Adaptive Sampling

On the Similarities between Learning and Evolving

Michiel B. de Jong
Leiden Institute of Advanced Computer Science
Leiden University, P.O. BOX 9512
2300 RA Leiden, The Netherlands
E-mail: mdejong@wi.leidenuniv.nl

August 24, 1998

Master's Thesis — Supervisors:
Dr. A.E. Eiben,
Dr. W.A. Kosters.

Abstract

A new framework is presented, that was designed to be a unification of two existing biologically inspired paradigms: Evolutionary Computation (EC) and Neural Computation (NC). In this thesis, the central goal is to untangle bio-inspired methods from their biological metaphors, so that issues like exploration, generalization, structural credit assignment, recombination, etcetera, can be studied in a more general language, instead of being studied separately in different metaphors. It is hypothesized that mimicing nature's principles, and not its epiphenomena, leads to better algorithms. The so-called Adaptive Sampling framework is used for analyzing the 3-SAT problem, which led to one neural method and five mixed methods, that mix elements of EC and of NC in different ways. These methods have been tested against the best currently known incomplete 3-SAT algorithm, the SAW-ing EA. Perhaps contradicting conventional intuition, the Neural method needs less evaluations to reach a higher success rate than the SAW-ing EA, but its floating point representation consumes much more time. However, one of the mixed approaches, called Lamarkian SEA-SAW, outperforms the SAW-ing EA in success rate, in number of evaluations and in runtime, on all test sets. This suggest that it is indeed beneficial to study EC and NC in the unifying Adaptive Sampling framework.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Subject

It is widely realized that genetic evolution and neural learning have more in common than is currently exploited in biocomputation [LON97]. This thesis investigates their similarities.

For the maturation of bio-inspired computation, there is a need for a general theory, that focuses on the principles and not on the epiphenomena, and puts the many individual experiments that have been performed through the years in one common context.

Instead of trying to find proofs of convergence or capacity, which has been tried, but has turned out to be very difficult, it might be useful to investigate the *common principles* of the studied methods. This would make them instantiations of a common superclass, and would create a common context for the different experiments.

The existence of common principles that are fundamental to the power behind both Neural Computation (NC) and Evolutionary Computation (EC), forms the central hypothesis of this thesis:

CENTRAL HYPOTHESIS
Although Evolutionary and Neural Computation come from different origins and are used for solving different problems, the principles that make them work are the same, and the remaining differences can be explained as problem dependent parameters. Concentrating on these common principles, and not on epiphenomenal features of either one, facilitates the design of better algorithms.

1.2 Goal

The proof for the hypothesis will not be a logical deduction; there are too many concepts involved that are defined only in intuitive terms. It is especially hard to give a good definition of the difference between principles and problem-specific parameters.

However, it is possible to give a specification of a set of common features. These features then define a class of methods, that can be described in a framework. The goal of this thesis is to prove the hypothesis by developing such a framework. The methods used in Evolutionary Computation (EC) and Neural Computation (NC) should be definable within this framework:

INCLUSION REQUIREMENT
The framework should define a class of methods, including at least Evolutionary Algorithms and Neural Networks.

Of course, any existing general purpose programming language (for instance C++) would in principle satisfy this goal, because NNs and EAs can both be implemented in C++. This is clearly not what is wanted, because any algorithm can be implemented in a general purpose programming language. This rules out C++as a candidate:

STRICTNESS REQUIREMENT
The framework should define a relatively small family of algorithms, to assure that the common principles found are not trivial properties of just any algorithm.

Another family that contains both NNs and EAs is the family that contains nothing else. This would neither be a very interesting family, because it does not *compare* NNs and EAs in any way. The parameters of the framework would not be used to tune for specific problems, but would work as a switch between the two subclasses.

LARGENESS REQUIREMENT
On the other hand, the framework should define a large enough family of algorithms, to assure that EC and NC are not simply represented separately, with the parameters being only a switch between the two.

The framework should be like a greatest common denominator. It should define the largest set of features that are present in both, i.e. all the similarities between EC and NC, and not the differences.

The hypothesis can be tested by trying to define a framework satisfying these three requirements. This involves an intuitional judgement of whether the similarities found are really the principles that make the methods work, and not just some trivial properties. Also, it should be judged whether the parameters of the framework are problem dependent, and do not represent the actual choice between two separately defined classes. Both these measures will be evaluated in the Conclusions, in Section 8.1.

The second part of the hypothesis will be investigated with two additional goals:

1. Exploring what new methods are suggested by the framework:
   If the framework takes away the epiphenomena from the description of bio-inspired methods, then this new, more fundamental description should open opportunities for applying the fundamental principles to new problem domains, by instantiating the framework in a way that is different from the two specific instantiations we find in nature.

2. Simulating different instantiations of the framework:
   Wherever possible, empirical results should support a theory. The 3-SAT

problem will test the usefulness of the framework for supporting the design of better algorithms.

It is *not* the purpose to invent a new class of algorithms, but to supply one model to support the design of methods within and in between the two classes, thus showing how closely related these are. The purpose of this thesis is closer to that of [Far90], which is a paper concerning connectionism, but is an example of a unifying framework, comparing terms used in different disciplines for the same concepts. Other work with a similar goal includes that of Dorigo et al. [DB94], who compared Q-learning to classifier systems, Eiben et al. [EAvHN95], who formulated a general procedure for search methods, and Radcliffe and Surry [Rad91], who supplied a mathematical tool ("formae") for comparing operators and assumptions of different Evolutionary Algorithms. All these studies investigate similarities between different methods.

1.3 Structure of this Thesis

The 3-SAT problem forms a *running example* to illuminate concepts in this thesis, wherever possible. In the following chapter, the Adaptive Sampling framework is introduced. Chapters 3 and 4 discuss Evolutionary and Neural Computation.

The central Chapters 5 and 6 then investigate Adaptive Sampling in more depth, dividing issues into Sampling and Adaptation. Chapter 7 discusses the experiments that have been performed to investigate to what extent the use of the Adaptive Sampling framework can increase performance, and is followed by the conclusions.

1.4 3-SAT: The Running Example

At the beginning of next chapter, an intuitive notion of adaptivity and sampling is developed. To give these intuitive concepts some context, we first introduce the running example of this thesis: 3-SAT.

The Adaptive Sampling framework has been tested by applying it to the 3-SAT problem class. An instance of the 3-SAT problem is uniquely defined by a Boolean expression, and any valuation that satisfies that expression is a solution of that instance. The expression has to be in conjunctive normal form (also known as a "product of sums"), in which each of the k clauses contains exactly three literals:

$$(p_{11} \vee p_{12} \vee p_{13}) \wedge (p_{21} \vee p_{22} \vee p_{23}) \wedge \ldots \wedge (p_{k1} \vee p_{k2} \vee p_{k3}).$$

In this expression, each literal p_{ij} is either one of the m free variables of the expression ($\{x_1, x_2, \ldots, x_m\}$) or the negation of one of those ($\{\overline{x_1}, \overline{x_2}, \ldots, \overline{x_m}\}$). For every i, a clause contains at most one of x_i and $\overline{x_i}$. A solution is a valuation $Val : \{x_1, x_2, \ldots, x_m\} \rightarrow \{\mathit{true}, \mathit{false}\}$, such that the expression yields *true* for this valuation. An example of an instance of the 3-SAT problem, with $k = 5$ clauses and $m = 4$ free variables, is:

$$(x_1 \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_4}) \wedge (x_1 \vee \overline{x_3} \vee x_4) \wedge$$
$$(\overline{x_1} \vee x_2 \vee x_3) \wedge (\overline{x_2} \vee x_3 \vee \overline{x_4}).$$

This problem instance has seven solutions (Val(x_1), Val(x_2), Val(x_3), Val(x_4)), for instance $(\mathit{true}, \mathit{true}, \mathit{true}, \mathit{false})$. The corresponding decision problem ("Does a given expression admit a solution?") is one of the most extensively studied NP-complete problems.

Several heuristics have been developed for it, the most prominent one being WG-SAT. It was introduced by Frank (see [Fra96, BEV96]) and is a variant of G-SAT, developed by Gent and Walsh (see [GW95, BEV96]).

It is possible to solve 3-SAT analytically, for instance using distributivity of conjunction over disjunction:

$$\mathrm{Solve}((a \vee b \vee c) \wedge (\mathit{Rest}\ldots)) :=$$
$$\mathrm{Solve}(a \wedge \mathit{Rest}\ldots)\, \mathit{or} (\mathrm{Solve}(b \wedge \mathit{Rest}\ldots)\ \mathit{or}\ \mathrm{Solve}(c \wedge \mathit{Rest}\ldots))$$

This is the way `ProLog` would solve 3-SAT. However, even if solving a 20-clause formula this way takes at most 1 millisecond, then solving a 25-clause formula would already take 243 milliseconds in the worst case, and a 70-clause formula can already take up to a few million millenia (although a lot of benefit for the average case might be drawn from backtracking, Most-Constrained-First search and other speed-ups).

Chapter 2

Adaptive Sampling

In this chapter, we will discuss a class of algorithms, some of which can solve 70-clause and even 100-clause 3-SAT problems within one minute. The basic principle behind these algorithms is to *sample* isolated features from the problem and from candidate solutions, that are *adapt*ed iteratively. Evolutionary and Neural Computation are the two main subfields of this class. Methods obtained by hybridization or cross-fertilization between the two, form the third main subfield.

This class of algorithms has been dubbed Adaptive Sampling, and it is described by the Adaptive Sampling framework. The basic definitions of the framework, namely a general problem format and a general procedure, will be introduced.

2.1 Defining what Adaptive Sampling is

Defining the difference between adaptive sampling (e.g., neural network methods, evolutionary algorithms, stochastic approximation) and conventional computation (e.g., complete algorithms, rule-based Artificial Intelligence, exhaustive search) is not a trivial matter. For instance, the fact that adaptive methods are adaptive, that the next step in the algorithm is determined by the information gathered so far, is not discriminative, because this holds for any Turing Machine. The fact that they mostly have a stochastic component, is more discriminative, because conventional methods are usually deterministic. But it is not essential: if the random seed is constrained to a specific value in the definition of the method, the method is deterministic (each run is predictable), but would still be called adaptive, because the underlying theoretical principles would be the same.

2.1.1 Iterative Improvement

What *does* seem an important intuitional difference is that the state of an adaptive method in progress, during the run, always contains a candidate solution, and that this candidate solution is improved iteratively.[1]

[1] However, this also holds for some methods that are not biologically inspired, for instance G-SAT [GW95, BEV96].

2.1.2 Sampling

Another characteristic is that this improvement is achieved by sampling. This sampling can be determining the output of a fitness function, that is given in the definition of the problem instance for a specific input, or considering one element of a training set. For 3-SAT, for instance, the `ProLog`-approach calculates a *tree* of features of a problem instance. Sampling methods for 3-SAT would sample only the leafs of such a tree, without traversing its branch-structure. The difference is in the fact that sampling does not require understanding the actual *structure* of the problem class or the problem instance, as long as its specific isolated elements are accessible.

2.1.3 Generalization

A third characteristic is that the information from a sample is combined with that from others, so that (hopefully) the more general information is conserved, while the other information (considered noise) is thrown away. In these vague terms, that is what might be considered as the essential principle of adaptive computation: a lot of specific information is gathered, and this long stream of data is iteratively compressed into the much smaller representation of candidate solutions, which would then hopefully convey the global information present in this data, while the noise cancels out in the process of compression. For EC, this stream of information consists of fitness evaluations that are to be improved; for NC, it contains the data that is to be learned. The useful thing about this approach is that the user does not need to specify exactly where this global information can be found, and what information has to be considered noise; the global information that is looked for is simply defined as “that information that is not canceled out when samples are combined”.

2.2 The Framework

In the rest of this chapter, the framework is described. The description contains a general problem format and a general procedure format. The procedure can in principle be used for the implementation of a simulation environment in which an algorithm can solve a problem. Because practice shows that simulation environments are hardly ever useful (most simulations can be programmed in a general purpose language within 500 lines), this possibility has not been investigated further within this project. The Adaptive Sampling framework was developed as a unification of two *paradigms*, not as a generalized *environment*. It should be used for designing algorithms, not for programming them.

However, a computational paradigm can best be explained in terms of computation, which is why the framework is presented here in terms of pseudo-Pascal.

2.3 General Problem Format

A real world problem should first be modeled before it can be represented in a computer. In this case, we assume that some computable error function exists that can be used for evaluating candidate solutions. This function is called the *sampling* function Φ. The term sampling refers to the fact that in the Adaptive Sampling paradigm, a problem is solved by generalizing from a set of specific *samples* from some distribution. In Evolutionary Computation, this is the distribution of "fitness" in the space of candidates. In Neural Computation, this is the distribution of the "data set" in the space of possible data set elements (more on this in Chapter 5).

In [CLR90], a concrete problem is defined as a relation on concrete problem instances and corresponding concrete solutions. Here, we take the problem instance to be represented as a set of k *objectives*, each of which is a vector in $\mathbb{R}^\ell$. If suitable, the number of objectives k can be taken to be one. This is merely a matter of how one wants to encode the problem.

A candidate solution (candidate, for short) is represented as a vector in $\mathbb{R}^m$. The evaluation calculated by Φ is a scalar in $\mathbb{R}$, and is an error value concerning one candidate and one objective. So Φ is a function from $\mathbb{R}^m \times \mathbb{R}^\ell$ to $\mathbb{R}$. The goal is to find a candidate for which the sum of errors on objectives is minimal. In Chapter 5, which discusses different types of problems, this goal is formulated precisely. For now, we can define the relationship between the sampling function Φ and the goal of the problem as follows:

DEFINITION 1 Given a sampling function Φ from $\mathbb{R}^m \times \mathbb{R}^\ell$ to $\mathbb{R}$ for some ℓ and m, and a set of k objectives $\mathcal{I} = \{o_1, o_2, \ldots, o_k\}$, in which each $o_i \in \mathbb{R}^\ell (1 \leq i \leq k)$, and $\varepsilon > 0$, an ε-solution to the problem defined by Φ and $\mathcal{I}$, is a candidate $c \in \mathbb{R}^m$ such that

$$\forall d \in \mathbb{R}^m : Error(c) \leq Error(d) + \varepsilon,$$

$$\text{in which for } x \in \mathbb{R}^m,\ Error(x) = \sum_{1 \leq i \leq k} \Phi(x, o_i).$$

2.3.1 3-SAT in the general problem format

In the general problem format, a *problem class* is represented by the sampling function Φ, and *problem instances* are represented by a set of objectives. In this section, we will see how this format can be used for representing 3-SAT.

Instances of the 3-SAT problem class are uniquely defined by a logical expression in three conjunctive normal form (3-CNF). To represent such an expression in a set of objectives, we choose to encode each clause as one objective. We take k to be the number of clauses, and ℓ and m both to be the number of variables occurring in the expression.

For encoding the clauses, we use the following key:

$$\text{EncodeClause}(p_1 \vee p_2 \vee p_3) = v \text{ such that for } 1 \leq i \leq \ell :$$

$$v_i = \begin{cases} 1 & \text{if } \exists j \in \{1,2,3\} : p_j = x_i, \\ -1 & \text{if } \exists j \in \{1,2,3\} : p_j = \overline{x_i}, \\ 0 & \text{otherwise.} \end{cases}$$

So for instance, the expression given in Section 1.1, would be represented with the vectors

$$(1,1,1,0,0), (-1,-1,0,-1,0), (1,0,-1,1,0),$$

$$(-1, 1, 1, 0, 0), \text{ and } (0, -1, 1, -1, 0).$$

We also need to define how we encode candidate solutions. For this, we will use a similar encoding. A valuation $\mathrm{Val} : \{x_1, x_2, \ldots, x_m\} \rightarrow \{\mathit{true}, \mathit{false}\}$ of the free variables x_1 to x_m is encoded as a vector w in $\mathbb{R}^m$, as follows:

$$\text{EncodeValuation}(\text{Val}) = w \text{ such that for } (1 \leq i \leq m) :$$
$$w_i = \begin{cases} 1 & \text{if } \mathrm{Val}(x_i) = \mathit{true}, \\ -1 & \text{if } \mathrm{Val}(x_i) = \mathit{false}. \end{cases}$$

Now, we have to define the sampling function Φ. It should yield an error value, given a specific objective (in this case a clause), and a specific candidate (in this case a valuation). We take the following specification:

$$\Phi_{3-\mathrm{SAT}}(\text{EncodeValuation}(\text{Val}), \text{EncodeClause}(p_1 \vee p_2 \vee p_3)) \overset{\text{spec}}{=}$$
$$\begin{cases} 0 & \text{if Val satisfies } (p_1 \vee p_2 \vee p_3), \\ 1 & \text{otherwise.} \end{cases}$$

The overall error, as specified in definition 1, then equals the number of unsatisfied clauses, which is zero for solutions, and at least 1 for valuations that are not solutions to the problem instance.

