INDUCTIVE LEARNING BY SELECTION OF MINIMAL COMPLEXITY REPRESENTATIONS

by

Arlindo Manuel Limede de Oliveira

Memorandum No. UCB/ERL M94/97

14 December 1994

INDUCTIVE LEARNING BY SELECTION OF MINIMAL COMPLEXITY REPRESENTATIONS

by

Arlindo Manuel Limede de Oliveira

Memorandum No. UCB/ERL M94/97

14 December 1994

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Abstract

Inductive Learning by Selection
of Minimal Complexity Representations

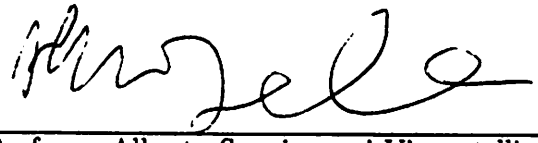
by

Arlindo Manuel Limede de Oliveira
Doctor of Philosophy in Engineering in Electrical Engineering and Computer Science

University of California at Berkeley

Professor Alberto Sangiovanni-Vincentelli, Chair

This dissertation addresses the problem of inferring accurate classification rules from examples. A formalization of Occam's razor, the minimum description length principle, is used to transform the problem of performing accurate induction from examples into the problem of selecting the minimal complexity rule that fits well the available data. Four different representation schemes are addressed: two-level threshold gate networks, multi-level Boolean networks, decision graphs and finite state machines. Heuristic algorithms for the inference of classification rules represented using each one of the first three representations are presented and their performance evaluated, both in terms of the size of the solution obtained and the quality of the induction performed. Exact algorithms are also proposed for the selection of minimal complexity classification rules represented either as decision graphs or as finite state machines. For these algorithms, proofs of optimality and an evaluation of their limitations are also presented. The generalization accuracy of the classifiers generated using the algorithms proposed is compared with the accuracy of alternative approaches in a variety of problems extracted from the machine learning literature. Finally, the applicability of the algorithms to real-world tasks is demonstrated with two large problems: hand-written character recognition and noise removal from gray scale images.



Professor Alberto Sangiovanni-Vincentelli
Dissertation Committee Chair

Contents

List of Figures

List of Tables

Acknowledgements

Many persons made it possible for me to conduct this research and write this dissertation and I will surely forget to mention some of them. The contributions of some others were, however, so important, that not even a disorganized person like me could possibly let them pass without a public acknowledgement.

First, I'd like to thank my wife Irene who left family, friends and a comfortable life in Portugal to come with me to Berkeley. This work would not have taken place without her continuous support and love and I cannot thank her enough. My daughter, Ana, also helped me, although unwittingly, to gather the strength needed to put together this dissertation while I was away from her. My parents, who always believed more in me than I did myself and supported me in the decision of coming to this faraway land also had a major influence on my decision to come here. Such a decision would probably never happened, however, were it not for Prof. Luís Vidigal advice, who convinced me that life would be more interesting as a student in Berkeley than as a new engineer in Portugal. He turned out to be right in all counts. I met in Berkeley and the Bay Area so many interesting people that these years will probably be remembered as the best and most interesting ones of my life.

Naturally, my advisor, Alberto Sangiovanni-Vincentelli, who supported me from my very beginning as a graduate student in Berkeley deserves special thanks. Not only he suggested the original idea that eventually led to this work, but he was always forthcoming with good advice and interesting directions to explore. The constant and apparently endless source of energy that he draws upon to handle the myriad of commitments he has while still making time for his students will always be an inspiration to me.

An advisor does not not an environment make, however, and many other persons contributed to make the life of a graduate student in the CAD group a very interesting one. I'd like to thank in a special way Prof. Brayton, who not only agreed to be in the thesis committee but will always be a reference to me by the way he advises his students and collaborators. I'd also like to thank Prof. Donald Glaser who also took the time to read this dissertation and give me valuable feedback.

In the many years I spent in Berkeley and my office in 550 Cory I had the privilege to meet many other students and office mates. With all of them, but specially with Rajeev Murgai, Greg Sorkin, Alan Kramer, Mark Beardslee, Brian Lee, Adnan Aziz, Andrea Casotto, Sriram Krishnan, Chuck Kring, Felice Balarin, Wendell Baker and Desmond

Kirkpatrick I had many interesting conversations that were almost always illuminating but always, without exception, entertaining.

Off all the other persons I met in the CAD group I'd like to thank specially Tiziano Villa, from whom I could always get that reference I could not find anywhere else and was always forthcoming with a more positive opinion about my research than I could master from myself. I also thank Timothy Kam for being an excellent office mate and for helping me set up the comparisons with implicit algorithms for the minimization of finite state machines and Stephen Edwards, who not only cooperated with me in the work described in chapter 6 but taught me many useful things about Unix, TEX, LATEX and computers, in general. I'd also like to thank Timothy Ross and the Pattern Theory Group at the Air Force Wright Laboratory for their willingness to try some of my algorithms in their benchmark set and making available the data from their experiments. Last, but not least, I'd like to thank Vigyan Singhal who was kind enough to suffer some inconvenience in letting me share his apartment during the last period of my stay in Berkeley.

Outside the CAD group, many people contributed to make my stay in the Bay Area a memorable one. My friends, Henrique and Isabel, made California feel very much like our home away from home by being our *de facto* close family. Clara and Derek were not only good friends but also a constant source of support in the harder times I spent alone in the beginning and end of the period I stayed here.

Finally, to all my other good friends in the Bay Area, Miguel and Guida, Jorge Nuno and Raquel, Miguel Villas Boas, Mário and Paula, Kimiko, Mike and Kathy, my sincere thanks for making this stay in the Bay Area a most enjoyable experience.

In the course of these years, several institutions supported this research, either directly or indirectly. Instituto Superior Técnico allowed me an extended leave to perform this work. The Portuguese Fulbright Commission and Portuguese Invotan Committee as well as the Joint Services Electronics Program under Contract Number F49620-94-C-0038 all supported, in part, the work that led to this dissertation. Their support is gratefully acknowledged.

