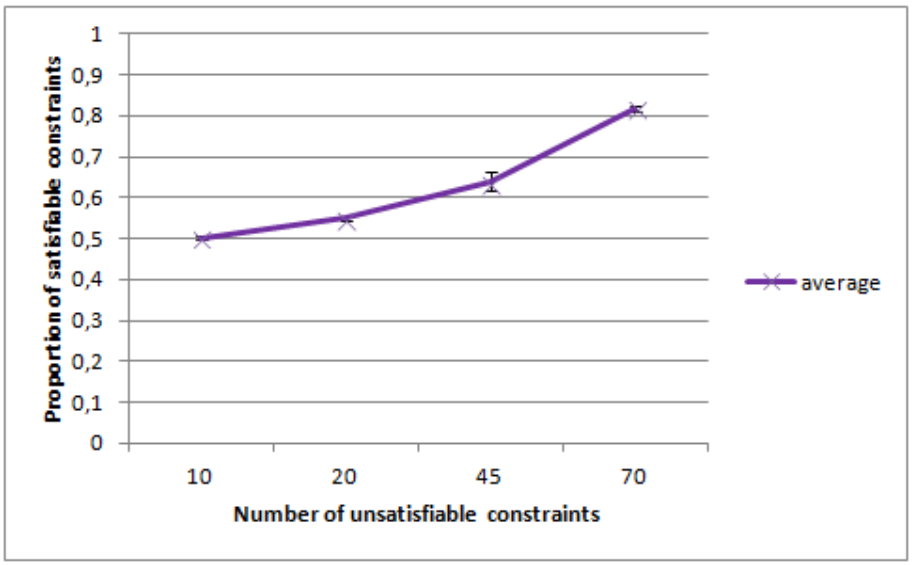
(c) Results of networks trained on instances with 10 unsatisfiable constraints.



(d) Results of networks trained on instances with 20 unsatisfiable constraints.



(e) Results of networks trained on instances with 45 unsatisfiable constraints.



(f) Results of networks trained on instances with 70 unsatisfiable constraints.

Figure 9: Simulation results for the influence of the u parameter. These networks were trained in various ways. After testing them, they all performed best on instances with fewer unsatisfiable constraints, except for the networks trained on instances with 70 unsatisfiable constraints. All of the instances had 16 elements and were fully connected, instead of the 50% possible constraints used in previous simulations.

5 Discussion

Models of cognitive capacities are often intractable, but for many, fp-algorithms exist [15]. Whether or not it is plausible that such algorithms can evolve was investigated in this thesis. We used the model Coherence, because it is widely applicable to many cognitive domains [25]. We used neural networks to evolve solutions to Coherence instances. The main goal was to investigate if the networks exploited parameters of the input for which fp-algorithms are known. In two research questions we looked at two specific parameters, the number of negative constraints ($|C^-|$) and the number of unsatisfiable constraints (u). Because neural networks do not produce clearly interpretable algorithms as output and we were more interested in solution quality, we decided to assess the networks on their parameter sensitive performance (PSP), rather than on their fixed-parameter tractability.

Our results are promising, as they suggest that evolved networks indeed perform better on instances for which the values of parameters for which Coherence is known to be fp-tractable are low. These results as well their implications (including future work) are discussed further below.

5.1 Q1: Simulations for the influence of parameter $|C^-|$

Our simulation results show that the influence of $|C^-|$ on solution quality was exactly as expected. In almost every simulation, networks perform better on instances with few negative constraints, regardless of instance size, network training regime or number of constraints. The results thus support the idea that evolved networks can exploit parameters for which fp-algorithms are known: the smaller the value of the parameter, the better the performance.

Only networks specifically trained on instances with many negative constraints had some trouble when tested on instances with few such constraints. It is not clear what instances agents might encounter in everyday life, which is why we trained the networks in so many different ways. However each of those still performed best or just as good on instances with few negative constraints. Important here is that once again the instance size could not grow beyond 20 elements. Perhaps instances with many more elements might make the networks behave differently, although it seems that instance size is not a big influence on the solution quality (as was shown in the previous simulations).