In the Sampling chapter, we will give a Sampling function that satisfies this specification.

2.4 General Procedure Format

Using this general problem format, we can define a general procedure format. An Adaptive Sampling algorithm maintains a number μ of candidate solutions, that are adapted for new sample-information in each cycle (this is called iterative improvement). Like the number of objectives k, the number of candidates μ can be one (this is considered default in Neural Computation: using only one neural network).

To define the general procedure format, we first declare the three main data structures that are used by the simulation process:

```
Instance: array[1..$k$, 1..$\ell$] of real;
State: array[1..$\mu$, 1..$m$] of real;
Samples: array[1..$\mu$, 1..$k$] of record
      Candidate: array[1..$m$] of real;
      Objective: array[1..$\ell$] of real;
      Error: real;
   end {record};
```

These data structures are used in the following general procedure, in which two functions are important. One is the sampling function Φ discussed above. The other one is the *adaptation* function α, which adapts the candidates represented in data structure State to the information that is gathered using Φ. It takes care of both the exploitation of this information, and the necessary random exploration. These issues will be discussed in Chapter 6.

The function Init initializes the data structure State with random initial candidates. The function Goal yields *true* if at least one of the candidates is

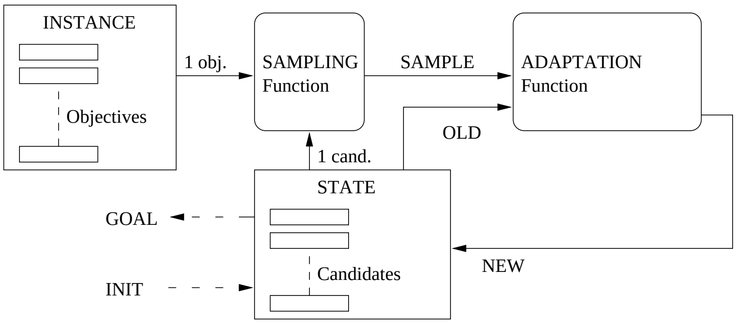

Figure 2.1: The flow of information for Adaptive Sampling.

an ε-solution (for a given ε), and *false* otherwise. Figure 2.1 gives a graphical representation of the data flow for Adaptive Sampling. In pseudo-Pascal, the general Adaptive Sampling procedure is as follows:

```
Instance ← the problem instance to be solved;
Init(State, Random);
repeat
    for $c$ ← 1 to $\mu$
      for $o$ ← 1 to $k$
        Samples[$c$, $o$].Candidate ← State[$c$];
        Samples[$c$, $o$].Objective ← Instance[$o$];
        Samples[$c$, $o$].Error ← $\Phi$(State[$c$], Instance[$o$]);
      rof
    rof
    State ← $\alpha$(State, Samples, Random);
until Goal(State, $\varepsilon$);
```

Let us have a look at what this procedure would do for our running example, the 3-SAT problem in the encoding given above. First, the problem instance to be solved is stored in the data structure Instance. This means that Instance comes to contain a set of vectors that each encode one clause from the expression to be satisfied. Then, State is initialized with random values. So this data structure then contains μ vectors, each encoding a random valuation. In the **repeat-until** loop, each of the μ candidates is evaluated for each of the k objectives, with the function Φ, that yields a 0 for satisfaction of a clause, and a 1 for non-satisfaction. These samples are stored in the data structure Samples, which is input to the adaptation function α, along with State and a random real number between 0 and 1. This adaptation function then produces μ updated encoded valuations, that fill the State data structure. This is repeated until the Goal function yields *true*. It is defendable here to choose ε between 0 and 1, if we assume a solution (with error 0) to exist, and do not want the algorithm to stop when the number of unsatisfied clauses is still 1 or more. We choose the goal function such that

$$\text{Goal}(\text{State}, \varepsilon) = (\exists i : \text{Error}(\text{State}[i, *]) \leq \varepsilon),$$

with

$$\text{Error}(c) = \sum_{i=1}^{k} \Phi(c, \text{Instance}[i, *]),$$

in which for $1 \leq i \leq \mu$,

$$\text{State}[i, *] = (\text{State}[i, 1], \text{State}[i, 2], \ldots, \text{State}[i, m])$$

is the i-th m-vector representing a candidate, and for $1 \leq i \leq k$,

$$\text{Instance}[i, *] = (\text{Instance}[i, 1], \text{Instance}[i, 2], \ldots, \text{Instance}[i, \ell])$$

is the i-th ℓ-vector representing an objective.

The adaptation function should change the state in such a way that it can reasonably be expected to improve on the long run. How to achieve this will be discussed several times in the sequel. First, in the next chapter, an evolutionary algorithm for 3-SAT will be described in terms of an adaptation function. In the Neural Computation chapter, a neural approach will be defined, and in Chapter 6, which is devoted to adaptation in general, the issue will be discussed more exhaustively, and more adaptation functions for 3-SAT will be proposed. Whereas the first two are partly defended by the fact that they are based on genetic and neural processes from nature, respectively, the others will only be constrained to be based on the common principles of these processes. It is part of the hypothesis of this thesis that this will enable the design of a more appropriate and therefore better adaptation function. This will be tested in the experiments.

The general procedure given above will be the basis for defining what Adaptive Sampling is, and in what way Evolutionary and Neural Computation are subfields of it. The sampling function Φ is discussed in Chapter 5. It yields the error that is to be minimized, and is therefore important for defining the problem. In the next two chapters, describing Evolutionary and Neural Computation, we just regard the sampling function as an arbitrary error function, as we discuss the adaptation function α in the following chapter.

Chapter 3

Evolutionary Computation

3.1 How EC Fits in Adaptive Sampling

In the Adaptive Sampling framework, Evolutionary Computation is characterized by the following features:

- There usually is more than one candidate solution (for Evolutionary Computation, the number of candidate solutions μ is sometimes mapped to the population size, and sometimes to the offspring size — details follow where appropriate).
- Possibly, there is only one objective (there is not necessarily a data set involved in the problem format).
- There is much exploration (new candidates are formed by random variation).
- The information from the samples is only used to decide *whether* a candidate should be altered totally, partially or not; there is no structural error assignment to decide *which part* of a candidate to alter.

This last one is the most important characteristic, and might be both the strength and the weakness of Evolutionary Computation in comparison with Neural Computation. On the one hand, it makes the approach applicable to problem classes for which structural error assignment is not possible, or hard instances, where structural error assignment might be misleading.

On the other hand, if the problem is such that it is known *which* features of the candidate should be altered to improve the evaluation on a certain objective, it can be a waste of information to change a *random* feature, instead of the one suggested. We will see examples of both in the experiments (Chapter 7).

The terminology used in Evolutionary Computation can be mapped onto the terms used in this description as follows:

- candidates are called “individuals”,
- the State is called the “population before selection” (not to be confused with the population *after* selection, which is smaller, of course),

- the sampling function Φ calculates (the k components of) the "fitness function",
- the evaluation Error(State$[c, *]$), calculated by (k iterations of) $\Phi(c, o)$, is called the "fitness of individual c",
- the adaptation function α is divided into "selection" (deleting unfit individuals), "recombination" (exploration of new candidates that combine features from old ones, with random influences) and "mutation" (exploration of new candidates that combine features from *one* old candidate with slight random influences).

In the general procedure, there are several variables and functions that can be filled in. For Evolutionary Computation, a lot of these can be derived from the fitness function used. The data structure Instance represents this fitness function. Sometimes, for instance in the case of 3-SAT, the fitness calculated by this fitness function can be said to represent *several* distinct objectives (clauses, for 3-SAT). If so, those objectives are represented by k vectors of length ℓ. If not, there is only $k = 1$ objective. This vector can then be decoded into a fitness function, for instance with a C++-interpreter.

Contrary to Neural Computation, there are several candidates. The population is represented in the data structure State, as μ vectors of length m. Here of course μ is the population size before selection, and m is the length of each vector that concretely represents an individual. Init initializes the population (usually this is entirely random), and Goal is the goal that at least one of the candidates (individuals) reaches a certain fitness.

In the general procedure, the sampling function Φ is calculated for every combination of one objective and one candidate. If there is only one objective, calculating Φ for that objective and one candidate boils down to determining that candidate's fitness. If there are several objectives, then several error-values are calculated for each candidate. It is then the task of the adaptation function α to combine these different error values into one overall evaluation. The adaptation function does all the rest, too: deleting candidates that are less fit than others, and replacing them by new candidates, obtained by applying recombination operators to fitter candidates.

As mentioned above, there is one important tabu in Evolutionary Computation, that separates it from other Adaptive Sampling methods:

> In Evolutionary Computation, which *part* of a candidate is changed by recombination and/or mutation operators does *not* depend on information gathered by sampling for that candidate. The samples influence *which* candidates are changed, and *how much*, but not *in what way*.

The information that is used to guide the adaptation always concerns an entire candidate, and therefore there is no structural error assignment for different features of a candidate.

3.2 Different Evolutionary Algorithms

Before anything more specific can be said about EAs, it is necessary to choose one of its specific subclasses. The EA class is usually considered to be the union of at least four subclasses:

- Genetic Algorithms [Hol75, Mic96] (GA), which use bitstrings as individuals and crossover and mutation as operators,
- Evolution Strategies [Rec73, Mic96] (ES), which use vectors of real values as individuals, and adapt the operators on-line,
- Genetic Programming [Koz92, Mic96] (GP), which use expressions as individuals and operations on the parse trees of these expressions as operators,
- Evolutionary Programming [FOW66, Mic96] (EP),which use finite state machines as individuals.

EAs that operate on other mathematical objects, like graphs, permutations or matrices, do not have a specific name.

3.3 Example: A Genetic Algorithm

As an example, a specific algorithm from the GA subclass will now be described. In this example (taken from Michalewicz's textbook [Mic96]), an individual is a bitstring of length $m = 33$. Each of the 33 bits can either contain a 0 or a 1 at any given moment. The population consists of $\mathit{pop\_size} = 20$ individuals $I_1, I_2, \ldots, I_{\mathit{pop\_size}}$. The initial value of each bit in each individual in the population is chosen randomly.

After the initialization, the following cycle is executed repeatedly:

1. Calculate the fitness of each individual,
2. Roulette Wheel Selection,
3. Crossover,
4. Mutation.

This cycle is repeated until no further significant change is observed, or simply for a fixed number of times. Each step, except the first one, changes the contents of the population.

The calculation of the fitness is done by a function $f : \{0,1\}^{33} \rightarrow \mathbb{R}^+$, that is part of the problem instance. The goal is finding the bitstring that has the highest fitness. The fitness function has the m bits of an individual as input and a positive real valued scalar as its output.

The procedure for the Roulette Wheel Selection is as follows:

- Calculate the total fitness F, that is the sum of the fitness values of all individuals,
- Calculate the selection chance p_i for each individual I_i, that is the fitness of I_i divided by the total fitness F,
- Calculate the cumulative selection chances $q_i = \sum_{j=1}^{i} p_j$, and define q_0 to be 0,
- Select *pop_size* individuals by doing the following *pop_size* times:

  - Generate a random scalar r from $(0,1]$,
  - Select individual I_i if $q_{i-1} < r \leq q_i$,

- The selected individuals form the new population after selection.

The next step is applying crossover to the new population. The chance for an individual to undergo crossover is called p_c, and has value 0.25 in this example. Determining to which individual this will happen, is done as follows. For each individual in the new population after selection:

- Generate another random scalar r from $(0,1]$,
- If $r \leq p_c$, the individual is selected for crossover.

The individuals that were selected for crossover in this procedure, are taken out of the population that was created after selection, and together form the mating pool.

The individuals in the mating pool are coupled two-by-two randomly; if the number of individuals in the mating pool is odd, one individual remains unchanged. For each couple formed, a random natural number *pos* ($1 \leq pos \leq m-1$) is determined, and the crossover then proceeds as follows. First, both individuals are cut in two, at the point *after* bit *pos*:

$$\begin{array}{l|l} (b_1 b_2 ... b_{pos} & b_{pos+1} ... b_m) \\ (c_1 c_2 ... c_{pos} & c_{pos+1} ... c_m) \end{array}$$

Then, the parts that come after the cut, are swapped from one individual to the other, which yields:

$$\begin{array}{l|l} (b_1 b_2 ... b_{pos} & c_{pos+1} ... c_m) \\ (c_1 c_2 ... c_{pos} & b_{pos+1} ... b_m) \end{array}$$

The new individuals obtained by this crossover process are put back into the population.

The last step is mutation. For each bit in every individual in the population just obtained, mutation depends on mutation chance p_m, which is the chance on mutation *per individual bit* within each individual in the population. It has value 0.01 in this example.

- For each individual bit, a random scalar r from $(0,1]$ is generated again.
- If $r \leq p_m$, the bit is "flipped" (from 0 to 1 or from 1 to 0); otherwise, the bit remains unchanged.

In the general procedure, this would look as follows. We take $k=1$, which ensures that each individual is evaluated only once in each cycle, and we take $\Phi(c,o)$ and $\mathcal{I}$ such that $\Phi(c,o_1) = f(c)$, with f the fitness function. The number of candidates can simply be taken $\mu = pop\_size = 20$, because then the sampling step of the general procedure amounts to calculating $f(c_i)$ for $1 \leq i \leq 20$. The length of each candidate is $m = 33$. The adaptation function α does the rest: selection, crossover and mutation.

3.4 3-SAT with EC: "SAW-ing EA"

In [BEV96] an Evolutionary Algorithm is presented, that was designed for the 3-SAT problem. It uses Stepwise Adaptation of Weights.

The algorithm starts with one randomly chosen valuation of the variables, and a vector of weights, one weight for each clause, initialized to 1. From this initial valuation, μ candidates are generated[1] by changing one of the Boolean variables from *true* to *false* or vice versa.

These μ valuations are represented by μ tuples of Boolean variables. These μ candidates are evaluated with the fitness function. The fitness function *adds the weights of the satisfied clauses*, to obtain the fitness of a valuation. If all weights are 1, then this yields the number of satisfied clauses; if for instance a certain clause has weight 2, then this clause counts double, etcetera.

When all the fitness values have been calculated, the one[2] valuation with the highest fitness is picked out, and the rest is deleted. After this, we are back at one candidate, and the cycle can start anew by generating μ new candidates.

After each 250 cycles, the weights are adapted. For this, the valuation is used that has just been picked as the best one. If a clause is not satisfied by this valuation, its weight is increased by 1. The rest of the weights stay the same. This way, clauses that are hard to satisfy, will get ever higher weights, which means that they get a more important role in the fitness function.

The SAW-ing mechanism was introduced in [EvdH97, BEV96]. It adapts the weights in the summation to distinguish between clauses that are rarely satisfied by the candidate solutions from clauses that have shown to be easy to satisfy.

An optimized number μ of candidates is produced in each cycle. For problem sizes 20, 40, 60, 80 and 100, μ is 6, 8, 10, 11 and 12, respectively. Consistent with the Adaptive Sampling model, each cycle consists of sampling and adaptation. The sampling is done by evaluating the μ valuations. The adaptation of the state does not depend on *which* clauses are satisfied; the fitness values determine which candidate is picked out to generate the μ new candidates from. Those are generated by applying the MutOne-operator, which changes exactly one variable (from *true* to *false* or vice versa), that is chosen at random. This does not depend on whether this particular variable has any connection to the evaluation. In the tradition of Evolutionary Computation, there is no structural error assignment (the outcome of the evaluation is not assigned to one specific variable in the valuation).