Chapter 1

Introduction

The main objective of the discipline known as artificial intelligence (AI) is to make computers behave in ways that can be defined as intelligent. One of the hallmarks of intelligence is the ability to learn from past experience and to adapt the behavior in accordance with this experience. Machine learning is the branch of AI that is concerned with the ability of systems to learn and adapt. In fact, this branch became so important, and the techniques developed have found applications in such a wide variety of fields that machine learning is now a very important discipline on its own right.

One of the central topics of research in machine learning is inductive inference, the study of algorithms that enable systems to learn from examples. This subject is also the central topic of this dissertation. In particular, this thesis describes algorithms for the inference of classification rules in discrete domains. The exposition made in the following sections intends to introduce the basic concepts and definitions involved in inductive inference but does not pretend to be exhaustive or even complete. The reader is referred to [43] and [64] for comprehensive reviews of both the empirical and theoretical issues involved in inductive inference.

1.1 Inductive Inference

Inductive inference problems are characterized by a *domain* D and a *learner*, or learning algorithm. The learner is able to observe some objects (or instances) in the domain. This set of objects is the *training set*, T. Given this information, the learner has to infer an *hypothesis*, i.e., a theory that can be used to explain the data observed. The objective is to

use the generated hypothesis to predict the characteristics of a set of previously unobserved objects, the *test set*. In general, some information is missing from the objects in the test set. The objective is to predict this information using the generated hypothesis. Although hypotheses may be used to predict a number of different characteristics, this thesis addresses only a particular family of problems, globally known as classification problems. In classification problems, each object in the domain is labeled with one label that defines the class it belongs to. The hypotheses generated by the learner will be used to recover these labels for the objects in the test set. A system that uses the hypothesis generated by the learner to classify objects is called a *classifier*. In single class problems, the instances are labeled as either belonging or not belonging to a given class. In multi-class problems, the instances can belong to one of several classes.

1.1.1 Motivation and Examples

Examples from two concrete domains will be used to motivate the reader for the wide range of applicability of inductive inference algorithms and illustrate in a more vivid way some of the concepts involved. The first domain is related with a particular ending in the game of chess. Consider the chess endings depicted in figure 1.1. If both players

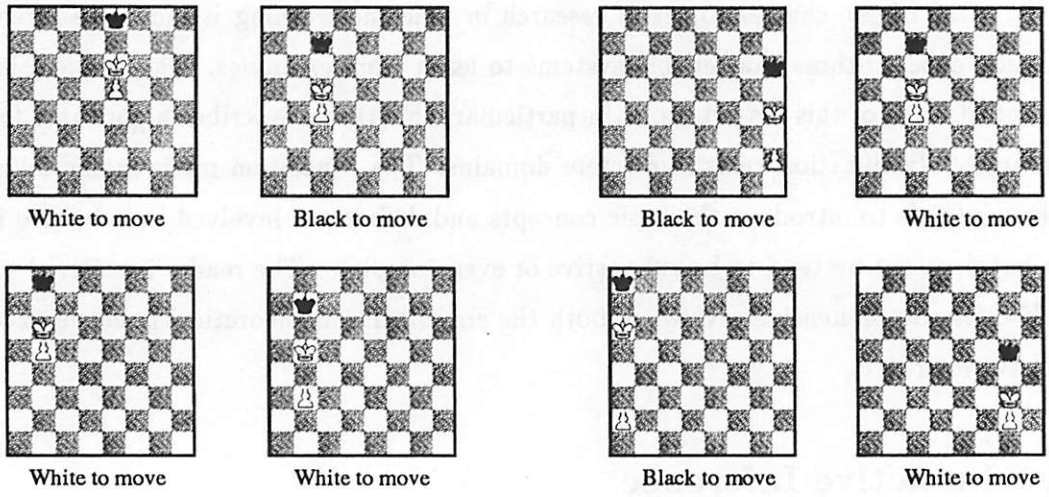


Figure 1.1: Instances of the *King+Pawn vs. King* chess endings

play the best moves available, the 4 positions on the left side will terminate with a win for

white, while the positions on the right side will terminate with a draw. This is a single-class classification problem: a position is either a win, or is not. The objective of the learning algorithm is to infer, from the training set data, a rule that can be used to decide whether, under perfect play, a given position will be a win or a draw.

In this example, the domain is the set of all King+Pawn vs. King chess endings that have all chess pieces in the same column and have the Kings in front of the pawn and in direct opposition, i.e., separated by one single row. The objective of the learner is to derive a classification rule that can be used to find whether or not a particular position has a forced win for white. Such a classification rule will be termed an hypothesis. The hypothesis can also be viewed as a representation of the set of all positions that are forced wins for white. This set can be represented implicitly by its characteristic function[1]. This view of the hypothesis as subsets of the input space (or the respective characteristic functions) is very useful and will be defined formally in chapter 2.

Any classification problem can, in principle, be formalized as an induction problem in discrete spaces by appropriately encoding the instances using an encoding scheme that maps the problem into a discrete domain. In this case, such an encoding of the problem is easy to obtain. For example, instances of this problem can be encoded by simply writing down the column where the pieces are, the row of each piece and an extra bit that describes which side is to move. If the row coordinates are listed in the order *White King, White Pawn, Black King* and both coordinates are between 0 and 7, the training set would then consist of the following description:

```
4,5,4,7,1,+
2,4,4,6,0,+
7,3,1,5,0,-
2,4,3,6,1,-
1,5,4,7,1,+
1,4,2,6,1,+
0,5,1,7,0,-
6,2,1,4,1,-
```

Figure 1.2: An encoding of the *King+Pawn vs. King* chess endings

[1]The characteristic function of a set is a function that takes the value 1 for the elements on the set and takes the value 0 otherwise.

The + signal describes positions where White has a forced win. It must be observed that almost all domain specific information was lost in the conversion performed using this encoding. If the original problem was solvable, at least for someone familiar with the domain, the encoded version of the problem looks much harder. The learning algorithms described in this thesis are usually faced with learning the problems without having the possibility of using any contextual information that is available for humans. Choosing an encoding scheme that preserves as much contextual information as possible is a critical step in the solution of any problem. The process by which an appropriate encoding scheme can be chosen is domain dependent and is not addressed in this dissertation, except in special cases for illustrative purposes. It will be assumed that a description in a form similar to the one described above was obtained in some way and will be used as the input to the learning algorithm. Naturally, the algorithms can be used even if the chosen representation does not preserve any contextual information, but the quality of the generalization obtained will suffer. Given the encoding described above, figure 1.3 shows a graph that can be used to classify this type of chess endings. It is easy to verify that a classification procedure that uses this graph classifies correctly all the positions in figure 1.1. In fact, it will also classify correctly all the possible positions in this domain.