We furthermore tested if the effects that we observed were specific to $|C^-|$ or that other parameters might have an influence. With a reduction of the number of negative constraints, one also increases the number of positive constraints and visa versa. So by that logic, it might very well be that the networks showed parameter sensitive performance not relative to $|C^-|$, but to $|C^+|$. Our simulations showed that this is not the case. When the number of overall constraints is increased the performance seemed to increase, even as the number of negative constraints was constant. However, when the number of negative constraints was increased, performance dropped drastically. The surprising result was that the number of positive constraints still has some influence: more positive constraints meant in some cases a better performance. It showed that the effect of $|C^-|$ is relative rather than absolute: if the number of negative constraints is relatively low, the solution quality is better.

5.2 Q2: Simulations for the influence of parameter u

In the previous simulations it was shown that networks tested poorer on instances with many C^-, compared to those with few. In these simulations, we tested if this effect could be generalized to other parameters, by performing similar simulations for parameter u. This is the case, although the effect is less clear and for networks trained on instances with many unsatisfiable constraints,

it seems to disappear. There are several possible reasons for the lack of clarity in these results compared to the ones for $|C^-|$, which will be given in the following parts.

The biggest challenge with these simulations was finding appropriate training and test instances. These needed to be generated which was not easy because the $|C^-|$ and u correlate: many unsatisfiable constraints requires many negative constraints. Instances with few unsatisfiable constraints do not necessarily need to have few negative constraints, but when one generates them randomly, this tends to be the case. Because the two parameters seem to correlate with each other, it does not come as a surprise that the results also seem to overlap. Many unsatisfiable constraints in an instance decreased solution quality of the network, just as many negative constraints did.

In future work, the scope of these simulations could be expanded. It would be very interesting to generate instances that have many negative constraints and only few unsatisfiable ones. These exist in theory, but it will be a challenge to create a program that can generate such instances (it could be done by hand probably, but that is not something I will recommend to anyone). Only tests with these instances will be able to show if the u and $|C^-|$ should be seen as separate parameters that are both causes of parameter sensitive performance.

5.3 Other interesting results

In this section, two sets of results from the simulations to determine characteristics of the network are discussed. These are not of direct importance to the main questions, but are still interesting in their own right.

The first set of results examined network computation time. As expected with a fixed number of generations, evolution times did not vary. However, one might expect that training of generations might take longer for certain instances. Since we could only use instances that had at most 20 elements, networks could always be trained in roughly the same time. As the number elements rapidly increases, this might change. It would be interesting future work to test whether or not HyperNEAT can always evolve networks fast, regardless of the sizes of instances used.

The second set of simulations examined the effect of recurrency for the evolved networks. The results of the recurrency simulations show that more recurrent cycles does not improve solution quality. As a matter of fact, more recurrent cycles is a bad thing, because it does increase runtime. The networks were trained and tested on instances with a high number of negative constraints (80%) and many elements (16), because under these conditions, recurrency could improve the solution quality the most. As can be seen in Figure 6, no improvement could be found by increasing the number of recurrent cycles. The reason for this is not clear, because in general, recurrency seems to be necessary to solve intractable problems like Coherence.

A possible reason for this lack of improvement could be the size of the instances. Even with 16 elements, the instances are not that big. It could be that for solving such smaller instances, no recurrency is needed and that the networks find the best possible solution regardless. However, no optimal solution is found for very instance, so apparently these smaller instances are not necessarily easy to solve. A different reason could be that the way Coherence was represented allowed for very little flexibility in the networks. This is supported by the fact that the standard settings with respect to parameters like population size or mutation rates barely effected network performance. The reason for this is unclear.

For future research, it would be very interesting to redo such tests as described in this project with much larger instances (with 50 up to 100 elements). In such situations parameter settings including population size or number of recurrent cycles could become much more important. Very powerful machines are needed to generate such large instances and they were simply not available at the time for this research.