The results from the tests that were done with this algorithm, were better than that of any known heuristic method for 3-SAT. For more details on this study, see [BEV96]. That paper served as a starting-point for testing whether the Adaptive Sampling model can provide a way of thinking about problems and methods that leads to better algorithms.

[1] This time, the number of candidates μ that is evaluated in one cycle is called the offspring size, in EC-terminology.

[2] In EC-terminology, the population size is 1 for this algorithm.

Chapter 4

Neural Computation

4.1 How NC Fits in Adaptive Sampling

In the Adaptive Sampling framework, Neural Computation (for an overview of the field, see Hertz et al. [HKP91]) is characterized by the following features:

- There usually is only one candidate (one neural network).

- There are a number of objectives (the data set).

- There is usually little explicit exploration.

- There is Structural Error Assignment.

This last one is the most important characteristic, and might be both the strength and the weakness of Neural Computation. It is the main difference with Evolutionary Computation.

In the general procedure, there are several variables and functions that can be filled in. For Neural Computation, the data structure Instance represents the data set. The data set has k elements, which are vectors of length ℓ (or tuples of vectors, whose total length is ℓ). The data structure State contains the weight vectors that represent the Neural Network. These are stored in $\mu = 1$ vector of length m.

Init initializes the concrete candidate vector (the weight vectors), usually to small values that are determined at random, and Goal is the goal that the sum of errors on data set elements drops beneath a certain level.

In the general procedure, the sampling function Φ is calculated for every combination of one objective and one candidate. For Neural Computation, there is only one candidate, the neural network, so calculating Φ (for that candidate and one objective) boils down to determining the error of the network function on one data set element.

The adaptation function does all the rest: calculating which specific weights the error can be assigned to, and changing those weights in the direction that decreases the error. It operates on the parameters (weight vectors) of the network. The archtecture of the network, which defines the interpretation of these weight vectors, is captured in the sampling function, as will become clear in the example to come.

As mentioned above, there is one important mechanism in Neural Computation, that separates it from other Adaptive Sampling methods:

> In Neural Computation, the *part* of a candidate that is changed by the learning rule *always* depends on information gathered by sampling for a specific objective. The samples explicitly influence which *parts* of the candidate are changed, *how much* and *in what direction.*

The information that is used to guide the adaptation always concerns one specific objective, and therefore structural error assignment is possible for different features of a candidate. This does bring the risk of early convergence, because always chosing the path that *looks* the most promising, might underestimate other routes to even better optima. The French proverb "reculer pour mieux chauter" (stepping back to jump better) does not apply for Neural Computation, which separates it from other Adaptive Sampling methods.

Before we introduce our example Neural Network, we will look at some terminology used in Neural Computation:

- the single candidate is called the "neural network",
- the Instance is called the "data set" — the objectives in it are "data set elements".
- the sampling function Φ evaluates the "network outputs" which are obtained by applying the "network function" to the "network inputs".
- the adaptation function α implements the "learning rule".

4.2 Example: A BackProp Neural Network

A neural network (NN) is a set of interconnected neural nodes. In this example, a feedforward NN with one hidden layer is considered. These have the additional property that the neural nodes are organized in three layers. The *input nodes* form the input layer, the *hidden nodes* form the hidden layer, and the *output nodes* form the output layer. Not every two nodes are directly interconnected. Each hidden node is connected with each input node and with each output node, and that are the only connections there are.

A weight is associated with each connection. Some more specific terminology:

- The number of input nodes is called n_{in},
- The number of hidden nodes is called n_{hidden},
- The number of output nodes is called n_{out},
- The weight of a connection from input node i to hidden node h is called $w_{h\leftarrow i}$,
- The weight of a connection from hidden node h to output node o is called $w_{h\rightarrow o}$.

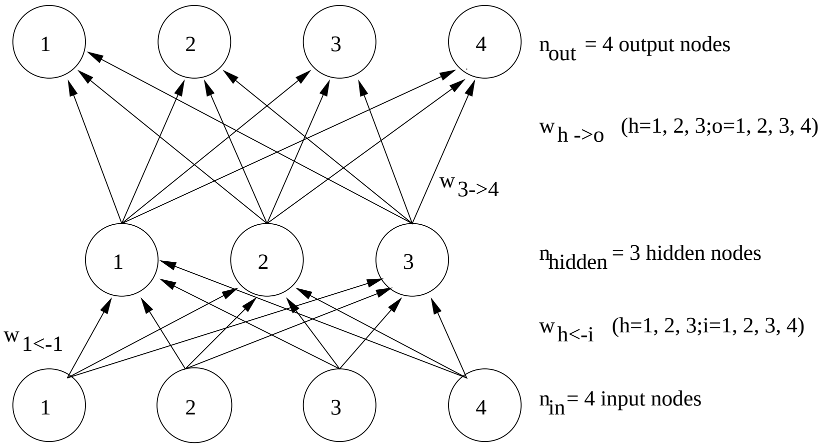
1
2
3
4
n_out = 4 output nodes
w_h ->o (h=1, 2, 3;o=1, 2, 3, 4)
w_3->4
1
2
3
n_hidden = 3 hidden nodes
w_h<-i (h=1, 2, 3;i=1, 2, 3, 4)
w_1<-1
1
2
3
4
n_in = 4 input nodes

Figure 4.1: A feedforward neural network with one hidden layer.

- The vector $(w_{h\leftarrow 1}, w_{h\leftarrow 2}, \ldots, w_{h\leftarrow n_{\text{in}}}, w_{h\rightarrow 1}, w_{h\rightarrow 2}, \ldots, w_{h\rightarrow n_{\text{out}}})$ is called the *concatenated weight vector* $\vec{W}_h$ associated with hidden node h.

Given an architecture $(n_{\text{in}}, n_{\text{out}}, n_{\text{hidden}}) \in \mathbb{N}^3$, the neural network paradigm defines a mapping $\text{NN\_BUILD}_{(n_{\text{in}}, n_{\text{out}})}$ from concatenated weight vectors to hidden node behaviors, and a mapping $\text{NN\_STRUCT}_{(n_{\text{in}}, n_{\text{hidden}}, n_{\text{out}})}$ from hidden node behaviors to network behaviors. Those mappings are as follows; for an (input, output)-pattern $\mu = (\xi, \zeta)$, let

- *excitation* E_h of hidden node h be $\sum_{i=1}^{n_{\text{in}}} \xi_i \cdot w_{h\leftarrow i}$,
- *activation* A_h of hidden node h be $\text{Sigmoid}(E_h)$ for some predefined inversible sigmoid-shaped function Sigmoid,
- for $o = 1, 2, \ldots, n_{\text{out}} : [X_h]_o = \text{Sigmoid}(w_{h\rightarrow o} \cdot A_h)$.

Now we can define what each of the hidden nodes contributes to the behavior of the network as a whole. These contributions are called hidden node behaviors. The hidden node behavior B_h of hidden node h with weight vector $\vec{W}_h$ is

$$B_h = \text{NN\_BUILD}_{(n_{\text{in}}, n_{\text{out}})}(\vec{W}_h), \text{ such that}$$

$$B_h(\xi_1, \xi_2, \ldots, \xi_{n_{\text{in}}}) = ([X_h]_1, [X_h]_2, \ldots, [X_h]_{n_{\text{out}}}),$$

and the network behavior *NN*, given m hidden node behaviors $B_1, B_2, \ldots, B_m$, is:

$$\text{NN} = \text{NN\_STRUCT}_{(n_{\text{in}}, n_{\text{hidden}}, n_{\text{out}})}(B_1, B_2, \ldots, B_m), \text{ such that}$$

$$\text{NN}(\xi_1, \xi_2, \ldots, \xi_{n_{\text{in}}}) = (N_1, N_2, \ldots, N_{n_{\text{out}}}),$$

where

$$N_o = \text{Sigmoid}\left(\sum_{h=1}^{m} \text{Sigmoid}^{-1}[X_h]_o\right).$$

Error Back Propagation [BH69, HKP91] was designed for neural network curve fitting. The curve fitting problem class will be discussed in Section 12. The neural network should “learn” the function specified by a set of (input, output)-patterns $\mu^p = (\xi^p, \zeta^p)$ $(p = 1, 2, \ldots)$

For a learning rate η, the procedure for error back propagation learning is as follows:

1. Initialize the weights to small random values,
2. Choose a pattern $\mu = (\xi^\mu, \zeta^\mu)$,
3. Calculate the output vector N of the network, given input ξ^μ:
$$N^\mu = NN(\xi^\mu)\,,$$
4. For each output node o, compare the output associated with the input by the network to the desired output, which the network should learn:
$$\delta_o^{\text{out}} = \text{Sigmoid}'(\text{Sigmoid}^{-1}(N_o^\mu))[\zeta_o^\mu - N_o^\mu].$$
5. For each hidden node h, calculate a
$$\delta_h^{hidden} = \text{Sigmoid}'(E_h^\mu) \cdot \sum_{o=1}^{n_{\text{out}}} [\vec{W}_h]_{n_{\text{in}}+o} \cdot \delta_o^{\text{out}},$$
in which E_h^μ is the excitation of hidden node h, given input vector ξ^μ,
6. For $1 \le o \le n_{\text{out}}$, use
$$[\vec{W}_h]_{n_{\text{in}}+o} := [\vec{W}_h]_{n_{\text{in}}+o} + \Delta[\vec{W}_h]_{n_{\text{in}}+o}$$
to update output weight $w_{h\to o}$, in which
$$\Delta[\vec{W}_h]_{n_{\text{in}}+o} = \eta \cdot \delta_o^{\text{out}} \cdot A_h\,,$$
and for $1 \le i \le n_{\text{in}}$, use
$$[\vec{W}_h]_i := [\vec{W}_h]_i + \Delta[\vec{W}_h]_i$$
to update input weight $w_{h\leftarrow i}$, in which
$$\Delta[\vec{W}_h]_i = \eta \cdot \delta_h^{\text{hidden}} \cdot \xi_i^\mu\,.$$

4.3 3-SAT with NC: “Neural Satisfaction”

The representation of 3-SAT as a Cloud Fitting problem, which will be presented in the next chapter, suggests that the Evolutionary Algorithm described earlier —though it is better than any heuristic method— uses a suboptimal choice of flow of information. The Evolutionary Algorithm uses only one scalar fitness value into which the information about the satisfaction of all the clauses has to be compressed. In this section, we will present a Neural Network that uses the specific information from each clause to update its state (Structural Error Assignment).

Neural networks are usually applied to data mining or prediction. The application of a Neural Network to the 3-SAT problem might not seem an obvious choice. However, the challenge of the results from the EC approach was taken on, because the Adaptive Sampling model suggests that it is a waste of information to use a fitness function when there is more than just a scalar evaluation available from the problem instance.

In every cycle, a clause was selected at random and presented at the inputs of a neural network with no hidden layer. Because neural networks take vectors of real values as their input (and not logical disjunctions), we define each objective as the vector of length ℓ, whose elements are as follows: If variable i does not occur in the clause, input i is 0. If variable i occurs positively ('$\ldots \vee x_i \vee \ldots$'), then input i is 1. And if variable i occurs negatively in the clause ('$\ldots \vee \overline{x_i} \vee \ldots$'), then input i is set to -1.

There is one output node in the neural network used. Each input node i is connected to this output node with a connection value[1] w_i. This connection value can be any real number between -1 and 1. The vector of all these values, $(w_1, w_2, \ldots, w_m)$, is the vector of connection values, which represents a valuation of variables $x_1, \ldots, x_m$: excitatory connections (positive values) represent *true* and inhibitory connections (negative values) represent *false.*

The network output can be calculated using an activation function that is specialized for 3-SAT; it is 1 if the connection vector corresponds to a valuation that satisfies the clause, and 0 if not. Therefore, we can transform the problem of satisfying each clause into a supervised learning problem with this strict special activation function: the target output associated with each clause is 1. So *learning* a clause with this model means *satisfying* it with the connection vector.

The learning rule was chosen as follows. If a clause is satisfied by the connection vector, then the network output is 1, like the target output, so the output error is zero. In this case, at least one of the three variables in the clause was satisfied, which means that at least one of the three connection values that received a non-zero input had the correct sign (+ or $-$). The connections that contribute to the satisfaction of a clause are strengthened (changed toward the correct $+1$ or -1). This can apply to either one, two or three connections. If none of the three values involved has the right sign the output activation is zero, which means the clause that was represented at the inputs is not satisfied. Note that in this case it would suffice to invert one of the three values (change its sign). However, there is no information available on which one this should be. Therefore, the penalty for not satisfying a clause goes to all three values involved: they are all weakened (changed toward the correct $+1$ or -1). If at least one of the three values passes 0 in this change, the new connection vector will satisfy the clause next time, so the clause has been "learned". Letting $A(c)$ be the output activation (network output) for a given clause c, the learning rule thus becomes, for learning rate η,

- If the output activation $A(c)$ is 1 for input vector c (indicating satisfaction

[1] In the description of this algorithm, we use the word 'connection value' instead of 'weight' to avoid confusion with the weights used by the Stepwise Adaptation of Weights mechanism.

of the clause represented by c), each value w_i is updated using

$$w_i \leftarrow \begin{cases} (1-\eta)\cdot w_i + \eta \cdot c_i & \text{if } c_i, w_i \text{ are } \neq 0 \text{ and have the same sign} \\ w_i & \text{otherwise} \end{cases}$$

- If the output activation $A(c)$ is 0 (indicating non-satisfaction of the clause represented by c), each value w_i is updated using

$$w_i \leftarrow \begin{cases} (1-\eta)\cdot w_i + \eta \cdot c_i & \text{if } c_i \neq 0 \\ w_i & \text{otherwise} \end{cases}$$

The algorithm, with learning rate $\eta = 0.3$, will be referred to as "Neural Satisfaction" in the chapter on the experiments. For the experiments, we used a slight variation of it. In the General Procedure Format (Section 2.4), the implementation is defined to be epoch-wise. This means that all samples are gathered first, and then all adaptations are made. In Neural Computation, epoch-wise learning is used for theory and proofs of convergence, but implementations are hardly ever epoch-wise: they execute the applicable part of the adaptation-step after each sample (immediate adaptation).

For the general procedure format, this possibility was not included, because it would make the procedure unnecessarily complex. However, we *did* want to copy the use of immediate adaptation from the tradition of Neural Computation in the implementation of the "Neural Satisfaction"-method, because this is usual in Neural Computation, and has showed to increase performance in most cases. So although its definition is in an epoch-wise format, the tests were done with its immediate adaptation variant.

In connection with this, the order in which the clauses are sampled is chosen at random. Note that this does not matter for an epoch-wise implementation, but it does here: a random sampling order can take away biases in the interactions between adaptations for different clauses, that might be caused by the order in which they appear in the original problem instance. A fixed sampling order might cause the adaptations made for clause $i+1$ to overwrite the adaptations for clause i, which could make it impossible to satisfy clause i, if this makes it harder to satisfy clause $i+1$. In the implementation used for the experiments, it is even possible that certain clauses are investigated more than once, and others are not investigated at all during that cycle.

Chapter 5

Sampling

This chapter discusses the general problem format in further detail. Bear in mind that wherever a function is given in a problem instance, this does not imply that there also is an explicit expression available for this function. Functions as such can be seen as black boxes that return a function value when given an input value.