This work describes algorithms that can be used to derive representations like the one depicted in figure 1.3. In this particular case, it is unlikely that any learning algorithm would be able to infer the exact solution with the limited amount of information provided in the 8 positions shown. Some of the algorithms described are, however, remarkably effective in this problem if a somewhat larger training set is provided. This problem is used as one of the case studies in chapter 7.

The second domain used to motivate this work is in the field of handwritten character recognition. Figure 1.4 shows instances of patterns that correspond to different handwritten digits extracted from the widely available NIST database [28]. The objective in this case is to infer a classification rule that allows a classifier system to recognize which particular digit the writer intended to write.

This problem is an example of a multi-class problem because the number of different labels is higher than 2. This problem is also used as a case study in chapter 7, where the strategies used to encode the instances and select the representation are also described.

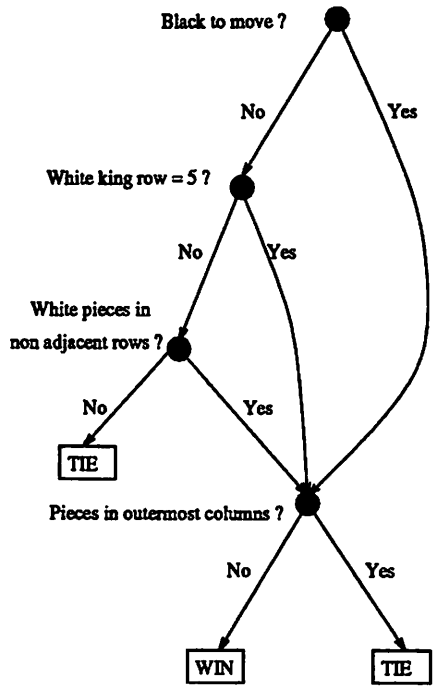


Figure 1.3: A decision graph for the chess endings problem

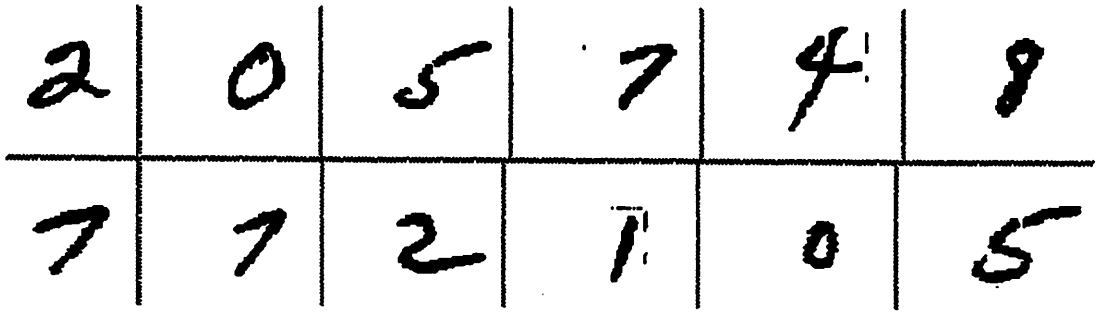


Figure 1.4: Examples of handwritten digits

1.1.2 The Effect of Noise in Classification Problems

In the chess example, it was assumed that all positions present in the training set are correctly labeled. This is the noise-free case, where no corruption of the attribute values or the class labels is assumed to take place.

The character recognition example, on the other hand, is bound to have some errors present in the data used as the training set. This complicates the task of the learner that has to take into consideration the fact that some of the labels or attribute values may be wrong and change its hypothesis accordingly.

Algorithms that can handle problems with classification and/or attribute noise are, in principle, more general and can also handle the noise-free case. Their performance may, however, be inferior to that of algorithms that use the fact that no noise is present in the training set. In real world problems, some amount of noise is always present. Large databases always have some fraction of the data corrupted by noise or incorrect. Therefore, algorithms that are designed to be applied to real world problems need to take into consideration this case. Nonetheless, many formal models are only applicable to the noise free case. In some cases, this restriction is only present to make the analysis of learning situations simpler. In others, it may be a fundamental limitation of the model or the algorithms addressed. Moreover, it may happen that even algorithms designed for the simple noise-free case turn out to work quite well in problems with noise, specially if the level of noise is not high.

1.1.3 Mapping Multi-Valued Attributes to Boolean Variables

The majority of the algorithms described in this dissertation assume that problems defined over discrete spaces with multi-valued attributes are first mapped into a Boolean space of N variables, $\{x_1, \ldots x_N\}$. This implies mapping attributes that can take $k > 2$ values into a set of $\lceil \log_2(k) \rceil$ binary variables. This solution is by no means unique, since all the algorithms described can be modified to handle multi-valued attributes. There are two major drawbacks to this solution:

- Concepts may become harder to represent in the modified Boolean space than they are in the original multi-valued space. This increase in complexity is, however, limited by a polynomial factor on the number of values that the original attributes can take and is therefore not a very important consideration. The quality of the inference performed

by the algorithms may, however, be reduced by this transformation. Comparisons performed with alternative approaches that use the original multi-valued attributes did not, however, show any significant degradation in the quality of the solutions observed.

- If the final solution needs to be presented to a human (instead of simply used to classify new instances), it may be harder to interpret because of the binary coding of multi-valued variables. Since the transformation from discrete valued attributes to Boolean valued ones is performed internally, it is completely transparent to the user and this problem is significant only if an easy human interpretation of the derived hypothesis is critical. This happens only in a relatively small number of applications. Furthermore, a simple post-processing step can be used to recover an hypothesis formulated in terms of the original variables if clarity of the derived rule is a critical factor.

For many problems, these drawbacks are a relatively small inconvenience compared with the advantages that come from being able to uniformly handle all the variables. This approach also provides additional generalization capability provided by the ability of the algorithm to use the binary representation of integers to its advantage in the identification of regularities.