5.4 Future work

The promising results of this thesis generate new questions and options for future work and projects. The results show that networks evolved for a Coherence instance show parameter sensitive performance towards certain parameters that are known to be fp-tractable. Additional simulations as proposed in the previous section are needed to clarify the effect of the u parameter, especially in relationship with $|C^-|$. If it can be shown that these parameters give the network parameter senstive performance seperatly from each other, that would be an interesting finding. Other cognitively relevant problems could be used in similar tests to further investigate the evolution of networks and their parameter sensitive performance (PSP). Especially problems where it is possible to get very large instances should be interesting. It is unfortunate that in this project only instances of sizes up to 20 elements could be used, because larger instances might break trends visible in these results, or might provide further proof for the PSP of the networks. Larger instances could be generated using more powerful computers or several computers linked to one another. It seems unlikely that instances of sizes up to 50 could be generated, but perhaps this is possible if computing power keeps increasing.

One of the big contributions of this project goes beyond the evolution of the networks. We have introduced a new term to assess the complexity of an algorithm: parameter sensitive performance (PSP). The results show its use in this project: the networks did not increase in computation time with the increase of certain parameters, but rather decreased in solution quality. Similar projects where the time for an algorithm is locked can have benefits from assessing performance based on solution quality. It is not always the case that an algorithm just has unlimited time to find solutions. Where the optimal solution is not needed, it is far better for a reseacher to give the algorithm a fixed time. Based on instance properties, it will then find solutions with varying quality, rather than always finding the optimal solution in what could possibly be a very long time period. Humans also work with restricted time when solving problems. We do not need many hours to solve most cognitive tasks. Rather, we will find solution that are suboptimal and are just the best we can come up with in a limited time period. Therefore, PSP can best be used to assess models of cognitive functions, because in most cases the goal is not to find optimal solutions, but good enough solutions in limited time. Regardless of whether similar projects like these will be performed with other cognitive models, other instances, other network or evolutionary techniques, the produced algorithms can all be assessed using PSP.

5.4.1 Future work in artificial intelligence

Besides the results that are relevant for cognitive science, the results here are also relevant for artificial intelligence (AI). Although the methods used here are specifically applied to cognitive models, they can just as easily be applied to other computational problems. The approach used here can get viable results, even when natural evolution is not important. This could result in a much more efficient method of finding fp-algorithms or PSP algorithms. Right now, possible parameters have to be found by the researcher himself and then tested for their validity. With techniques similar to those used in this project, the neural networks would be able to find and exploit the parameters themselves. By feeding all sorts of problem instances to the network, instances that have low values of certain parameters will be solved with better solution quality. By looking for patterns between such instances one can find the parameter that is restricted. This process can save substantial effort for researchers that want to find the sources of complexity in computational problems.

6 Conclusion

Evolving artificial neural networks and looking at their performance is not new, as can for example be seen in the many applications of NEAT. However, we were particularly interested in the adaptation of these networks to exploit certain parameters in terms of changes in running time and solution quality as parameters vary in size. Therefore, high performance of the network is not really of interest, while that is in most other researches. We expected poor performance of networks on instances with high values of parameters of which fp-algorithms are known, which we called parameter sensitive performance (PSP). The results indeed showed that networks perform better on instances where these parameters were small in size. This project thus proved that it is plausible that algorithms evolve that exploit parameters of the input. For cognitive science, this means that the many intractable models of cognitive capacities (such as Coherence) can still have algorithmic level implementations that are tractable and plausibly used by cognitive agents.

References

[1] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M Protasi. *Complexity and approximation: Combinatorial optimization problems and their approximability properties*. Springer, 1999.

[2] Mark Blokpoel, Johan Kwisthout, Theo P van der Weide, Todd Wareham, and Iris van Rooij. A computational-level explanation of the speed of goal inference. *Journal of Mathematical Psychology*, 57(3):117–133, 2013.

[3] Dmitri B Chklovskii and Alexei A Koulakov. Maps in the brain: What can we learn from them? *Annual Review of Neuroscience*, 27:369–392, 2004.

[4] Rodney G Downey, Michael R Fellows, and Ulrike Stege. Parameterized complexity: A framework for systematically confronting computational intractability. In *Contemporary Trends in Discrete Mathematics: From DIMACS and DIMATIA to the Future*, volume 49, pages 49–99. AMS-DIMACS Proceedings Series, 1999.