Because the interval $[-1, 1]$ will be used often in the definitions to come, it will be denoted using $\mathcal{L}$:

- $\mathcal{L}(1)$ stands for $[-1, 1]$,
  - for instance, $(0.87) \in \mathcal{L}(1)$ (in this case the parenthesis are often omitted),
- $\mathcal{L}(n)$ stands for $[-1, 1]^n$, for $n = 1, 2, \ldots$,
  - for instance, $(-0.5, 0.3, 0.87) \in \mathcal{L}(3)$,
- For $a \in \mathcal{L}(n)$ and $1 \leq i \leq n$, a_i is the i-th element of a,
  - for instance, $(-0.5, 0.3, 0.87)_2 = 0.3$,
- $\mathcal{L}(m \to n)$ stands for $\{f : \mathcal{L}(m) \to \mathcal{L}(n)\}$,
- Given two vectors $x \in \mathcal{L}(m)$ and $y \in \mathcal{L}(n)$, their concatenation $x \bullet y$ is defined as $(x_1, x_2, \ldots, x_m, y_1, y_2, \ldots, y_n)$, which is an element of $\mathcal{L}(m+n)$,
  - for instance, $(-0.5, 0.3) \bullet (0.87) = (-0.5, 0.3, 0.87)$,
- Likewise for tuples in general, given a tuple x of length m and a tuple y of length n, their concatenation $x \bullet y = (x_1, x_2, \ldots, x_m, y_1, y_2, \ldots, y_n)$.

5.1 Cloud Fitting

Cloud Fitting is a new problem class. Its power is in the fact that it includes four existing problem classes: optimization, curve fitting, tesselation and simple reinforcement learning. It is defined in terms of a new mathematical concept called *cloud*, that is a generalization from three existing mathematical concepts: vectors, schemata and functions.

A cloud is a *tuple* defining a *set*. Before introducing its definition, we will first define how schemata, as used in the analysis of Genetic Algorithms, can be seen as tuples defining sets. After that, we will define four types of clouds: vectors, functions, and two combinations of those, obtained by concatenation.

5.1.1 Schemata as tuples defining sets

A schema (plural: schemata) is a tuple $(s_1, s_2, \ldots, s_n)$, where s_i should be either 0 or 1 or $\sim$, so for instance $(0, 1, \sim, 1)$ is a schema. Such a schema defines a set of bitstrings that agree with it. Intuitively, the don't care-symbol $\sim$ means that either a 0 or a 1 can be chosen to replace it. This is how it it possible that different bitstrings agree with the same schema. We define the function *Schema*, that takes a length n, and a tuple of this length containing only 0's, 1's and $\sim$'s, and yields the *set of agreeing bitstrings* (i.e., the set of all bitstrings that agree with the schema).

DEFINITION 2 Given a schema $(s_1, s_2, \ldots, s_n)$ of length n, where for all i $(1 \leq i \leq n)$, $s_i \in \{0, 1, \sim\}$, we define $\mathit{Schema}(n, (s_1, s_2, \ldots, s_n))$ to be the set of all vectors v such that:

- $v_i = 0$ if $s_i = 0$,
- $v_i = 1$ if $s_i = 1$,
- $v_i \in \{0, 1\}$ if $s_i = \sim$.

So for instance $\mathit{Schema}(0, 1, \sim, 1)$ is the set $\{(0, 1, 0, 1), (0, 1, 1, 1)\}$. In this example, there is one don't care symbol (in the third position), that can be replaced with either a 0 or a 1. This definition of schemata is an example of the basic setup for a tuple defining a set of vectors (here bitstrings).

5.1.2 Functions as tuples defining sets: (r, s)-clouds

Now we will define how we can see function as tuples defining sets. In this context, we denote functions with a concatenation of two tuples. The first tuple represents the inputs of the function, and contains only don't care symbols. The second tuple represents the outputs of the function, and contains mathematical expressions. We concatenate these two tuples to obtain one tuple, that defines the function. So for instance $(\sim, \sim, \sim, [{x_1}^2 + {x_2}^2 + {x_3}^2])$ is a function, because it is the concatenation of $(\sim, \sim, \sim)$, which is a tuple of don't care symbols, and $[{x_1}^2 + {x_2}^2 + {x_3}^2]$, which is an expression.

The yield of this expression is the output of the function (it has a one-dimensional output), and its free variables x_1, x_2, and x_3, are the inputs of the function. Here, x_1 in the expression corresponds to the position of the first don't care symbol $\sim$, x_2 to the second, and x_3 to the third.

To define what this function does, we will define a set of vectors that *agree* with the function. A vector that agrees with a function, is the concatenation $x \bullet y$ of an input vector x and an output vector y ($\bullet$ denotes concatenation). For this vector to agree with the function, the output vector y should be the expression's output for input vector x. We will now formalize this, with the *Fun*-function, that takes an input vector length r, an output vector length s, and a defining tuple f, such that for $1 \leq i \leq r$, $f_i = \sim$, and for $r+1 \leq i \leq r+s$, f_i is a mathematical expression with free variables $x_1, x_2, \ldots x_r$, each of which corresponds to one input dimension. Taking these arguments, *Fun* yields the set of all agreeing vectors:

DEFINITION 3 Given a tuple $(f_1, f_2, \ldots, f_r, f_{r+1}, \ldots, f_{r+s})$ of length $r+s$, where for all i $(1 \leq i \leq r)$, $f_i = \sim$, and for all i $(r+1 \leq i \leq r+s)$, f_i is a mathematical expression, we define $\mathit{Fun}(r, s, (f_1, f_2, \ldots, f_r, f_{r+1}, \ldots, f_{r+s}))$ to be the set of all vectors $x \bullet y$ such that:

- $x \in \mathcal{L}(r)$, and
- $y \in \mathcal{L}(s)$, such that $y_i = f_{r+i}$ holds, when $x_1, x_2, \ldots, x_r$ are filled in.

So to give an example, $\mathit{Fun}(2, 2, (\sim, \sim, [-{x_1}^2], [-{x_2}^2]))$ is the set of all vectors (x_1, x_2, y_1, y_2) such that $y_1 = -{x_1}^2$ and $y_2 = -{x_2}^2$. So this set contains for instance $(0.5, 0.5, -0.25, -0.25)$ and $(-0.6, 0.3, -0.36, -0.09)$. For an arbitrary tuple a of length n, $\mathit{Fun}(r, s, a)$ is not always defined; only if $r + s = n$ and for every $i : 1 \leq i \leq r$, $a_i = \sim$, and for every $i : r+1 \leq i \leq r+s$, a_i is a mathematical expression containing at most $x_1, x_2, \ldots, x_r$ as free variables, that yields a value in $\mathcal{L}(1)$ if $x \in \mathcal{L}(r)$. From now on, we will refer to function defined in this way as (r,s)-*clouds*: for $r, s \in \mathbb{N}$, tuple a is an (r,s)-cloud, if $\mathit{Fun}(r, s, a)$ is defined.

5.1.3 Vectors as tuples defining sets: (t)-clouds

Because we will want to define the more general clouds as tuples defining sets of vectors shortly, we will need to define a trivial function *Vec*, that translates a vector to the singleton set containing that vector. The only thing it does, is putting the vector in a set with nothing else in it. Ranges are confined to $[-1, 1]$:

DEFINITION 4 Given a vector v of length t in $\mathcal{L}(t)$, we define $\mathit{Vec}(t, v)$ to be the set $\{v\}$.

To show that this function really does nothing important, we give an example: $\mathit{Vec}(3, (0, 1, -0.4)) = \{(0, 1, -0.4)\}$. For an arbitrary tuple a, $\mathit{Vec}(t, a)$ is only defined if $a \in \mathcal{L}(t)$, otherwise it is undefined. From now on, we will refer to vectors defined in this way as (t)-*clouds*: for $t \in \mathbb{N}$, tuple a is an (r,s)-cloud, if $\mathit{Vec}(t, a)$ is defined.

5.1.4 Concatenating vectors with functions and vice versa

Using (r,s)-clouds for functions and (t)-clouds for vectors, we can now define two combinations: (r,s,t)-clouds and (t,r,s)-clouds.

An (r,s,t)-cloud is obtained by concatenating an (r,s)-cloud and a (t)-cloud. It is interpreted with the *FunVec*-function:

DEFINITION 5 Given $r, s, t \in \mathbb{N}$, an (r,s)-cloud f and a (t)-cloud v, we define $\mathit{FunVec}(r, s, t, f \bullet v)$ to be the set containing all vectors $w = p \bullet q$ such that

- $p \in \mathit{Fun}(r, s, f)$ and

- $q \in \mathit{Vec}(t, v)$.

The *FunVec*-function is undefined in all cases not defined by this definition. As an example, $(0.25, 0.5, 0.23) \in \mathit{FunVec}(1, 1, 1, (\sim, \sqrt{x_1}, 0.23))$.

A (t, r, s)-cloud is simply obtained by concatenating an (r, s)-cloud and a (t)-cloud in the opposite order. It is interpreted with the *VecFun*-function:

DEFINITION 6 Given $t, r, s \in \mathbb{N}$, a (t)-cloud v, and an (r, s)-cloud f, we define $\mathit{VecFun}(t, r, s, v \bullet f)$ to be the set containing all vectors $w = p \bullet q$ such that

- $p \in \mathit{Vec}(t, v)$ and
- $q \in \mathit{Fun}(r, s, f)$.

The *VecFun*-function is undefined in all cases not defined by this definition. As an example, $(0.23, 0.25, 0.5) \in \mathit{VecFun}(1, 1, 1, (0.23, \sim, \sqrt{x_1}))$.

5.1.5 The General Definition of Clouds

At this point, we have defined four interpretation functions: *Fun* for (r, s)-clouds, *Vec* for (t)-clouds, *FunVec* for (r, s, t)-clouds, and *VecFun* for (t, r, s)-clouds.

We will now define clouds in general, using these definitions:

DEFINITION 7 Given a tuple $\mathcal{C}$ of length n, we define $\mathit{Cloud}(\mathcal{C})$ as follows:

- If $\exists t : \mathit{Vec}(t, \mathcal{C})$ is defined, then $\mathit{Cloud}(\mathcal{C}) = \mathit{Vec}(t, \mathcal{C})$;
- otherwise, if $\exists r, s : \mathit{Fun}(r, s, \mathcal{C})$ is defined, then $\mathit{Cloud}(\mathcal{C}) = \mathit{Fun}(r, s, \mathcal{C})$;
- otherwise, if $\exists r, s, t : \mathit{FunVec}(r, s, t, \mathcal{C})$ is defined, then we define $\mathit{Cloud}(\mathcal{C})$ to be $\mathit{FunVec}(r, s, t, \mathcal{C})$;
- otherwise, if $\exists t, r, s : \mathit{VecFun}(t, r, s, \mathcal{C})$ is defined, then we define $\mathit{Cloud}(\mathcal{C})$ to be $\mathit{VecFun}(t, r, s, \mathcal{C})$.
- If none of these four are defined, $\mathit{Cloud}(\mathcal{C})$ is $\emptyset$.

5.1.6 Cloud Application

Clouds can be "applied" to another cloud, just as a function can be applied to a vector. Cloud application can be defined syntactically, but this involves extra renumbering of indices, which is not very enlightening. Instead, we give the semantical definition here:

DEFINITION 8 Given two arbitrary clouds $\mathcal{C}$ and $\mathcal{D}$ of the same length,

$$\mathit{Cloud}(\mathcal{C}\ \mathit{AppliedTo}\,\mathcal{D}) =$$

$$\{v | v \in \mathit{Cloud}(\mathcal{C}) \wedge \exists w \in \mathit{Cloud}(\mathcal{D}) : (\mathcal{C}_i = \sim\ \Rightarrow v_i = w_i)\}.$$

Note that this set might be empty. Intuitively, this definition says that the free elements in $\mathcal{C}$ should be filled in with corresponding elements from some vector w in $\mathcal{D}$, which corresponds to applying the function to the values needed from this vector w.

With this, we can define observation, which is the operator that is used in the definition of Cloud Fitting. It uses the *AppliedTo*-operator twice. If this yields a singleton set, the unique vector in this set, is the result of the *Observes*-operator:

DEFINITION 9 Given two arbitrary clouds $\mathcal{C}$ and $\mathcal{D}$ of the same length,

$$\mathcal{C}\, \mathit{Observes}\, \mathcal{D} = c, \text{ such that}$$

$$\mathit{Cloud}(\mathcal{C}\, \mathit{AppliedTo}(\mathcal{D}\, \mathit{AppliedTo}\, \mathcal{C})) = \{c\}.$$

Note that $\mathcal{C}\, \mathit{Observes}\, \mathcal{D}$ does not necessarily exist. However, it does exist in all the cases that we will meet in the sections to come. For optimization, where objectives represent objective functions and candidates represent vectors, it is equivalent to applying the objective function to a candidate. For supervised and unsupervised learning, where objectives are dataset elements and candidates are network functions, it is equivalent to calculating network outputs.

For reinforcement learning, the case is somewhat more complex, because both objectives and candidates represent functions. This is the only case for which it is necessary that the *AppliedTo*-operator occurs twice in the definition of the *Observes*-operator. We will see that the use of clouds for defining sampling functions is simpler than it might look at first glance.

5.1.7 Sampling in terms of clouds

We now redefine the sampling function Φ using two models: an *objective model*, that converts the concrete objectives stored in the Instance data structure to corresponding *objective clouds* $\mathcal{O}$, and a *candidate model*, that converts the concrete candidates stored in the State data structure to corresponding *candidate clouds* $\mathcal{C}$. With these, the sampling function determines the Euclidean distance between the vector that is obtained by letting $\mathcal{C}$ observe $\mathcal{O}$, and the vector that is obtained by letting $\mathcal{O}$ observe $\mathcal{C}$:

$$\Phi(c, o) = ||\mathcal{O}\, \mathit{Observes}\, \mathcal{C} - \mathcal{C}\, \mathit{Observes}\, \mathcal{O}||,$$

where $\mathcal{O} = \text{ObjectiveModel}(o)$ and $\mathcal{C} = \text{ObjectiveModel}(c)$. Furthermore, $|| \cdot ||$ denotes the Euclidean norm. Substituting this definition of Φ in the definition of the general problem format given in Section 5.1.7, brings us the Cloud Fitting problem class.

In sections to come, it will become clear how this problem class can be used, with the correct objective model and candidate model, to represent optimization, tesselation, curve fitting or reinforcement learning. The next section describes how the two models can be instantiated for the running example 3-SAT.

5.2 Sampling for 3-SAT

The 3-SAT problem is a constraint satisfaction problem: a number of constraints (clauses in this case) are given, and a solution is any valuation of the free

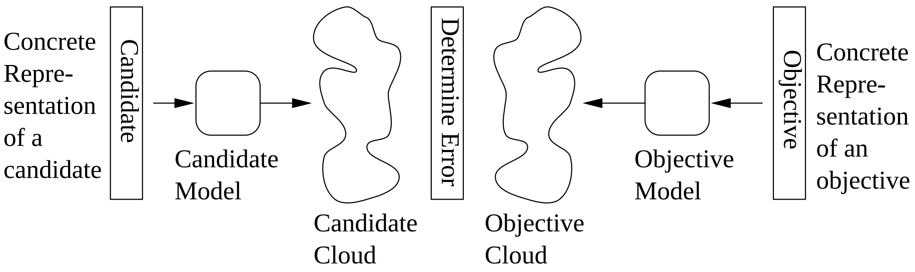

Figure 5.1: The relations between the different clouds and the corresponding concrete representations.

variables that satisfies all constraints at the same time. Note that such a solution does not necessarily exist. However, the methods discussed in this thesis are incomplete methods, that are not guaranteed to find a solution, and that cannot distinguish *whether* a solution exists. If they try to solve an instance with no solution, they will never halt. Therefore, we only used solvable instances in the simulations that will be discussed in Chapter 7, and in the discussion of 3-SAT methods, we will only consider instances of the problem class for which a solution exists.

Before constraint satisfaction problems can be represented in the Adaptive Sampling framework, they should be transformed to Cloud Fitting problems. Recall that in Section 2.3.1, we gave a specification for $\Phi_{3-\mathrm{SAT}}$:

$$\Phi_{3-\mathrm{SAT}}(\mathrm{EncodeValuation}(\mathrm{Val}), \mathrm{EncodeClause}(p_1 \vee p_2 \vee p_3)) \overset{\mathrm{spec}}{=}$$

$$\begin{cases} 0 & \text{if Val satisfies } (p_1 \vee p_2 \vee p_3), \\ 1 & \text{otherwise.} \end{cases}$$

There are two easy ways to satisfy this specification. One is to let the objective model translate objectives into (t)-clouds representing clauses and to let the candidate model translate candidates into (r,s)-clouds representing a function that yields 1 for a clause that is satisfied by the candidate, and 0 for any other clause.