1.1.4 Transforming Multi-Class Problems Into Single-Class Problems

General classification problems can be transformed into single class problems using a variety of encodings. Assume that the instances in the training set belong to one of k possible classes. The following alternatives can be used to encode the value of the label that describes the class of an instance:

- **Binary code:** Encode the label value using a binary code with $\lceil \log_2 k \rceil$ bits. This option generates the smallest number of single class problems from the original classification problem. However, with this encoding, an error in one of the single class problems usually causes a mis-classification.

- **One-hot code:** Encode the label value using a one-hot encoding that uses k bits. This encoding sets to one the bit that corresponds to the particular class. An error in one of the resulting single class problems will transform the code into a non-existing

one and a class value can not be attributed to that instance, but will not result in a mis-classification.

- **Error correcting code:** Encode the label value using an error correcting code [23, 73]. This approach preserves most of the compactness of a binary encoding while being much less sensitive to errors in one of the single class problems. Additionally, the Hamming distance between the observed outputs and the closest valid codeword gives a measure of the certainty of the classification. This can be useful in problems where a failure to classify is less serious than the output of a wrong classification.

It should be noted that the use of the first or third alternatives creates single class problems that are, in general, more complex, than the ones generated by the second alternative. Nonetheless, some classification errors made using the last encoding are recoverable, given the properties of error correcting codes.

After both the mappings defined above are performed, the solution of any classification problem is equivalent to the synthesis of a number of single output Boolean functions and a discussion on the power of any hypothesis representation scheme reduces to a discussion on the power of that representation scheme to represent Boolean functions.

1.2 Overview of the Approach

The approach described in this dissertation is based on the fact that simpler hypothesis usually perform better in the training set than complex ones. The theoretical justifications for this assumption are addressed in chapter 2. In particular, this work is concerned with the design of algorithms that generate hypothesis of minimal complexity that are consistent, to some degree, with the training set labeling. In all cases, the algorithms accept as input a labeled training set and generate an hypothesis of minimal complexity described using one of the representation schemes described later in this section.

Although the classification rules (or hypotheses) generated by the learner can be described using any representation scheme, for problems defined over discrete spaces, some representations are particularly natural.

This dissertation addresses 3 different representations for hypotheses in problems defined by a fixed number of discrete attributes: two-level threshold gate networks, multi-level Boolean networks and reduced ordered decision graphs. It also describes algorithms

for the synthesis of deterministic finite state machines, a representation appropriate for problems defined by attribute sequences of variable length. This section provides an informal definition of these representations. When required, more formal definitions will be introduced in the chapter dedicated to each one of these representations.

1.2.1 Two-Level Threshold Gate Networks

A threshold gate is a gate with k inputs, $\{x_1, x_2, \ldots, x_k\}$ that outputs the logic value 1 if and only if

$$\sum_{i=1}^{n} w_i x_i \geq w_0 \tag{1.1}$$

where $w_1, w_2, \ldots, w_k$ are (integer) input weights and w_0 is the (integer) threshold value of the gate.

A two level threshold gate network consists of two levels of gates, where the input variables are connected to the inputs of the gates in the first level and the outputs of these gates are connected to the inputs of the second level gates.

1.2.2 Multi-Level Boolean Networks

A Boolean network is a directed acyclic graph where each node implements a simple single-output primitive Boolean function, i.e., an *and*, *or* or *not* operation. In this graph, each node with no incoming edges corresponds to one of the input variables. Some special nodes in the graph are defined as the outputs of the network. For a single output Boolean network, there is a single output node and this node has no outgoing edges.

1.2.3 Reduced Ordered Decision Graphs

A decision graph is a rooted, directed, acyclic graph where each node is labeled with the name of one variable. A decision graph has two terminal nodes n_z and n_o that correspond to the leaves of the graph. Every non-terminal node n_i has one *else* and one *then* edge that point to the children nodes, n_i^{else} and n_i^{then}, respectively.

A decision graph is called read-once if each variable occurs at most once along any computation path. All decision graphs considered in this work are read-once decision graphs and references to this will be omitted. A decision graph is called ordered if there is an ordering of the variables such that, for all paths in the decision graph, the variables are

always tested in that order (possibly skipping some variables). A decision graph is called reduced if no two nodes are equal (same label and same descendents) and no node has the *else* and *then* edges pointing to the same node [18]. A decision graph that is both reduced and ordered is called a reduced ordered decision graph (RODG).

Reduced ordered decision graphs are known in the logic synthesis community as Boolean decision diagrams (BDDs). Both notations are widespread in the different communities and one of them had to be chosen for the present text. I choose to use the term RODG to denote this type of representation. Readers more familiar with the BDD notation should read BDD every time the term RODG is used.

1.2.4 Deterministic Finite State Machines

Following the standard conventions, a finite state machine (FSM) is represented by a directed graph where each edge is labeled with the value of one input and the corresponding output. Each node in this graph corresponds to one state of the machine. One of the nodes is distinguished as the reset state and represents the state where the machine is started. In the presence of one input, the machine outputs the value present in the edge that leaves the current state and is labeled with that value of the input and the current state changes to the one pointed by that edge. A finite state machine is deterministic if, for any node in the graph, only one outgoing edge with a given input label exists.

1.3 Expressive Power of Different Representations

The hypotheses generated by the learning algorithms are described using one of the representations described above. The representation selected can have a critical impact on the quality of the accuracy of the generalization obtained.

In particular, if the representation used is not well adapted to the problem at hand, the complexity of any hypothesis that matches the training set will be too large and its accuracy poor. The critical point is that the representation scheme used has to be powerful enough to allow for hypothesis with small complexity to match the training set data to some degree of accuracy. Assume, for example, that the family of problems under consideration is the family of symmetric Boolean functions defined over n variables. A function is called symmetric if its value doesn't change when two input variables are exchanged. This implies that the value of a symmetric function depends only on the number of bits at 1. Assume

now that the learning algorithm represents the hypotheses using a disjoint normal form[2] (DNF) representation. Since some symmetric functions require a number of terms in the DNF representation that is exponential[3] in the number of input variables the performance of this algorithm for some problems in this family will be very poor.

The relations between the sets of Boolean functions that can be represented by polynomial sized descriptions for each one of these representations is represented in figure 1.5.