[5] Aguston E Eiben and Marc Schoenauer. Evolutionary computing. *Information Processing Letters*, 82(1):1–6, 2002.

[6] David B Fogel. *Evolutionary computation: toward a new philosophy of machine intelligence*, volume 1. John Wiley & Sons, 2006.

[7] Lance Fortnow. The status of the P versus NP problem. *Communications of the ACM*, 52(9):78–86, 2009.

[8] Michael R Garey and David S Johnson. *Computers and intractability*. WH Freeman and Company, San Fransisco, 1979.

[9] Marco Gori and Alberto Tesi. On the problem of local minima in backpropagation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 14(1):76–86, 1992.

[10] Alex Graves, Marcus Liwicki, Santiago Fernández, Roman Bertolami, Horst Bunke, and Jürgen Schmidhuber. A novel connectionist system for unconstrained handwriting recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 31(5):855–868, 2009.

[11] Tijl Grootswagers. *Having your cake and eating it too: Towards a fast and optimal method for analogy derivation*. Master's thesis, Radboud University Nijmegen, 2013.

[12] David Marr. *Vision: A computational investigation into the human representation and processing of visual information*. WH Freeman and Company, San Francisco, 1982.

[13] M Minsky and S Papert. *Perceptrons*. MIT Press, Cambridge, MA, 1969.

[14] Moritz Müller, Iris van Rooij, and Todd Wareham. Similarity as tractable transformation. In *Proceedings of the 31st Annual Conference of the Cognitive Science Society*, pages 50–55, 2009.

[15] Rolf Niedermeier. *Invitation to fixed-parameter algorithms*, volume 3. Oxford University Press Oxford, 2006.

[16] Riccardo Poli, Leonardo Vanneschi, William B Langdon, and Nicholas Freitag McPhee. Theoretical results in genetic programming: the next ten years? *Genetic Programming and Evolvable Machines*, 11(3-4):285–320, 2010.

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

[18] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. *Learning representations by back-propagating errors*. MIT Press, Cambridge, MA, USA, 1988.

[19] Hava T Siegelmann and Eduardo D Sontag. On the computational power of neural nets. *Journal of Computer and System Sciences*, 50(1):132–150, 1995.

[20] Jiří Šíma and Pekka Orponen. General-purpose computation with neural networks: A survey of complexity theoretic results. *Neural Computation*, 15(12):2727–2778, 2003.

[21] Olaf Sporns. Network analysis, complexity, and brain function. *Complexity*, 8(1):56–60, 2002.

[22] Kenneth O Stanley. Compositional pattern producing networks: A novel abstraction of development. *Genetic Programming and Evolvable Machines*, 8(2):131–162, 2007.

[23] Kenneth O Stanley, David B D'Ambrosio, and Jason Gauci. A hypercube-based encoding for evolving large-scale neural networks. *Artificial Life*, 15(2):185–212, 2009.

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

[25] Paul Thagard. *Coherence in thought and action*. The MIT Press, 2000.

[26] Paul Thagard and Karsten Verbeurgt. Coherence as constraint satisfaction. *Cognitive Science*, 22(1):1–24, 1998.

[27] John K Tsotsos. Analyzing vision at the complexity level. *Behavioral and brain sciences*, 13(3):423–469, 1990.

[28] Stefan van der Meer, Iris van Rooij, and IG Sprinkhuizen-Kuyper. Evolving fixed-parameter tractable algorithms. In *BNAIC 2008 Belgian-Dutch Conference on Artificial Intelligence*, pages 153–161, 2008.

[29] Iris van Rooij. *Tractable Cognition: Complexity Theory in Cognitive Psychology*. PhD thesis, University of Victoria, 2003.

[30] Iris van Rooij. The tractable cognition thesis. *Cognitive Science*, 32(6):939–984, 2008.

[31] Iris van Rooij, Patricia Evans, Moritz Müller, Jason Gedge, and Todd Wareham. Identifying sources of intractability in cognitive models: An illustration using analogical structure mapping. In *Proceedings of the 31st Annual Conference of the Cognitive Science Society*, pages 915–920, 2008.