The other way is to let the candidate cloud be a (t)-cloud representing a valuation, and let the objective cloud be an (r,s)-cloud, representing a function that yields 1 for a valuation that satisfies the objective, and 0 for any other valuation.

In Neural Computation, objectives are traditionally seen as vectors (data set elements) and the candidate(s) as (network) functions. In Evolutionary Computation, at least for Genetic Algorithms, this is the other way around: objectives are represented with (fitness) functions, and candidates are represented with vectors (chromosomes).

As desired, the Adaptive Sampling framework is not biased towards either representation. We will give them both:

In the habit of Neural Computation, for an objective $o \in \mathcal{L}(m)$ and a candidate $c \in \mathcal{L}(m)$,

$$\mathrm{ObjectiveModel}(o) = (o_1, o_2, \ldots, o_m, 1) \text{ and}$$

$$\text{CandidateModel}(c) = (\underbrace{\sim,\sim,\ldots,\sim}_{m},(\prod_{i=1}^{m} min(x_i \cdot c_i + 1, 1)).$$

The 1 concatenated at the end of ObjectiveModel(o) indicates that 1 is the target value for the output of the "network function" CandidateModel(c).

In the habit of Evolutionary Computation,

$$\text{CandidateModel}(c) = (c_1, c_2, \ldots, c_m, 1), \text{ and}$$

$$\text{ObjectiveModel}(o) = (\underbrace{\sim,\sim,\ldots,\sim}_{\ell},(\prod_{i=1}^{m} min(o_i \cdot x_i + 1, 1)).$$

The 1 concatenated at the end of CandidateModel(c) indicates that 1 is the target value for the output of the "fitness function" ObjectiveModel(o). Note that this representation amounts to the same sampling function Φ as the NC-representation of the problem. In both cases, $\Phi(c, o)$ indicates whether the valuation represented as candidate c satisfies the clause represented as objective o.

5.3 Optimization

In Evolutionary Computation, problems are formulated as (or translated to) optimization problems. Optimization is simply the problem of finding an optimum of a given objective function, but the notation can vary. Sometimes, "optimum" means "minimum" (in the metaphor of error values), and sometimes it means "maximum" (in the metaphor of evolutionary fitness). The number of inputs and their range can vary, as well as the range of the output. Although these are only questions of representation, they need to be answered.

In this scope, the inputs and the output are assumed to be real-valued scalars in between -1 and $+1$. From now on, "optimal" means "minimal", and not maximal. A problem instance should specify an objective function F. We suppose that F yields an error value between 0 and 1, so an ε-solution v of such an instance, is a vector for which the objective function yields an error value of at most ε. The problem class then becomes the following:

DEFINITION 10 An ε-solution to an instance of an optimization problem $(n \in \mathbb{N}, F : \mathcal{L}(n) \rightarrow [0,1])$ is a vector $v \in \mathcal{L}(n)$ such that

$$\forall w \in \mathcal{L}(n) : F(v) \leq F(w) + \varepsilon\,.$$

Optimization can be defined as a subclass of cloud fitting as follows. There is only one objective. It encodes the mathematical expression [EXP$_F$] for the objective function F. The ObjectiveModel should be chosen such that it decodes the objective to the (r, s)-cloud with $r = n$ and $s = 1$, corresponding to F:

$$\text{ObjectiveModel}(o) = (\underbrace{\sim,\sim,\ldots,\sim}_{n},[\text{EXP}_F])$$

Here, [EXP$_F$] is the expression for objective function F from the definition of optimization, with free variables $x_1 \ldots x_n$.

The candidate model should be chosen such that it yields the (t)-cloud $(v_1, v_2, \ldots, v_n, 0)$, which has $t = n + 1$. The concatenated 0 indicates that the goal is to minimize the distance between $F(v)$ and 0 for a candidate solution v, i.e. to minimize $F(v)$. There is only one objective, so the definition of the general problem format reduces to the definition of optimization, with Φ instead of F. As described above, Φ is defined with the "Observes"-operator on clouds. In this case, this is relatively easy:

$$\mathcal{O}\, Observes\, \mathcal{C} = (v_1, v_2, \ldots, v_n, F(v))$$

$$\text{and } \mathcal{C}\, Observes\, \mathcal{O} = (v_1, v_2, \ldots, v_n, 0),$$

$$\text{and thus with } k = 1, \text{ for a candidate } v \in \mathcal{L}(n),$$

$$\text{Error}(v) = \sum_{i=1}^{1} \Phi(v, o_i) = \Phi(v, o_1) =$$

$$\sqrt{(0 + 0 + \ldots + 0 + (F(v) - 0))^2} = |F(v)| = F(v).$$

5.4 Unsupervised Learning

Neural Networks are said to *learn* a data set, instead of to optimize an objective function. However, learning can be described as a subclass of cloud fitting by seeing the elements of the data set as objectives. Apart from reinforcement learning, which is discussed in Section 5.6, two kinds of learning have been studied more extensively: supervised and unsupervised learning. The difference between the two is that for supervised learning, each objective represents a value x and a value $f(x)$, thus defining a feature of the function f that has to be learned. For unsupervised learning, each objective represents simply one value x from a distribution, thus defining a feature of the distribution that has to be learned.

In terms of cloud fitting, it does not matter much whether the candidate solutions represent distributions or functions — in a way, distributions can be seen as functions too, and likewise functions can be seen as simultaneous distributions. Both distributions and functions can be described as clouds, so in both cases we consider the goal of the problem to minimize a cloud fitting error.

In a tesselation problem, a set of prototype vectors is wanted, that identify clusters in a data set. The number of prototypes wanted has to be specified in advance, to define the amount of generalization desired. If the number of vectors in the solution is at least the number of vectors in the data set, the optimal solution has the dataset as a subset, so the tesselation is optimal in a trivial way. If the number of prototypes is smaller than the number of vectors in the dataset, the information in the dataset will have to be compressed in order to obtain an optimal solution; some form of general information will have to be extracted from the specific information from each sample.

A problem instance should specify a number n of elements in a dataset D, and the number e of prototypes wanted:

DEFINITION 11 An ε-solution to an instance of a tesselation problem $(n \in \mathbb{N}, e \in \mathbb{N}, D \subseteq \mathcal{L}(n))$ is a set $S = \{S_1, S_2, \ldots, S_e\}$ of e vectors in $\mathcal{L}(n)$ such that

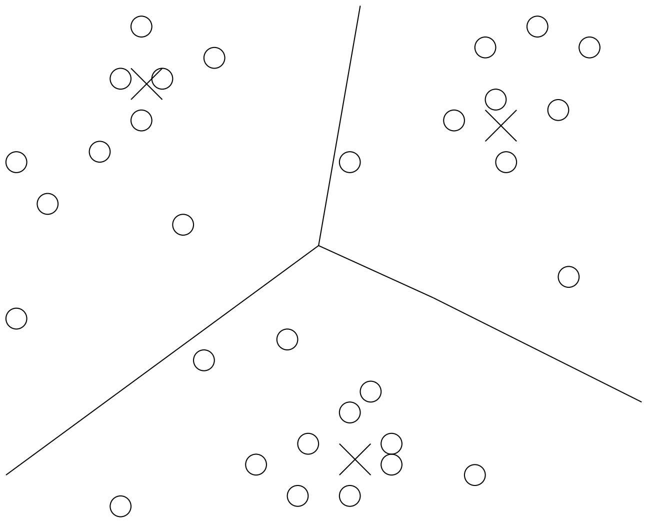

Figure 5.2: A Voronoi diagram; the optimal solution of a tesselation problem with 3 prototypes. Crosses are prototypes, and circles are data set elements.

$\forall T = \{T_1, T_2, \ldots, T_e\}$ with $T_i \in \mathcal{L}(n)$ for $i = 1, 2, \ldots, e$:

$$\sum_{d \in D} \text{Error}(S, d) \leq \sum_{d \in D} \text{Error}(T, d) + \varepsilon\,.$$

where Error(S, d) is the distance between d and the closest prototype in S:

$$\text{Error}(S, d) = \min_{i=1,2,\ldots,e} ||S_i - d||$$

This is also known as Voronoi tesselation, and the result is known as a Voronoi diagram [HKP91] (see figure 5.2 for an example).

In the tesselation problem, the minimum-function makes that the error of a set of prototypes on each sample is influenced only by the one prototype that is closest to that sample. Now we will define Φ for this problem class.

For defining tesselation as a subclass of Cloud Fitting, we take the data set D as our Instance. So each data set element corresponds to an objective. In terms of the definition of tesselation, k is the number of elements in D and the length of each objective is that of each data set element: $\ell = n$. A candidate solution is uniquely defined by a set $S = \{S_1, S_2, \ldots S_e\}$. We represent it with $c = S_1 \bullet S_2 \bullet \ldots \bullet S_e$. The total length of this concatenation is $m = e \cdot n$. The sampling function should yield an error value for a given candidate on one specific objective (data set element). The sum of these errors over the data set, for a candidate $c = S_1 \bullet S_2 \bullet \ldots \bullet S_e$, should be equal to the error of the corresponding set of prototypes $\{S_1, S_2, \ldots, S_e\}$, as defined in Section 11:

$$\sum_{i=1}^{n} \Phi(c, o_i) \overset{\text{spec}}{=} \min_{j=1,2,\ldots,e} ||S_j - o_i||.$$

Because each candidate vector is a concatenation of multiple vectors, the candidate model is not as simple as it was with optimization. We cannot just use the candidate vector as a (t)-cloud. Therefore, we first transcribe the set of prototypes that define the tesselation into a *prototyping function*, and then we use an (r, s)-cloud for the candidate instead. Given a set of prototypes, a prototyping function yields the closest prototype from this set. For instance, if we have a set S of three prototypes in one dimension $\{0.5, -0.8, 0.1\}$, the prototyping function P_S would be

$$P_S(x) = \begin{cases} 0.5 & \text{if } 0.3 < x \leq 1, \\ 0.1 & \text{if } -0.35 < x \leq 0.3, \\ -0.8 & \text{if } -1 \leq x \leq -0.35. \end{cases} .$$

Intuitively, we need the objective cloud to supply both the inputs and the target outputs for this function. Because we want to find a set of prototypes S such that the distance between prototypes yielded by the prototyping function, $P_S(x)$, have minimal distance to the original x, we can simply take input x *itself* as the target output for $P_S(x)$. So for each objective o:

$$\text{ObjectiveModel}(o) = (o_1, o_2, \ldots, o_n, o_1, o_2, \ldots, o_n).$$

Then

$$\text{CandidateModel}(S_1 \bullet S_2 \bullet \ldots \bullet S_e) = (\underbrace{\sim, \sim, \ldots, \sim}_{n}, [\text{EXP}_P]),$$

where given the prototypes $S_1, S_2, \ldots, S_e \in \mathcal{L}(n)$, for an input vector x of length n,

$$[\text{EXP}_P] = S_i,$$

$$\text{such that } \forall j 1 \leq j \leq e : ||S_j - x|| \geq ||S_i - x||.$$

Then as desired, with $o = (x_1, x_2, \ldots, x_n)$, and $c = (S_1 \bullet S_2 \bullet \ldots \bullet S_e)$ such that $\{S_1, S_2, \ldots, S_e\} = S$, and Error$(S, d)$ as defined in Section 11,

$$\Phi(c, o) =$$

$$||(x_1, x_2, \ldots, x_n, x_1, x_2, \ldots, x_n) -$$

$$(x_1, x_2, \ldots, x_n, [P_S(x)]_1, [P_S(x)]_2, \ldots, [P_S(x)]_n)|| =$$

$$||(x_1, x_2, \ldots, x_n) - ([P_S(x)]_1, [P_S(x)]_2, \ldots, [P_S(x)]_n)|| =$$

$$\text{Error}(S, d).$$

5.5 Supervised Learning

Supervised learning is the NC-name for Curve Fitting. The goal in a Curve Fitting problem is finding a function within a certain class, that is as close as possible to a function of which (input, output)-samples are given. Despite the name of the class, the object to be approximated is a function, not just any curve. Furthermore, the solution function can not just be any function, it should have certain predetermined properties. If just any function would be satisfactory, the optimal solution would be the problem function itself, which might not be desired. The solution function should be as simple as possible. This notion is often referred to as Occam's razor [Rus79]. For instance, a restriction could be that the third and higher derivatives of the solution function must be zero. A curve fitting problem is defined with a number of input dimensions n_{in}, a number of output dimensions n_{out}, the length of concrete candidate vectors m, a set of inputs V, a function F that is to be fitted, at least for the inputs in V, and a model Model, that decodes a concrete candidate vector into a function:

DEFINITION 12 An ε-solution to an instance of a Curve Fitting problem

$$(n_{\text{in}} \in \mathbb{N}, n_{\text{out}} \in \mathbb{N}, m \in \mathbb{N}, V \subseteq \mathcal{L}(n_{\text{in}}), F : \mathcal{L}(n_{\text{in}} \to n_{\text{out}}),$$

$$\text{Model} : \mathcal{L}(m) \to \mathcal{L}(n_{\text{in}} \to n_{\text{out}}))$$

is a vector c in $\mathcal{L}(m)$, such that Model(c) is a function $G \in \mathcal{L}(n_{\text{in}} \to n_{\text{out}})$, such that for all H in $\mathcal{L}(n_{\text{in}} \to n_{\text{out}})$:

$$\sum_{v \in V} (F(v) - G(v))^2 \leq \sum_{v \in V} \left[(F(v) - H(v))^2\right] + \varepsilon.$$

In this case, we take k to be the cardinality of V, and the concrete objectives to be k vectors, each corresponding to one element of V. Suppose $x \in V$ and x corresponds to objective o_i. Then, $o_i = x \bullet F(x)$, where F is the function to be fitted. This means that $\ell = n_{\text{in}} + n_{\text{out}}$. The objective model transcribed such a vector into the corresponding (t)-cloud. So for $x \in V$ a test set element, $F \in \mathcal{L}(n_{\text{in}} \to n_{\text{out}})$ the target function, and o_i the specific objective corresponding to test set element x,

$$\text{ObjectiveModel}(o_i) = (x_1, x_2, \ldots, x_{n_{\text{in}}}, [F(x)]_1, [F(x)]_2, \ldots, [F(x)]_{n_{\text{out}}}).$$

For instance, if $F(x) = {x_1}^2 + {x_2}^2$ and $x = (0.3, 0.4) \in V$, then one of the concrete objectives (the one corresponding to x), is $o_i = (0.3, 0.4, 0.25)$ and the objective cloud for o_i is ObjectiveModel(o_i) $= \mathit{Vec}(3, (0.3, 0.4, 0.25)) = \{(0.3, 0.4, 0.25)\}$.

The candidate model converts concrete candidate vectors of length m into functions. For instance, if the model is a Neural Network, then the concrete candidate is a set of weight vectors (*that* is what is stored in the computer's memory), and the model converts these vectors into the corresponding network function. As the reader might have expected, we derive the candidate model from the model given in the problem definition:

$$\text{CandidateModel}(c) = (\underbrace{\sim, \sim, \ldots, \sim}_{n_{\text{in}}}, [\text{EXP}_1], [\text{EXP}_2], \ldots, [\text{EXP}_{n_{\text{out}}}]),$$

$$\text{such that } [\text{Model}(c)](x) = ([\text{EXP}_1], [\text{EXP}_2], \ldots, [\text{EXP}_{n_{\text{out}}}]) \text{ holds.}$$

Then, as was specified, $||\mathcal{C}\ \mathit{Observes}\ \mathcal{O} - \mathcal{O}\ \mathit{Observes}\ \mathcal{C}||$ yields the Euclidean distance between the target output $F(x)$ and the approximating output $[\text{Model}(c)](x)$, which is produced by the fitting function $[\text{Model}(c)]$.