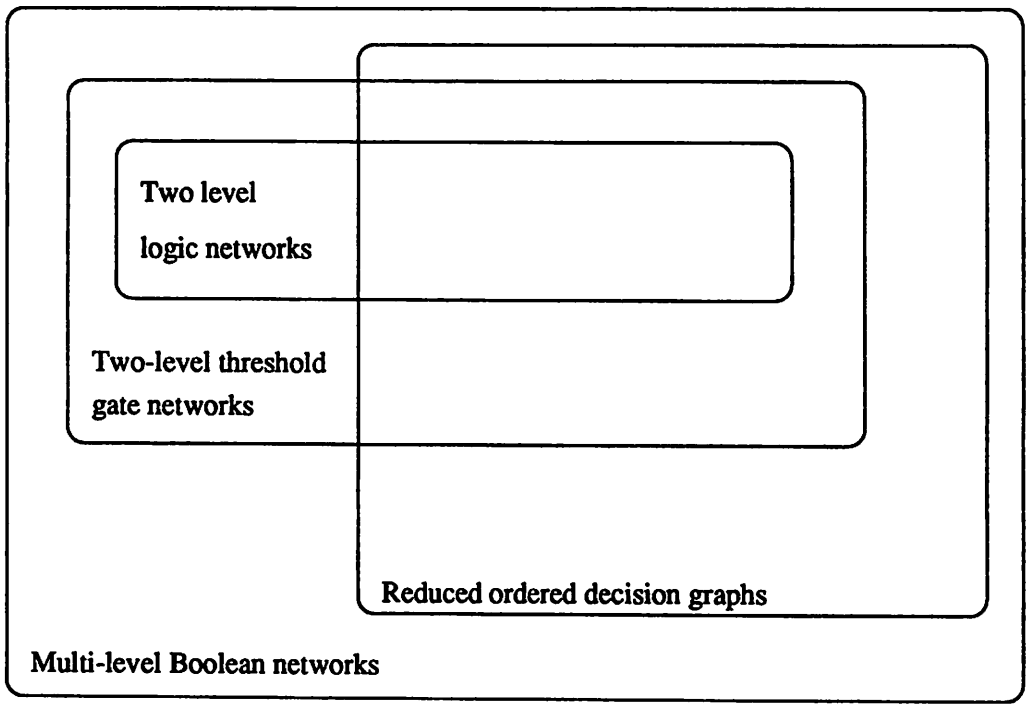


Figure 1.5: The expressive power of different representations

A particular representation scheme X is more powerful than representation scheme Y if any concept that can be represented by a polynomial sized description using scheme Y can also be represented using a polynomial sized description using scheme X. In this sense, multi-level Boolean networks are more powerful than any one of the other 3 representations and two-level threshold gate networks are strictly more powerful than two-level Boolean gate networks. It is also important to note that the class of functions implementable by polynomial size unbounded level threshold gate networks is equal to the class of functions

[2]Equivalent to a sum-of-products representation.
[3]For example, the parity function requires 2^{n-1} factors in DNF form.

implementable by unbounded level Boolean gate networks. This equality comes from the fact that one can always replace a threshold gate by a polynomial number of Boolean gates.

Regrettably, the same flexibility that makes multi-level Boolean networks so powerful also makes it very hard to design algorithms that are effective in deriving a compact multi-level network that matches the training set to some specified degree of accuracy. On the other hand, the properties inherent to the reduced ordered decision graphs representation makes it possible to design and implement algorithms that perform this task in a more effective way.

1.4 Alternative Approaches

The representations addressed in this dissertation represent only some of the possible choices. Many other representations are possible and the following two deserve a special reference because of their popularity.

1.4.1 Neural Networks

Neural networks have emerged as one of the representations of choice not only in the machine learning community but also in many other fields. Although there are many different types of neural networks, the models most commonly used can be viewed as interconnected networks of simple processing elements. Each processing element computes a weighted sum of its inputs and outputs a continuous function of this sum. Among all algorithms that can be used to select the node functions, the back-propagation [41] algorithm is the most popular one. This approach is remarkably effective in a wide variety of problems and the algorithms used are straightforward. However, there are some drawbacks that make the use of neural networks trained using back-propagation harder to use than alternative techniques, the most relevant being probably the difficulties inherent to the process of choosing the right architecture. Solutions proposed to this limitation have not met widespread acceptance. Furthermore, in some problems, being able to realize an hardware implementation is important. Neural networks trained using standard algorithms are hard to implement because each node in the network implements a soft threshold function with real weights. Since these weights have to be stored with a relatively high precision, either analog weight storage or expensive D/A converters have to be used. In contrast, the three last representations addressed in this dissertation are straightforward to implement using

standard digital technology, an alternative that is more cost effective than solutions based on analog designs. One of the representations addressed, two-level networks of threshold gates, can be viewed as a special case of neural networks, where only two levels are allowed, the weights are integer valued and the output is a step function. This limitations make it both easier to implement this type of networks in hardware (although not as easy as the other three representations) and to design minimal size networks. A more detailed comparison between the two approaches is presented in chapter 3.

1.4.2 Decision Trees

Decision trees [17, 76, 79], on the other hand, do not suffer from these drawbacks. Little or no extra information is required apart from the data present in the training set and the resulting tree can be easily implemented using standard digital technology. Empirical results [6] have shown that the performance of decision trees is similar to that of trained neural networks for many problems. There are, however, some domains where the algorithms described in this dissertation perform much better than decision trees. In fact, one of the representations studied, RODGs, can be viewed as a generalization of decision trees. Decision trees and RODGs are, however, less well adapted than neural networks to perform induction in domains that are inherently continuous.

1.5 Organization of the Dissertation

The remaining of this dissertation is organized as follows:

Chapter 2 describes the theoretical results that support the selection of minimal complexity hypotheses as the most likely to exhibit high generalization accuracy. In particular, it describes how the minimum description length principle can be applied to transform the induction problem into an optimization problem. This chapter also analyzes the computational complexity issues involved in the process of hypothesis selection and introduces some basic definitions that are used in the succeeding chapters.

The following four chapters are dedicated to the description of the algorithms that select the minimal complexity hypothesis for each one of the representations described in section 1.2.

Chapter 3 describes an algorithm for the selection of minimal two-level networks of threshold gates. The algorithm is based on extensions of some of the well known concepts

and techniques developed for two-level circuit minimization and its applicability is restricted to noise-free problems.