[32] Iris van Rooij, Cory D Wright, and Todd Wareham. Intractability and the use of heuristics in psychological explanations. *Synthese*, 187(2):471–487, 2012.

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

[34] B Yegnanarayana. *Artificial neural networks*. PHI Learning Pvt. Ltd., 2009.

Appendices

A Coherence fp-algorithm for $|C^-|$

In this section we give an overview of the fp-algorithm that was implemented in an early stage of the thesis. The algorithm can be used to generate larger Coherence instances and their corresponding solutions. However, the algorithm is severely limited in the amount of negative constraints that can be present in the instance, rendering it useless for this project. Readers interested in the code of this algorithm can contact the author of this thesis.

The algorithm contains several reduction rules that transform the Coherence instance to an instance of Min-cut, a problem that is in P. In the first rule, every element linked to a negative constraint is set either true or false. In the next rule, every constraint linking nodes that are now already set is removed. In the third rule, all true assigned nodes are merged and all false assigned nodes are merged and their constraints linking them to unassigned nodes are merged where needed. The resulting graph can be solved as a Min-cut instance (the steps for this algorithm are given in very simple pseudo-code in Algorithm 1. Note that in these steps, the first one can take a lot of time if there are many negative constraints. However, if this number stays low, the first step and the following steps do not take a lot of time. The process is illustrated in Figure 10.

Algorithm 2 The $|C^-|$ fp-algorithm for Coherence

Load Coherence instance
AC1 generates several constraintmatrices, each with different elements set
AC2 removes certain constraints
AC3 merges set elements
for all matrix $\in$ constraintmatrices **do**
  mincut
  satisfiedconstraints = satisfied + constraints - cutsize
end for
solution = MAX(satisfiedconstraints)
return *solution*

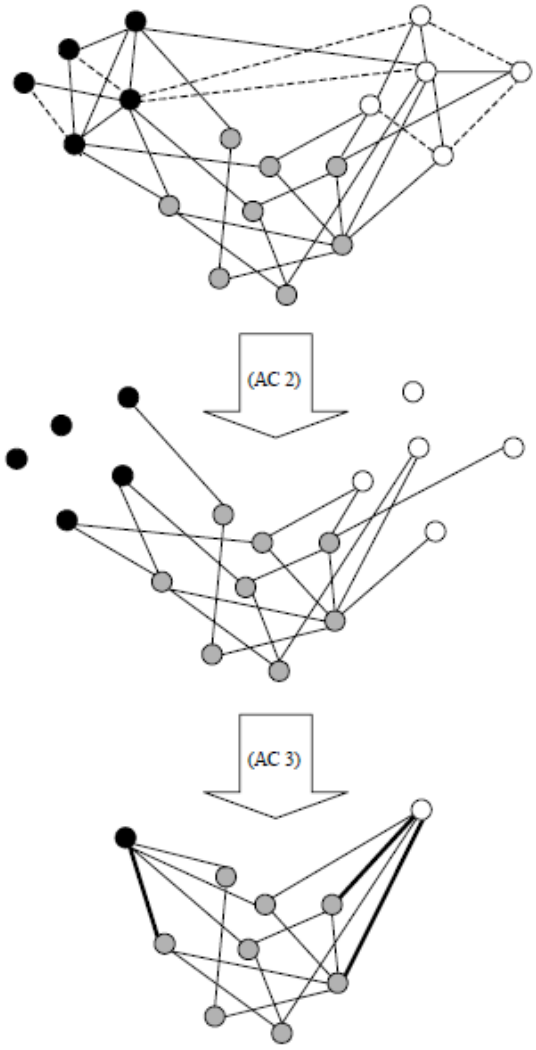


Figure 10: Reduction steps of Coherence fp-algorithm. The first step has set all elements linked to a negative constraint either false (black) or true (white). The second step (AC2) removes all constraints linked to already set elements. The third step (AC3) merges all the assigned elements, leaving only two assigned elements and the unassigned ones (the grey dots). The thick black line indicates that the weights of those constraints have changed. Figure taken from [29].