5.6 Simple Reinforcement Learning

Simple Reinforcement Learning means Reinforcement Learning (RL) in a stationary environment, with a non-stochastic reinforcement signal, that represents a direct-reward evaluation of the last action. The environment is assumed to have no hidden state and to be deterministic. This means that any (state, action)-combination yields the same reinforcement signal under any circumstance.

RL is like optimization in many ways, but now the input of the function that is to be optimized, is not a vector, but is itself a function (possibly even a function with multiple outputs), called a "policy". The ranges are constrained to be $[-1, 1]$ again. Advanced versions of this class have also been studied [Sut88], in which stochastic values play a role [BBS91], or the iterative closure of the solution [Wat89, RN95]. These are not considered here. A candidate (policy) has n_{in} inputs and n_{out} outputs. The output of the objective function is a scalar, the reinforcement signal. If the reinforcement signal has a positive value, this can be interpreted as a reward; a negative reinforcement signal can be interpreted as a penalty. The average reinforcement should therefore be maximized.

Whereas the error of candidate solutions for the optimization problem and the tesselation problem could be measured exactly, this is not true for RL. The error depends on the reinforcement received for *every* possible (input, output)-tuple of the policy. The policy is a total function, so there are infinitely many possible inputs. This is why the introduction of a *test set* V is needed. It is a set of random points in the input space of the policy, and the error is measured only in those points.

DEFINITION 13 An ε-solution to an instance of a reinforcement learning problem

$$(\, n_{\text{in}} \in \mathbb{N}, n_{\text{out}} \in \mathbb{N}, V \subseteq \mathcal{L}(n_{\text{in}}), \mathcal{R} : \mathcal{L}(n_{\text{in}}) \to \mathcal{L}(n_{\text{out}} \to 1)\,)$$

is a policy $P \in \mathcal{L}(n_{\text{in}} \to n_{\text{out}})$ such that

$$\forall Q \in \mathcal{L}(n_{\text{in}} \to n_{\text{out}}) : \sum_{v \in V} [\mathcal{R}(v)](P(v)) \geq \sum_{v \in V} ([\mathcal{R}(v)](Q(v))) - \varepsilon.$$

The term "reinforcement learning" is usually meant to define a bigger class of problems than reinforcement learning with a specific model. Finding an appropriate model is considered part of the problem. The CandidateModel-function should however be defined before the general procedure of Adaptive Sampling can be started. So before that, the researcher has an extra important parameter to tune: choosing the model.

In real world curve fitting, the model should in most cases be a specific one (often polynomes of a certain order). In real world reinforcement learning problems, this is usually not the case. Although the distinction is quite arbitrary, the difference is kept visible here, to reflect the existence of such issues. So the choice of model is up to the researcher, instead of being predefined in each problem instance. Of course, choosing a model is not a simple question, and with the current state of science, it is still mainly a matter of intuition. Examples of models are normal distributions, neural networks, logical expressions, etcetera. Only after a specific model has been chosen, the reinforcement learning problem can be defined as a Cloud Fitting problem (by instantiating the CandidateModel) and can it be solved by a method from the Adaptive Sampling family.

Each of the k concrete objectives $o_i = v \bullet \mathrm{BinEnc}([\mathcal{R}(v)])$ (for $1 \leq i \leq k$) specifies an input vector $v \in \mathcal{L}(n_{\mathrm{in}})$, which is an element from V, and a corresponding reinforcement function $[\mathcal{R}(v)] \in \mathcal{L}(n_{\mathrm{out}} \to 1)$, which is encoded in binary with BinEnc (any function must be encoded into a vector before it can be stored in computer memory). The vector o obtained by the concatenation is converted to a (t, r, s)-cloud by the objective model:

$$ObjectiveModel(o) = (v_1, v_2, \ldots, v_{n_{\mathrm{in}}}, \underbrace{\sim, \sim, \ldots, \sim}_{n_{\mathrm{out}}}, [\mathrm{EXP}_{\mathcal{R}}])$$

such that for any $x = (x_1, x_2, \ldots, x_r)$, $[\mathrm{EXP}_{\mathcal{R}}] = [\mathcal{R}(v)](x)$ holds.

The candidate model should be chosen by the researcher, as mentioned above. However, it should decode a concrete candidate vector into an (r, s, t)-cloud with $r = n_{\mathrm{in}}$, $s = n_{\mathrm{out}}$ and $t = 1$. This last, fixed element should be 1, to indicate that the error is the distance between the reinforcement value and 1. We can show what happens then, with the concatenation operator $\bullet$ on tuples:

$$Vec(n_{\mathrm{in}} + n_{\mathrm{out}} + 1, (\mathcal{C}\ Observes\ \mathcal{O})) = Cloud(\mathcal{C}\ AppliedTo(\mathcal{O}\ AppliedTo\,\mathcal{C})) =$$

$$Cloud(\mathcal{C}\ AppliedTo(\mathcal{O})) =$$

$$\{(v_1, v_2, \ldots, v_{n_{\mathrm{in}}}) \bullet [\mathrm{CandidateModel}(c)](v) \bullet (1)\}$$

is compared with

$$Vec(n_{\mathrm{in}} + n_{\mathrm{out}} + 1, (\mathcal{O}\ Observes\ \mathcal{C})) = Cloud(\mathcal{O}\ AppliedTo(\mathcal{C}\ AppliedTo\,\mathcal{O})) =$$

$$Cloud(\mathcal{O}\ AppliedTo(\{(v_1, v_2, \ldots, v_{n_{\mathrm{in}}}) \bullet [\mathrm{CandidateModel}(c)](v) \bullet (1)\})) =$$

$$\{(v_1, v_2, \ldots, v_{n_{\mathrm{in}}}) \bullet [\mathrm{CandidateModel}(c)](v) \bullet [\mathcal{R}(v)]([\mathrm{CandidateModel}(c)](v))\},$$

to compute

$$\Phi(c, o) = |1 - [\mathcal{R}(v)]([\mathrm{CandidateModel}(c)](v))|.$$

This meets the specification if $\forall v, x : 0 \leq [\mathcal{R}(v)](x) \leq 1$.

Chapter 6

Adaptation

This chapter discusses the adaptation function α from the general procedure given in Chapter 2. It is a function that takes the old state, containing μ candidates, a list of samples, and a random number, and yields a new updated state.

6.1 Ingredients of α: Delete, Store, Explore.

The α function takes three arguments. This format was chosen carefully, because each argument corresponds to one of the three main issues in adaptation distinguished here:

1. State — Determining which parts of it to maintain (and thus which to *delete*),

2. Instance — Determining which new information to *store* (and thus filtering the useful information from the massive stream of samples),

3. Random — Determining what randomly chosen new directions to *explore* (and thus preventing early convergence and, with luck, provoking unexpected improvements).

6.1.1 Determining what to Delete

The candidates that have high errors, should in general be (partially) deleted to make room for new information. In Evolutionary Computation, this is called selection.

In Neural Computation, there is only one candidate, so totally deleting it would simply equal restarting the algorithm. Therefore, the candidate is deleted only partially. This does not necessarily mean that some of the m elements of the vector representing the candidate have to be changed more than others. The vector is only a representation of the candidate, and the part of a candidate that is changed does not have to be recognizable in it. Changing a candidate partially means that the new candidate resembles the old one, in one way or another. Back Propagation [BH69, HKP91] is a good example of a scheme for deciding where (in which weights) to store the new information from a sample.

The way in which a candidate solution is divided into different features, that are kept together, must be based on assumptions that one makes about the structure of the problem. If the error values are expected to depend on the presence of a specific type of features, these features should be kept together. For instance, Genetic Algorithms are sometimes said to assume the Building Block Hypothesis [Gol89, Mic96]. This hypothesis can be used to defend the use of crossover operators that keep substrings together. As another example, neural networks often use some form of gradient descent for adjusting the weights, which assumes a smooth change of errors with a change of weight values, for which a gradient exists.

Such assumptions are very important in the choice of an adaptation function, because if an adaptation mechanism is used that is based on a false assumption, the mechanism will simply not be appropriate. The analysis of a problem in terms of assumptions is often hard, but it is the only basis for choosing an adaptation mechanism, so it deserves careful examination. For 3-SAT, we assume that errors can be *decreased* by satisfying one individual clauses, and that errors will not dramatically *increase* if the valuation of one individual variable is changed.

6.1.2 Determining what to Store

If the sampling function Φ is easy to understand, the reason why a specific objective leads to a high error for a given candidate can be figured out. In that case, the adaptation function can replace the deleted information by new values that will probably reduce the error. For instance, in supervised learning, each objective represents an input and a corresponding desired output.

The extent to which the samples inform the adaptation function on the state, can differ. We distinguish the following three levels of informedness, each indicated with the question the information answers:

1. "How bad is it?": Only an error value is given.
2. "Which part is bad?": The error value specifically concerns a certain feature of a candidate.
3. "Why is it bad?": An adequate correction is also derivable from the sample.

Evolutionary algorithms operate on the first level of informedness: only an error value (a fitness value) is given, indicating how bad (or good) a candidate is. Sometimes, this is all the information available. In other cases, for instance when using an EA for classification or in fact 3-SAT, this is a deliberate choice.

The second level of informedness is used in reinforcement learning: a reinforcement signal does tell how good or bad a certain feature of the policy is (namely, its output for a specific input), but it does not tell why it is bad, so the samples do not tell how to repair this error.

If an objective can be interpreted as a simple target value, and the candidate model is understandable, we have the third level of informedness. We know how bad it is, we know what part is bad, and we know why it is bad. This is the level on which error back propagation for neural networks operates.

These three levels of informedness should not be confused with the distinction between unsupervised, reinforcement and supervised learning. These three types of learning differ in the problem format; the problem might contain a set of vectors, a reinforcement function or a target function. Although reinforcement learning problems have second level informedness, and supervised learning problems have third level informedness, the parallel does not hold for unsupervised learning. In the tesselation problem class we discussed in the sampling chapter the first level of informedness ("How bad is it?") is provided by the error value, that is defined as the Euclidean distance between an objective vector and the closest prototype. The second level is also available, because we know which prototype was closest, so that specific prototype is the bad part of the candidate, which is a *set* of prototypes. Moreover, third level informedness is present, because we know that moving the appropriate prototype towards the objective at hand will decrease the distance between the two, and thus the error.

This is what makes unsupervised learning very close to supervised learning within the Adaptive Sampling framework. The distinction between supervised (associative) and unsupervised (non-associative) that is made in Neural Computation refers to problem formats. Since we use clouds to represent any problem in the sampling function, the difference between learning a set of vectors and learning a function is almost trivial here. Both are represented as clouds, and the goal is simply finding another cloud that fits it.

6.1.3 Random Exploration

At the first and second level of informedness, no information can be extracted from the sample, to determine what to store. The sample only tells which candidates to delete (first level), or at most which parts of these candidates to delete (second level). Suppose the samples show that the evaluations of one or more candidates are suboptimal for one or more objectives, but it is not known what should be done to improve these evaluations. In that case, the only possibility is to delete the suggested parts of the current State (i.e., the suggested candidates or the suggested features of those candidates) and replace it with random values.

Apart from lack of information, there is another reason for using random exploration, even on the third level of informedness. It might be that there is information available on how to repair an error on a certain objective. However, using this information, will lead to the *suggested optimum*, which might not be the *global optimum*. To prevent early convergence to a local optimum, it can be useful to make a change that cannot be justified from the available information, but might unexpectedly lead to new possibilities.

6.2 How to Adapt, Given the Samples

Of course, an adaptation function should always be designed so that it tries to minimize the error values calculated by the sampling function. The three basic ingredients of adaptation described above (delete, store, explore), can be a rule of thumb in this design. Also, the three levels of informedness can be used to investigate what is possible and what is not.

We already discussed the name of the framework a little in previous chapters. Here, we will see more accurately what the core of Adaptive Sampling is. As mentioned earlier, sampling stands for solving a problem by isolating specific features and then generalizing over them. Whereas conventional computation often uses formal derivations to obtain global features of a problem, Adaptive Sampling uses a large amount of *specific* features and puts them together to let noise cancel out and obtain *global* information (which might for instance be a compromise or a maximum of all those specific features).

Gathering all those specific features is Sampling. Extracting the more global information is Adaptation. This global information can either concern a candidate (for instance a candidate compromising between different objectives), or an objective (for instance an optimum of a fitness function). This makes us also distinguish between two kinds of sampling, discussed in the following two sections. The first one is the extraction of isolated features from a candidate. The second one is the extraction of isolated features from objectives.

6.2.1 Sampling a candidate with objectives

If there are many objectives, it is often a trivial matter to come up with a candidate that has a lower error on *one* specific objective. By adjusting weights in that direction, the error on the specific objective will decrease. For the next objective, the weights will probably be changed in another direction. The combined effect of all these little changes will then hopefully decrease the overall error. This is an example of generalization over objectives: for every objective, a little change is made, and the combined effect will be a compromise for all the objectives. Isolating different features of a candidate iteratively is what we call "*sampling the candidate with objectives*". It is what makes Neural Computation a subclass of Adaptive Sampling.

6.2.2 Sampling an objective with candidates

In Evolutionary Computation, things are the other way around. It is easy to calculate the fitness function for one specific input. However, it is hard to understand it in such a way that a global optimum can be derived. Therefore, a huge number of candidates is "proposed", and evaluated. The evaluation of one candidate isolates one specific feature of the objective function. This is an example of generalization over candidates: every evaluation of a candidate has its influence on the population, and the combined effect will be a generalization over all these isolated features of the objective function. Isolating different features of an objective iteratively, is what we call "*sampling the objective with candidates*". It is what makes Evolutionary Computation a subclass of Adaptive Sampling.

6.2.3 Combining the two

If a method uses the first kind of sampling, this will result in at least second level informedness, because specific features of a candidate are examined. This means that Structural Error Assignment is possible, and if there is third level informedness too, even approximation of the optimum with some variation on Newton's method. The second kind of sampling does not automatically lead to second level informedness. The asymmetry here is caused by a simple circumstance:

Although both objectives and candidates can be *sampled*, only candidates can be *adapted*.

Therefore, when sampling an objective, adaptation should be towards more interesting features of this objective, and when sampling a candidate, adaptation should be towards a candidate with more interesting features. In both cases, "more interesting" is defined as "such that errors are minimized".

We can define a classification of different Adaptive Sampling methods, based on how they sample. We can use S_c to denote the average number of different candidates for which an objective is sampled in one cycle. Likewise, S_o denotes the number of different objectives for which a candidate is sampled in one cycle. Samples that occur more than once (for instance, if two candidates are exactly the same), count only once. Furthermore, if two or more samples are taken together without each sample having its individual effect (for instance if one fitness value is generated from several outputs of sampling function Φ), we count this fitness value as one sample. Roulette Wheel Selection can cause S_c to be smaller than μ, because individuals can occur more than once in the population. Evolutionary Algorithms always use $S_o = 1$, even if $k > 1$. Therefore, $S_c \leq \mu$ and $S_o \leq k$, but not necessarily $S_c = \mu$ or $S_o = k$. The definition of S_c and S_o might become more intuitional, as we see how they are used:

- Adaptive Sampling methods in general can be denoted with AS$[S_c, S_o]$.
- Evolutionary Algorithms are Adaptive Sampling methods that do not sample the candidates with objectives; they *do* use multiple candidates to sample the objective(s) in each step, so they can be denoted with AS$[S_c, 1]$.
- Neural Network methods are Adaptive Sampling methods that do not sample the objectives with candidates; they *do* use multiple objectives to sample the candidate(s) in each step, so they can be denoted with AS$[1, S_o]$.
- Hybridizations of Neural Network methods and Evolutionary Algorithms are Adaptive Sampling methods in AS$[S_c, S_o]$, that are neither in AS$[S_c, 1]$, nor in AS$[1, S_o]$.
- AS$[S_c, 1] \neq$ AS$[1, S_o]$, AS$[S_c, 1] \subseteq$ AS$[S_c, S_o]$ and AS$[1, S_o] \subseteq$ AS$[S_c, S_o]$.