Chapter 4 describes and analyzes the algorithms developed for the synthesis of minimal multi-level networks of Boolean gates. Apart from the different representation chosen, these algorithms differ from the ones described in the previous chapter in that they can handle problems that have some level of noise.

Chapter 5 is dedicated to the study of RODG representations. It has two major parts that address the same problem with radically different approaches. The first part describes how an exact solution for the problem can be formulated as a set covering task. This is the first formulation of an exact solution to this problem that does not involve an extensive search algorithm. Since the exact approach is limited to relatively small problems, the second part of this chapter is dedicated to the study of an heuristic algorithm that can be used in much larger problems. The exact solution is restricted to the noise free case while the heuristic approach can handle the more general problem of classification in the presence of noise.

Chapter 6 describes an exact implicit algorithm for the inference of finite state machines from examples of accepted and rejected strings. This implicit formulation is a radically different approach to this problem and departs completely from the explicit search based methods developed by previous authors.

The discussion of the previous work developed using each of these representations has relatively little overlap and is relegated to the beginning of each of these chapters.

Evaluating such a wide variety of algorithms in a meaningful way is a complex task and many interesting comparisons had to be left out due to space and time considerations. Since the inductive inference task is formulated as an optimization problem using a variety of representations, the solutions have to be evaluated with respect to two different criteria: how well do the algorithms solve the optimization problem proposed and how does the inference performed by the algorithms compare with alternative techniques.

An evaluation of the algorithms with respect to the first criterion is described immediately after each algorithm is presented. For exact algorithms, the critical parameter to measure its their speed, or, equivalently, how large are the problems that they can solve in a fixed amount of time. For heuristic algorithms, both speed and solution quality (i.e., how close to the optimum are the solutions obtained) are important. Results listing the data obtained in problems with known solutions are included at the end of each of these

chapters.

Evaluating the algorithms with respect to the second criterion is more difficult because one has to chose from among the alternative techniques the ones that should be used as a standard of comparison. In the interest of fairness, the algorithms are compared with the most popular techniques used for inference from examples and these comparisons are made in a wide variety of problems. This is important because an algorithm may perform particularly well in a particular problem and perform poorly in a different one. In fact, the ability of an algorithm to handle problems from different domains in a robust way will ultimately make the difference between popular and unpopular algorithms. From all the algorithms described in this dissertation, the heuristic algorithms for the inference of RODGs and the heuristic algorithms for the inference of combinational Boolean networks exhibited a higher degree of robustness and were more flexible in different domains. For this reason, the majority of the comparisons with respect to the second criterion, the quality of the inference performed, are made using these algorithms. These comparisons are performed in chapter 7 that also contains two examples of the application of the algorithms to more complex problems. Chapter 7 not only describes the solutions obtained and analyzes how the algorithms performed, but also shows how a VLSI implementation of the resulting classifiers is straightforward to obtain. This is done mainly to illustrate how the use of these representations makes it easy to implement classifiers using digital VLSI technology.

Finally, chapter 8 summarizes and puts in perspective the results obtained. An analysis of the major strengths and weaknesses of the algorithms is presented and directions for future research are proposed.

Chapter 2

Inductive Biases and Complexity

The problem of selecting the hypothesis that will be more accurate in the test set is, in general, an ill-posed problem because many hypotheses that are consistent with the training set exist. A preference, either explicit or implicit for one hypothesis over an alternative one is called a bias.

No bias is intrisically superior to any other bias for all problems and an argument for the superiority of a particular bias can only be made in a particular context. In fact, a conservation law that states in a particularly simple way the inherent equivalence of all biases in the absence of a context can be easily derived. It turns out, however, that not all problems are equally likely to appear and that a very general rule can be used to select an hypothesis that has good generalization accuracy. This chapter is dedicated to the study of these arguments.

2.1 Definitions and Conventions

The problem domain D, defines the set of possible input objects, $d \in D$. The training set $T = (\{d_1 \ldots d_m\}, t)$ is defined by a set of m elements of D and the corresponding label vector, t.

Each element of D is defined by a number of discrete attributes, $\{a_1, \ldots, a_{N'}\}$. This domain is mapped into a Boolean one defined over a space of N Boolean variables $\{x_1, \ldots, x_N\}$ as described in section 1.1.3.

The label vector t contains m components and t_j is defined to be 1 if the object d_j is labeled as belonging to the target concept, 0 otherwise.

It will be useful, sometimes, to consider x_i not only as the ith input variable, but also as the vector of values that x_i takes for the instances in the training set. In this case, x_i^j represents the value that variable x_i takes in the jth instance in the training set. These binary vectors can also be manipulated using the common Boolean operators by applying them successively to each element of the vectors.

An incompletely specified Boolean function $f : \{0,1\}^N \rightarrow \{0,1,X\}$ is defined by its *off*, *on* and *dont-care* sets (f_{off}, f_{on}, f_{dc}). If a function has $f_{\text{dc}} = \emptyset$ then f is a completely specified function and can be viewed as the characteristic function of some subset of the input space. The training set defines an incompletely specified function, where the positively and negatively labeled examples correspond to the *on* and *off* sets respectively.

A minterm z represents an assignment of values $z_1, z_2, ..., z_N$ to the input variables and corresponds to one vertex in the input space hyper-cube. Two functions f and g are called compatible iff $f_{\text{on}} \bigcap g_{\text{off}} = \emptyset$ and $f_{\text{off}} \bigcap g_{\text{on}} = \emptyset$.

When describing Boolean functions the conjunction sign will, in general, be omitted and the $+$ or $\wedge$ signs will be used to represent disjunction.

2.2 Deterministic vs. Probabilistic Concept Learning

In deterministic single concept learning, *concepts* and *hypotheses* are subsets of the domain D. *Concept* will be used to describe the underlying true rule and *hypothesis* will be used to describe the approximations generated by the learner to the target concept. The objects in the training set are labeled positive if they are a member of the target concept and negative otherwise. The objective is to generate an hypothesis that is a good approximation of the target concept, in a sense that will be defined precisely later.

In some domains, the characteristics of the problem may be such that the correspondence between the label of an object and the value of its attributes is not defined in a deterministic way. This means that an object $d \in D$ with a certain combination of attributes will have a certain probability $\gamma(d)$ of being labeled as a positive instance of the target concept. The function $\gamma(d) : D \rightarrow [0,1]$ gives the probability that a given element of D is assigned a positive label. This corresponds to the problem of classification in the presence of noise, as defined in section 1.1.2. To formulate this problem as a single concept learning problem, the target concept can be defined as the set of all objects d such that $\gamma(d) > 0.5$. The hypothesis that coincides with C is the one that minimizes the error in

the test set. In this case, it is important to define the quantity $P_L^m(T|\gamma)$, that represents the probability of the observed training set under the condition that the labeling function is $\gamma(d)$. For a training set $T = (\{d_1 \ldots d_m\}, t)$ this quantity is given by

$$P_L^m(T|\gamma) = \prod_i^m (1 - |t_i - \gamma(d_i)|) \tag{2.1}$$

In practice, access to the true concept C is not available and therefore the only way by which a generated hypothesis can be evaluated is by testing it in unseen instances. Therefore, whether or not the hypothesis is a good approximation to the somewhat artificial definition of the *true* concept above is irrelevant. What matters is the performance of that hypothesis for the instances in the test set. This work does not address the more general problem of actually estimating the value of $\gamma(d)$ for the points in the domain. In fact, and purely for the effect of performing classification, the above approach is sufficient if the target concept can be well approximated by a generated hypothesis. In many problems, all instances that are likely to appear have $\gamma(d)$ very close to 0 or very close to 1. For instance, in the chess endings example discussed before, it may be the case that there exists a small probability that a given chess ending is mis-labeled. If this probability is small, this should not preclude the learner from inferring the exact rule, something that would be impossible if a deterministic labeling is assumed. This type of situation is very common in classification problems and can be handled in an effective way by assuming that a probabilistic labeling takes place, although, in fact, a deterministic labeling was made but was corrupted by some level of noise. In this case, for a given level of noise, there is a bidirectional correspondence between each choice of γ and each concept C, and the conditional probability in expression (2.1) can be written $P_L^m(T|C)$, representing the probability of the labeling observed conditioned on the fact that the target concept is C.

The deterministic case is simply the special case where the function $\gamma(d)$ is 1 for all instances in C and 0 for the remaining ones. In fact, even though a unified approach can be defined for the more general case it is sometimes useful to use the knowledge that a given problem is deterministic.

2.3 Formal Learning Models

2.3.1 Identification in the Limit

One of the first formal treatments of the problem addressed here is due to Gold [30]. He studies the problem of identifying a given language given a sample of strings in that language. The learner is presented with an infinite sequence of strings in some language $\mathcal{L}$ that is known to belong to a given class of languages.

In Gold's model, after each string is presented, the learner makes a guess of which language the strings belong to. The learner is said to *identify in the limit* a given language if, after a finite time, the guesses are all the same, equal to $\mathcal{L}$ and this behavior happens for all possible orders in which the strings can be presented.

Identification by enumeration is the proposed method to perform identification in the limit. The method lists all languages in the class in some order. At any given time, the learner outputs as a guess the first language that is consistent with the data observed so far. This method can always be used if is it possible to enumerate the languages and if an effective test for membership in a language exists. In practice, it is limited to relatively simple problems because, in general, it requires exponential time.

The main results of Gold's work on the classes of languages that are identifiable in the limit are essentially negative. In particular, he proved that no family of languages that contains all languages of finite cardinality and at least one of infinite cardinality is identifiable in the limit.

2.3.2 PAC-Learning

Gold's approach can be too pessimistic because it requires the learning algorithm to output an hypothesis that is exactly equal to the target concept for all possible ways of presenting the data. Valiant's proposed a more useful definition of learnability, the *Probably Approximately Correct* (PAC) model. This model not only allows the learner to output an hypothesis that is only a close approximation to the target concept, but also opens the possibility that the learner will output, in some cases, an hypothesis that is completely wrong. This is necessary because, in the probabilistic setting used by Valiant, there is a finite chance that the learner will have access to a training set that is totally non-representative of the target concept. In these conditions, the learner has no way of

generating an hypothesis that is a good approximation to the target concept.

In the PAC-learning framework, it is assumed that the labeling is deterministic and the training and testing instances that are presented to the learner are chosen according to a given probability distribution defined over the input space, $P_D(d)$. This eliminates the hopeless case where the learner is forced to learn under one probability distribution and is tested under a totally different distribution. The probability distribution $P_D(d)$ is extended to apply to subsets of the input space, $S \in D$, in the standard way:

$$P_D(S) = \sum_{d \in S} P_D(d) \tag{2.2}$$

PAC-learnability can now be defined formally:

Definition 1 *A family of concepts C, is said to be PAC-learnable with sample complexity m if for all probability distributions $P_D(d)$ and all concepts $C \in \mathcal{C}$ there exists a learner that* for most training sets of size m, *outputs an hypothesis H that is a* close approximation *to C.*

H is called a *close approximation* to C if

$$P_D(H \Delta C) \leq \epsilon \tag{2.3}$$

where $H \Delta C$ is the symmetric difference of H and C, defined as

$$H \Delta C \equiv (H \cap \overline{C}) \cup (\overline{H} \cap C) \tag{2.4}$$

The phrase *H is a good approximation to C for most training sets of size m* means that

$$P\{T | P_D(H \Delta C) \geq \epsilon\} \leq \delta \tag{2.5}$$

Expression (2.5) states that the probability of observing a training set $(\{d_1, \ldots d_m\}, t)$ that causes the algorithm to generate an hypothesis that is not a close approximation to C is smaller than a given constant, δ. In general, the size of the training set, m, is a function of the parameters ϵ and δ. Learning is of little interest if very large training sets are required. Particularly interesting are families of concepts that are *polynomial sample PAC-learnable*, that is, families of concepts for which the size of the training set required to satisfy definition 1 is polynomial in $1/\epsilon$ and $1/\delta$.

It will be assumed in the rest of this work that the training sets are generated with a procedure similar to the one defined in the PAC framework. More precisely, instances

in T are generated according to an underlying probability distribution $P_D(d)$. However, the labeling of the examples in the training set will, in general, be obtained using the probabilistic labeling defined by the function $\gamma(d)$ described in section 2.2.