In the Adaptive Sampling framework, there is room for both kinds of sampling. It should be noted that, in a way, the second kind of sampling is done by *any* Adaptive Sampling method. After each adaptation of the state, the candidate(s) have changed, so a different part of the search space is examined, that is influenced by different features of the problem instance. However, generalization over these features requires bookkeeping of the method's history. [1] There have been experiments with combining the two kinds into one method. For instance, neural network architectures can be evolved with an evolutionary algorithm, by testing each architecture on a simple problem in each step (see [SCE90] for an example). This is a combination of the two, where the training process of the neural network is included entirely in each cycle of the evolutionary adaptation. This is the way in which evolution and learning relate in nature, and maybe this

[1] TABU search [Glo89, RN95] uses such bookkeeping, and can be said to sample objectives with only one (changing) candidate.

setup can best be studied within Moshe Sipper's Philogeny-Ontogeny-Epigenesis model [SSM$^+$97]. This model defines a three dimensional space for classifying bio-inspired systems as points in this space. One axis represents the amount of *philogeny* (evolution). The *ontogeny* axis indicates whether physical hardware is improved within a system. The *epigenesis* axis indicates if this hardware learns after the physical development. This model includes physical adaptation, and classifies the rest in terms of "before ontogeny" (development of a species) and "after ontogeny" (development of an individual). In this context however, we are not interested in the role an adaptation process plays, but in how it works. It is just as well possible to invert the relation between neural and evolutionary processes (i.e., including an entire evolutionary process in one learning step).

The fact that the Adaptive Sampling framework puts EC and NC methods in one common notation, also makes it suitable for supporting the study of different kinds of hybridization and cross-fertilization. For instance, Lamarckian evolution is created easily by letting the adaptation function alternate between an α_{learn} and an α_{evolve}. We will refer to methods that are obtained from NC and EC by hybridization or by cross-fertilization as *mixed methods*. They are the third subfield of biocomputation that fits in the Adaptive Sampling framework. The following section shows how the Adaptive Sampling model can be useful for exploring possible EC-, NC- and mixed methods for 3-SAT.

6.3 Adaptation for 3-SAT

In previous chapters, two algorithms for 3-SAT were already described: one based on the tradition of Evolutionary Computation, and one in the spirit of Neural Computation.

The Adaptive Sampling framework can be used for classifying and comparing Evolutionary and Neural Methods. This was for instance done to design the neural 3-SAT method. Although the 3-SAT problem is not a learning problem in the canonical sense used in Neural Computation, the general problem format made it easy to derive a network function which *makes* it a learning problem.

In this section, we will show how Adaptive Sampling can be put to work to support the design of EC-methods, NC-methods and mixed methods, combining well-chosen features of both EC and NC. First, we will determine the level of informedness. They are described in Table 6.3.

Although we know that any change that does only one or two of the three sign-flips in the direction of the target will be enough to correct a wrong valuation, we can still call this direction the target direction. For real-valued representation, it makes sense to move all three variables toward this target over a certain distance ≤ 1. For Boolean representation, the only possible change to a variable is an entire flip to the opposite value. Admittedly, this makes the presence of an explicit target somewhat questionable. Unless we want to invert all three variables (which might be unnecessarily destructive), we will still have to choose which variable to change. This is why a Boolean representation of candidates makes the use of this third level informedness partially impossible, although it is present. However, for problem instances with $m \ggg 3$ variables, this effect might not be as severe as it looks, because the information from a specific sample still reduces the choice of which variable to change from m to 3, of course. Using this information is still classified as structural error assign-

Level of informedness

1. The first level of informedness ("How bad is it?") can be obtained from the samples generated with $\Phi_{3-\mathrm{SAT}}$, for instance by counting the number of unsatisfied clauses for a certain candidate.
2. The second level ("Which part is bad?") is also available, because the samples tell us exactly *which* clauses were unsatisfied for a certain candidate. Since each clause involves exactly 3 of the m variables that make up a candidate, we know that those 3 can be blamed for not satisfying that specific clause.
3. The third level is also present, which is best seen for the real-valued representation of the Neural method. Note that there are eight different ways to valuate three Boolean variables. Only one is wrong (i.e., makes the clause yield *false*), the other seven have at least one of the three signs right. Of these seven, there is one which has *all three* signs right. We take this valuation as the target value.

Table 6.1: Levels of Informedness for 3-SAT.

ment (delete+store) in this context, to distinguish it from random adaptation (delete+explore) used by Evolutionary Algorithms.

Since we have third level informedness, we can choose to use this information for defining the "Store"-ingredient, or to ingore it. If we ignore it, we must focus on the "Explore"-ingredient instead, because we must determine something to put in the empty spaces left by the "Delete"-ingredient.[2] However, this does not hold in the other direction: if we use the "Explore"-ingredient, this does not imply that we cannot use the "Store"-ingredient. So for third-level informed problems, there are three basic designs for the Delete-Store-Explore mechanism. If we use the "Store"-ingredient to change candidates in way that depends on the samples, then we have Structural Error Assignment — we only delete what is overwritten by what we store. If we use only the 'Explore'-ingredient, we treat the problem as if it had first-level informedness. We only determine how bad candidates are, and replace the bad ones by new ones, obtained from the good candidates with random exploration — in other words, selection. The options are shown in table 6.3.

Each option leads to one main subclass of Adaptive Sampling: the Evolutionary Computation approach, the Neural Computation approach, and the mixed approach (hybridizations), respectively. Besides these three options for the general setup, there is more to choose. Neural and Evolutionary Computation might benefit from collaboration not only by developing hybridizations

[2] Leaving out the "Delete"-ingredient is no option, because this would lead to an algorithm that does nothing at all.

Delete-Store-Explore

1. Delete + Store: Structural Error Assignment,
2. Delete + Explore: Selection,
3. Delete + Store + Explore: Mixtures.

Table 6.2: Options for the Ingredients of Adaptation.

Additional Bookkeeping

1. No additional bookkeeping,
2. Additional importance-measure per objective: SAW-ing,
3. Additional importance-measure per variable: real-valued representation.

Table 6.3: An extra choice out of three, created by cross-fertilization between NC an EC.

of the ideas from both, but also by importing ideas from each other (cross-fertilization). In the case of 3-SAT, solutions have been found for additional information storage. An additional mechanism that works for EA's might well work for the other approaches too.

Recall that the adaptation function of the SAW-ing EA from Chapter 3 uses additional bookkeeping per objective. It keeps one natural number per objective in a static data structure throughout the run, to keep track of how important the content of that objective is. This information can support the choice of which candidate is better than another, if they satisfy an equal number of clauses[3]. Likewise, the neural approach from Chapter 4 uses additional bookkeeping per variable. It keeps one real number per variable, instead of just a Boolean variable. The sign of this real number corresponds to the Boolean value of that variable in the valuation, and its absolute value keeps track of how important the content of that variable is. This information can support the choice of which variables should be changed, out of three variables involved in a clause.

The combination of those two extra mechanisms seems superfluous, and has not been tested (although this might deserve some further research). Leaving this option out, we have three possibilities for the additional bookkeeping, as shown in table 6.3.

Combining the three options for Delete-Store-Explore with the three options for additional bookkeeping, yields a total of nine suggested designs for the

[3] It is even possible that a candidate satisfying less clauses is selected over a candidate that satisfies more.

	Del.+Exp.	Del.+Sto.	Del.+Sto.+Exp.
plain	(G-SAT)	Simple SEA	Plain Ensemble
+ reals	(ES)	Neural Satisfaction	Neural Ensemble
+ SAW	SAW-ing EA	Lonesome SEA-SAW	Lamarkian SEA-SAW

Table 6.4: Nine method designs for 3-SAT, and names of methods of those designs.

adaptation function, as shown in table 6.4

For each design, a name of an algorithm with that design has been filled in. The implementations of the nine designs used here, are of course not unique. Other examples are also possible, and their performance might differ with other methods classified as being of the same design. However, the way in which these methods differ from each other can be described as differences in their design.

We have seen two of them already: The SAW-ing EA discussed in Chapter 3 is at the bottom left, and the Neural approach from Chapter 4 is in the center.

Methods between braces have not been investigated in the experiments, because they already existed before the SAW-ing EA, and have already proven less powerful (the SAW-ing EA is the best currently known incomplete 3-SAT algorithm [BEV96]).

G-SAT (see [GW95, BEV96]) is a local search technique that takes one valuation, and generates all its neighbors. A neighbor of valuation Val_1 is a valuation Val_2 such that there exists a unique i with $1 \leq i \leq m$ such that $\mathrm{Val}_1(x_i) \neq \mathrm{Val}_2(x_i)$. The neighbor that satisfies the highest number of clauses is chosen to generate a new set of neighbors from.

ES is an evolution strategy that uses a real-valued representation, and evaluates a candidate with a function of these real values. It adapts not only the valuations, but also the parameters of the adaptation mechanism. It was described in [BEV96], but showed not to perform as good as the SAW-ing EA. The other five methods are discussed in the rest of this chapter. Like “Neural Satisfaction”, they all use structural error assignment, as indicated with “SEA” in some of the names. Each of them was implemented with immediate instead of epoch-wise adaptation, and a randomized sampling-order, for the same reasons as mentioned in Section 4.3.

6.3.1 “Simple SEA”

The “Simple SEA”-method uses no selection, no real-valued representation and no SAW-ing mechanism. It maintains one candidate valuation c, and every sample $(c, o_i, \Phi(c, o_i))$ indicates the error on one specific clause o_i. If this error is $\Phi(c, o_i) = 1$, indicating non-satisfaction of that clause, one of the three variables involved in the clause (chosen at random) is inverted.

6.3.2 “Plain Ensemble”

The “Plain Ensemble”-method does the same as the Simple SEA method, but it deals with several valuations in parallel, with an additional selection scheme. The number of candidates μ is 6, 8, 10, 11 and 12, for problem sizes 20, 40, 60, 80 and 100, respectively. In each cycle, there is a chance for each valuation to be “killed”. This chance depends on the number of clauses it satisfies, so this implements a selection mechanism. The selection mechanism is based on incidental extinction: each time a candidate solution does not satisfy a clause, it has a chance $p_c = 0.0025$ to be replaced by a new vector, that is created by global uniform crossover: for each variable a parent is selected at random, and the new value is copied from that parent.

6.3.3 “Neural Ensemble”

Recall that the Neural Satisfaction method uses a real-valued representation, where 1 and -1 are used for *true* and *false*, and values in between are used to indicate uncertainty about the valuation of that variable. The values are strengthened (i.e., moved away from zero) for a satisfied clause, and the variables involved are weakened or inverted if a clause is not satisfied.

The “Neural Ensemble” is based on the Neural Satisfaction method, but with the same selection scheme added that distinguishes the Plain Ensemble from the Simple SEA method. The learning rule for penalty is also adapted a little. It is replaced by a restricted version of the MutOne-operator from the EC approach (Section 3.4): if a clause is not satisfied, exactly one of the three weights involved is inverted (i.e., multiplied by -1). Which one of the three weights is inverted is determined at random, like in the Plain SEA and the Plain Ensemble. Note that *only* weights that are involved in clauses that are unsatisfied, can be inverted by this scheme, whereas the EC approach did not make this distinction.

6.3.4 “Lonesome SEA-SAW”

The Lonesome SEA-SAW method is derived from the Plain SEA method, by adding a Stepwise Adaptation of Weights-mechanism. Because each clause has its own effect, and no overall fitness is calculated (there is only one candidate and no selection, hence the adverb “lonesome”), the SAW-ing mechanism cannot be directly copied from the SAW-ing EA. Instead, the following mechanism is used:

- Each 250 cycles, a SAW-ing tuple $Hard$ is generated, containing Boolean variables. Each of these corresponds to a clause, and is set to *false* if that particular clause is satisfied by the candidate valuation, and to *true* if it is not.

- For each sample, if the error is $\Phi(c, o_i) = 1$, indicating non-satisfaction of that clause, n_i of the three variables involved in the clause are inverted. The number of variables to invert is derived from the SAW-ing tuple: if clause i is listed as hard ($Hard[i] = \mathit{true}$), then $n_i = 3$, if not ($Hard[i] = \mathit{false}$), then $n_i = 1$.

6.3.5 “Lamarkian SEA-SAW”

“Lamarkian SEA-SAW” combines Structural Error Assignment, selection and SAW-ing. Its implementation is derived from the the Plain SEA (for the Structural Error Assignment) and the SAW-ing EA (for selection and SAW-ing). It starts with a random valuation. From this, μ neighbors are derived with the SAW-ing EA's MutOne-operator. Again, μ is 6, 8, 10, 11 and 12, for problem sizes 20, 40, 60, 80 and 100, respectively.

From this point, we *alternate* the Plain SEA and the SAW-ing EA:

1. Execute one cycle of the Plain SEA, for one of the candidates. The other candidates are not changed in this step.

2. Execute one cycle of the SAW-ing EA (selection of the best candidate and regeneration of a new set of neighbors). Either the candidate that was changed by the Plain SEA, or one of the others, might be selected to generate of a new set of neighboring valuations from.

3. If none of the candidates solve the expression, return to Step 1.

Chapter 7

Experiments

In the experiments, implementations of the seven methods from Section 6.3 were each run 1500 times in total. The test problem was 3-SAT, as described in Section 1.4. Five different problem sizes were used: 20, 40, 60, 80, and 100 variables, with a number of clauses that was 4.3 times the number of variables, i.e., 86, 172, 258, 344 and 430 respectively. For each problem size, 100 satisfiable problem instances were generated with the `mkcnf` formula generator from [vG93], with randomseeds generated by `mkcnf.sh` (with `start` = 1). Each method was tested three times on each of the 500 problem instances. The results are presented in this chapter.

7.1 Success Rates

Figure 7.1 shows the success rates. They were calculated by counting the number of instances for which a method reached a solution within 300,000 candidate evaluations. Four of the methods drop to a success rate of around 30% for larger problems. The three that have a better success rate are Neural Satisfaction, the Lamarkian SEA-SAW and the SAW-ing EA. On average, Neural Satisfaction's Success Rate is 0.008 above that of the Lamarckian SEA-SAW, and 0.04 above that of the SAW-ing EA.

7.2 Average Number of Evaluations to Solution

Figure 7.2 shows the average number of evaluations that a method does in one run, *if* it is successful in that run. The evaluations done in runs that did not reach a solution, were not counted. One evaluation always concerns one candidate, and all the objectives. So the number of evaluations can be calculated by counting the number of cycles in the general procedure, and multiplying this by μ, the number of candidates used. The same three methods that had a better success rate (the SAW-ing EA, Neural Satisfaction and the Lamarkian SEA-SAW), have an almost flat curve between problem sizes 60 and 100. The others have a very steep increase on that trajectory. On average, Neural Satisfaction needs about half the number of evaluations the Lamarkian SEA-SAW needs, and just over one third of the evaluations needed by the SAW-ing EA.

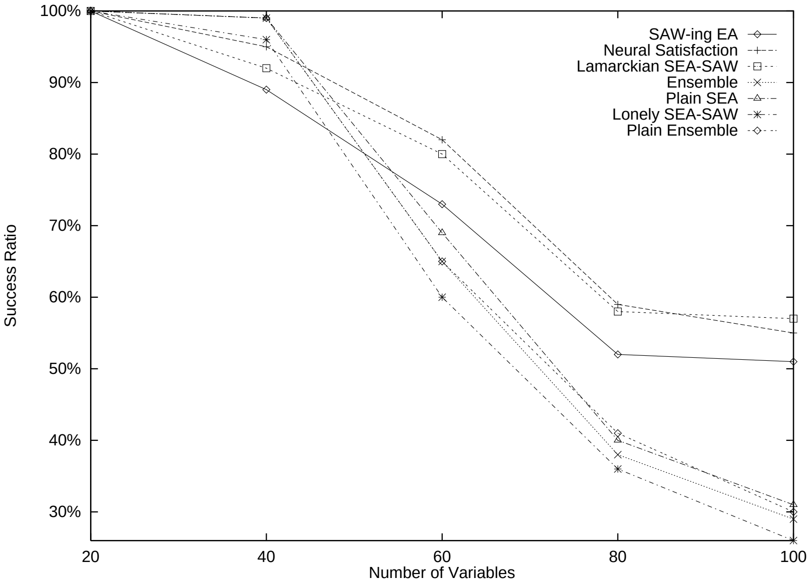

Figure 7.1: Success rates for the seven methods tested on 3-SAT Problems.

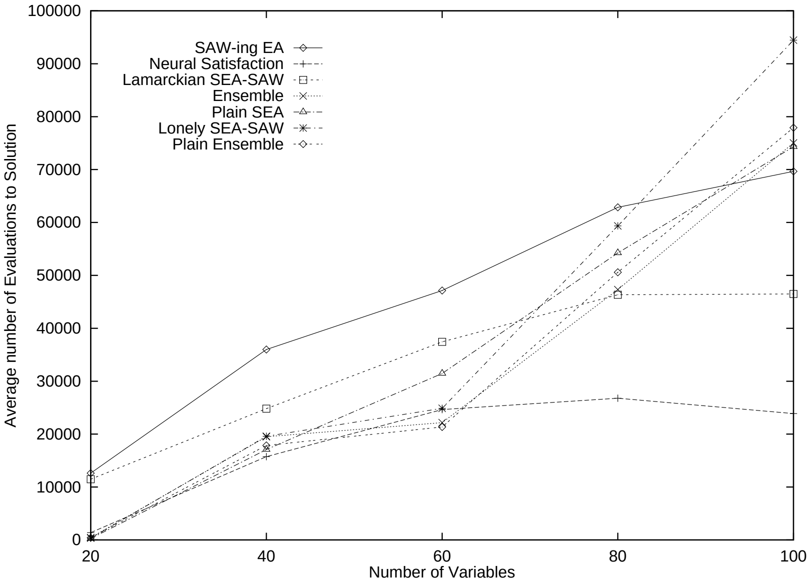

Figure 7.2: Average number of evaluations to reach a solution for an instance of 3-SAT, if one is reached.

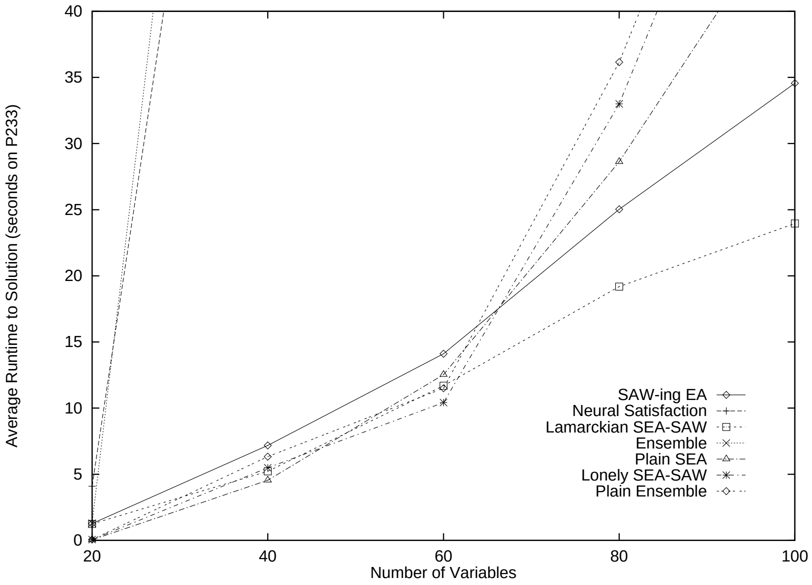

Figure 7.3: Average runtime to reach a solution for an instance of 3-SAT, if one is reached.

7.3 Average Time to Solution

Besides the average number of evaluations to reach a solution, we measured how much time the calculation took on a Pentium 233 computer. As we partly expected, this changed the picture entirely. When looking only at the success rate and the number of evaluations needed, Neural Satisfaction would be the winner.

There are real world problems, for which evaluations are very expensive, for instance if the sampling function Φ is implemented in a physical object. However, for 3-SAT evaluations are not more expensive than the other steps in the cycle. Therefore, it is fairer to compare the time it takes for the different methods on a standard computer.

Figure 7.3 shows that the real-valued representation used by Neural Satisfaction and Ensemble has a devastating effect on their time consumption, in comparison with the other five, that use a Boolean representation of candidates, with or without SAW-ing. Neural Satisfaction is out of the race now, and we may conclude that the Lamarkian SEA-SAW is the best one, of the methods tested. It has almost the same Success Rate as Neural Satisfaction, but it is much faster. The SAW-ing EA is second best, being about a factor 4/3 slower than the Lamarkian SEA-SAW, with a success rate that is about 0.032 lower than that of the Lamarkian SEA-SAW.

The use of the Adaptive Sampling model inspired a classification of nine designs, of which three had already been used before. It is of course hard to prove that the other six could not have been developed without it, but we conjecture that the use of the model was at least partially responsible for the development of the Lamarkian SEA-SAW, which showed to perform better than any other known method.

Chapter 8

Conclusions

8.1 Did we Succeed?

In the chapters so far, we have seen, among others, a general problem format, a general procedure format, a discussion of Evolutionary and Neural Computation, the cloud fitting problem class, with some of its relevant subclasses, and a discussion of different adaptation mechanisms and how to design them. In the last few pages, the usefulness of the Adaptive Sampling was tested, by applying different Adaptive Sampling methods to 3-SAT. In this last chapter, we will investigate whether the goals of this project were reached.

The Adaptive Sampling framework meets all three requirements we formulated. We will repeat them here:

INCLUSION REQUIREMENT
The framework should define a class of methods, including at least Evolutionary Algorithms and Neural Networks.

In Chapters 3 and 4, we saw how EC and NC fit in the framework.

STRICTNESS REQUIREMENT
The framework should define a relatively small family of algorithms, to assure that the common principles found are not trivial properties of just any algorithm.

The framework is certainly not as general as to include any algorithm. For instance, the `ProLog`-approach to 3-SAT that was mentioned in Section 1.4, cannot like the other methods we saw, be described in objectives, candidates, samples and adaptation.

LARGENESS REQUIREMENT
On the other hand, the framework should define a large enough family of algorithms, to assure that EC and NC are not simply represented separately, with the parameters being only a switch between the two.

The framework can be seen as describing something like a greatest common denominator of Neural and Evolutionary methods. Other methods, like G-SAT, which stems from a local search approach, or the five mixed methods, which can be classified in terms of hybridization and cross-fertilization between EC and NC, can also be described in the framework, which shows that the framework is not restricted to two distinct subclasses.

In Section 1.2, we formulated two additional goals, not directly involved in the development of the framework itself:

1. Exploring what new methods are suggested by the framework:
   If the framework takes away the epiphenomena from the description of bio-inspired methods, then this new, more fundamental description should open opportunities for applying the fundamental principles to new problem domains, by instantiating the framework in a way that is different from the two specific instantiations we find in nature.

2. Simulating different instantiations of the framework:
   Wherever possible, empirical results should support any theory. The 3-SAT problem will test the usefulness of the framework for supporting the design of better algorithms.

The first was done in the Chapter 4, when we saw how the principles of neural learning can be applied to (as far as we could find out) a new problem domain (constraint satisfaction), In Chapter 6 we saw that, although EC and NC have been combined before, the Adaptive Sampling framework can be useful to support the design of such combinations. They instantiate the common principles of learning and evolving in ways that are different from the two specific instantiations we find in nature (being Darwinian evolution and neural learning).

Beforehand, we stated that satisfying these requirements would test the central hypothesis:

CENTRAL HYPOTHESIS
Although Evolutionary and Neural Computation come from different origins and are used for solving different problems, the principles that make them work are the same, and the remaining differences can be explained as problem dependent parameters. Concentrating on these common principles, and not on epiphenomenal features of either one, facilitates the design of better algorithms.

We conclude that sampling and adaptation, in the way that they have been described in this thesis, can indeed be seen as these common principles that make EC and NC work. The remaining difference, Structural Error Assignment, can be seen as a problem dependent parameter: either it is or it is not possible to use SEA, and if it is possible, it depends on the problem whether SEA will lead to faster convergence to the global optimum (as desired) or to a local optimum (not desired). In the case of 3-SAT, the winning method turned out to be somewhere in between NC and EC. The framework, with its emphasis on the principles, and not on the epiphenomena, led to the design of this winning algorithm, as was predicted by the hypothesis.

Combining these results, we may say that both the central hypothesis of this thesis, and the Adaptive Sampling framework, were tested with a positive result.

8.2 Further Research

Further improvements of the “Lamarckian SEA-SAW” might be possible, for instance if a third component (besides structural error assignment and SAW-ing evolution) is added, from a complete 3-SAT algorithm. If this is done, it should also be possible to construct a complete method, that is only a bit slower than the “Lamarckian SEA-SAW” on the easier instances, but is still guaranteed to find a solution for the harder instances. For an improved incomplete variant, it might also be profitable to actively influence the sampling order, on the basis of relations between clauses: after adapting for clause i, there is an added importance for the clauses that have variables conflicting with clause i.

One mechanism that might be added as a third component would thus be some kind of “chain weaving”, which would be the following principle. Suppose variable x_i has just been changed from *false* to *true* by either an evolution step or a structural error assignment step. On the basis of this, a “chain” of adaptations can be started by picking out a clause that contains the literal $\overline{x_i}$, and inverting the valuation of one of the other two variables involved in this clause. The same can be done after *that* adaptation, etcetera, for instance for a predetermined number of times.

Also, some extra fine-tuning of parameters (e.g., further optimizing the number of candidates, or dividing candidates into separate evolutionary “tribes”), might result in some minor improvements. The proportions in which evolution and structural error assignment are mixed, can also be tuned (e.g., multiple evolution steps between each structural error assignment step, or vice versa).

The Adaptive Sampling framework can be enhanced to cover recurrent neural networks explicitly. Also, more advanced versions of Reinforcement Learning could be included in the Cloud Fitting problem class. Both would probably require the introduction of a recurrent notion of clouds: tuples that do not define sets of vectors, but sets of clouds. The Observes-operator should then probably be defined as the iterative closure of an (altered) AppliedTo-operator, instead of its current definition, in which the (current) AppliedTo-operator occurs only twice.

Earlier versions of this thesis used the fact that, in the general procedure, the instance and the state are sets of vectors (objectives and candidates, respectively). This makes that they can also be defined as clouds, and picking out one candidate or one objective can be defined as sampling. This takes place on a higher level than the sampling of one candidate or one objective. This part was taken out because it needed the concept of superposition (as used in the theory of quantum physics), which turned out to be too complex for this scope. However, it might be desirable for future versions of the model, to undo this simplification.

The “Lamarckian SEA-SAW” should be tested against other existing methods, to see if it really is the best one (here, we assumed that no other existing incomplete method performs than the “SAW-ing EA”). The algorithm will be submitted for the SATLIB-database[1], and additional publications on the Adaptive Sampling framework have already been planned.

[1] http://www.informatik.tu-darmstadt.de/AI/SATLIB.

Acknowledgements

The author would like to thank his supervisors Guszti Eiben and Walter Kosters, as well as Bram Bakker, Edwin de Jong, Dick de Ridder, and Paul den Dulk for their proof reading and fruitful discussions on this project. Furthermore, Irene Verweij and Martijn Meijering, as well as all the students of the PTT-IT program are gratefully acknowledged for their kind permission to use their computers.

Bibliography

[BBS91] A.G. Barto, S.H. Bradtke, and S.P. Singh. Real-time learning and control using asynchronous dynamic programming. Technical Report TR-91-57, University of Massachusetts Computer Science Department, Amherst, Massachusetts, 1991.

[BEV96] T. Bäck, A.E. Eiben, and M.E. Vink. A superior evolutionary algorithm for $3-\mathrm{SAT}$. In *Proceedings of the Ninth Dutch Conference on Artificial Intelligence (NAIC'97)*, pages 47–57, 1996.

[BH69] A.E. Bryson and Y.-C. Ho. *Applied Optimal Control.* Blaisdell, New York, 1969.

[CLR90] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms.* The MIT Press, Cambridge, Massachusetts, 9th edition, 1990.

[DB94] M. Dorigo and H. Bersini. A comparison of q-learning and classifier systems. In *Proceedings of From Animals to Animats, Third International Conference on Simulation of Adaptive Behavior (SAB94)*, Brighton, UK, 1994.

[EAvHN95] A.E. Eiben, E.H.L. Aarts, K.M. van Hee, and W.P.M. Nuijten. *A unifying approach to heuristic search*, pages 81 – 99. J.C. Baltzer AG, Science Publishers, 1995.

[EvdH97] A.E. Eiben and J.K. van der Hauw. Solving $3-\mathrm{SAT}$ with adaptive genetic algorithms. In *Proceedings of the 4th IEEE Conference on Evolutionary Computation*, pages 81–86. IEEE Press, 1997.

[Far90] J.D. Farmer. A rosetta stone for connectionism. In M. Mitchell, editor, *Emergent Computation (Physica D 42)*, pages 153 – 187, 1990.

[FOW66] L.J. Fogel, A.J. Owens, and M.J. Walsh. *Artificial Intelligence Through Simulated Evolution.* John Wiley, Chichester, UK, 1966.

[Fra96] J. Frank. Weighting for godot: Learning heuristics for GSAT. In *Proceedings of the AAAI*, 1996.

[Glo89] F. Glover. Tabu search: 1. *ORSA Journal on Computing, 1(3)*, pages 190–206, 1989.

[Gol89] D.E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley, Reading, MA, 1989.

[GW95] I. Gent and T. Walsh. Unsatisfied variables in local search. In J. Hallam, editor, *Hybrid Problems, Hybrid Solutions.* IOS Press, 1995.

[HKP91] J. Hertz, A. Krogh, and R.G. Palmer. *Introduction to the Theory of Neural Computation.* Addison-Wesley, Reading, 1st edition, 1991.

[Hol75] J.H. Holland. *Adaptation in Natural and Artificial Systems.* University of Michigan, Ann Arbor, 1975.

[Koz92] J.R. Koza. *Genetic Programming.* MIT Press, Cambridge, MA, 1992.

[LON97] Dan Lundh, Björn Olsson, and Ajit Narayanan, editors. *Biocomputation and Emergent Computation: Proceedings of BCEC97.* World Scientific, 1997.

[Mic96] Z. Michalewicz. *Genetic Algorithms + Data structures = Evolution programs.* Springer, Berlin, 3rd edition, 1996.

[Rad91] N.J. Radcliffe. Forma analysis and random respectful recombination. In *Proceedings of the fourth International Conference on Genetic Algorithms*, pages 222 – 229, San Mateo, CA, 1991. Morgan Kaufmann.

[Rec73] I. Rechenbach. *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution.* Frommann-Holzboog Verlag, Stuttgart, 1973.

[RN95] S. Russell and P. Norvig. *Artificial Intelligence a modern approach.* Prentice Hall, New Jersey, 1st edition, 1995.

[Rus79] B. Russell. *History of Western philosophy, and its connection with political and social circumstances from the earliest times to the present day.* Allen and Unwin, London, UK, 2nd edition, 1979.

[SCE90] J.D. Schaffer, R.A. Caruana, and L.J. Eshelman. Using genetic search to exploit the emergent behavior of neural networks. In M. Mitchell, editor, *Emergent Computation (Physica D 42)*, pages 244 – 248, 1990.

[SSM+97] M. Sipper, E. Sanchez, D. Mange, M. Tomassini, A. Pérez-Uribe, and A. Stauffer. The poe model of bio-inspired hardware systems: A short introduction. In J.R. Koza, K. Deb, M. Dorigo, D.B. Fogel, M.Garzon, H.Iba, and R.L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 501 – 511, San Francisco, CA, 1997. Morgan Kaufmann.

[Sut88] R.S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning, 3*, pages 9 – 44, 1988.

[vG93] A. van Gelder. Cnfgen formula generator. Available by anonymous ftp at ftp://dimacs.rutgers.edu/ pub/challenge/satisfiability/ contributed/UCSC/instances, 1993.

[Wat89] C.J. Watkins. *Models of Delayed Reinforcement Learning.* PhD thesis, Psychology Department, Cambridge University, Cambridge, UK, 1989.

Index