2.4 Equivalence of all Biases

The equivalence of all biases (and consequently, of all learning algorithms) in the absence of a context is well known [35, 60]. Schaffer [88] presented a very general argument for this equivalence in the form of a conservation principle.

He restricts the analysis to single concept learning problems in discrete domains. Since the domain is discrete, there will be a finite number of possible attribute combinations, or objects in D. A learning situation in a particular domain is defined by a triple (P_D, γ, m), where P_D is the probability distribution according to which the training sets are generated (as in section (2.3.2)), γ defines the probability distribution that defines the way the labels are assigned to instances in the domain (as in section (2.2)) and m is, as before, the size of the training sets.

The generalization performance of a learner L in learning situation S, $\mathrm{GP}_L(S)$ is defined as the average accuracy of the generated hypothesis when applied to examples not present in the training sets minus the constant 0.5. The subtraction of this constant takes into account the fact that a classification scheme that randomly guesses the class obtains an average generalization accuracy of 0.5. A generalization performance of 0.1 signifies, therefore, that the generated hypothesis will correctly classify, in the average, 60% of the unseen instances, sampled according to the probability distribution $P(d)$. Given this definitions, Schaffer shows that

$$\int_\gamma \mathrm{GP}_L(P_D, \gamma, m) = 0 \tag{2.6}$$

for any learning algorithm L where the integral is computed over all possible choices for γ. Since γ defines the target concept, this result states that the sum over all concepts of the generalization performance is 0 for any learner L. If the labeling is deterministic then there are only $2^{|D|}$ possible choices for γ (each defining a different concept) and the integral in expression (2.6) becomes a summation over all possible concepts

$$\sum_\gamma \mathrm{GP}_L(P_D, \gamma, m) = 0 \tag{2.7}$$

The fact that this sum equals 0 means that it is impossible for a learning algorithm to improve its performance in a particular set of concepts without showing decreased performance in another set. Some interesting examples of possible and impossible learners are given in [88].

However, not all possible concepts are equally likely to appear in the real world. In fact, if there were no regularities, learning would be fundamentally impossible because, no matter how large the training sets, there would be no basis to infer any particular rule. The statement that not all concepts are equally likely is equivalent to the assumption that there exists a probability distribution $P_C(C)$ that selects the target concepts from the total universe of possible ones. In this case, expressions (2.6) and (2.7) do not apply because the learner can perform well in concepts that are very likely to appear and badly in concepts that are unlikely to be the target. If $P_C(C)$ exists the learner can use Bayes rule to select the hypothesis that is more likely to perform well in unseen instances.

2.5 The Application of Bayes Law to Hypothesis Selection

Assume that the target concept is chosen according to some probability distribution $P_C(C)$, defined over some family of concepts, $\mathcal{C}$. Bayes law states that the probability that a given hypothesis H is equal to C, the target concept, is given by

$$P_P(H|T) = \frac{P_L^m(T|H)P_C(H)}{\sum_H P_L^m(T|H)P_C(H)} \tag{2.8}$$

where $P_L^m(T|H)$ is, as before, the probability of the observed training set conditioned on the fact that the target concept is equal to H. The *maximum a posteriori* (MAP) rule says that the learner should select the hypothesis that maximizes the posteriori probability $P_P(H|T)$. The MAP rule maximizes the probability that the learner will pick the correct hypothesis and is the best that can be done if the learner has to output a single hypothesis in $\mathcal{H}$.[1] If the labeling is deterministic, then $P_L^m(T|H)$ is either 1 for hypotheses that are consistent with the training set or 0 for hypotheses that are inconsistent. In this case, the selection of the best hypothesis is made based only on the value of $P_C(H)$.

Expression (2.8) can only can be used if the *a priori* probability distribution for the concepts in $\mathcal{C}$, $P_C(C)$ exists and is known. This is usually not the case. A variety

[1]The Bayes optimal algorithm uses the probabilities computed for each hypothesis in $\mathcal{H}$ using expression (2.8) to perform a weighted vote. However, the classification obtained may not correspond to any hypothesis in $\mathcal{H}$ and, therefore, the Bayes optimal algorithm does not fit in the current framework.

of different but closely related approaches justifies the attribution of probabilities to the concepts in C (and therefore to the hypotheses in $\mathcal{H}$) in different ways. The selection of these prior probabilities is the main subject of section 2.6.

2.6 Searching for Simple Representations

The problem with the application of expression (2.8) is that the probability distribution that generates the target concepts is usually not known or not accessible and has to be approximated using general rules that are as universal as possible. How Occam's razor can be used to perform this function is the subject of this section.

2.6.1 Occam's Razor

William of Occam's[2] famous principle *non sunt multiplicanda entia praeter necessitatem* literally means *entities should not be multiplied unnecessarily.* However, it is usually interpreted as stating that the *simplest theory that fits the available data is the one more likely to predict correctly the future.* This statement can be viewed as a general rule for the selection of a probability distribution that approximates $P_C(C)$.

Occam's razor is eminently reasonable and, at first sight, hardly needs a justification. Both statements *"the Sun rises every morning"* and *"the Sun has risen every morning until today but will not rise tomorrow"* fit all the available data. In the absence of further information, they would predict equally well the future. However, even in the absence of other knowledge, one is more tempted to accept the former than the later, if nothing else because the later is unnecessary complex. Physicists have long aimed for the most simple and elegant theories and have, to a remarkable degree, succeeded in making good predictions using these theories.

The problem lies in the fact that, in general, the complexity measures used are not unique and depend on the particular approach that is used to describe the hypotheses. What is simple under one representation scheme may be very complex under another and the complexity of an hypothesis depends heavily on the primitives used to express it. It is a remarkable fact that there exists a complexity measure that is, in a sense, universal.

[2]William of Ockham (1285-1349), usually spelled Occam.

2.6.2 A Universal Complexity Measure

Kolmogorov, Solomonoff and Chaitin arrived, in an independent way, at the definition of what is usually known as Kolmogorov complexity.[3] A full discussion of the concepts involved in the definition of this complexity measure is outside the scope of this introduction. The reader is referred to one of the existing reviews of the subject [56] for a more complete treatment. The Kolmogorov complexity theory defines a universal complexity measure for finite binary strings. Since any hypothesis in a given domain can be encoded as a finite binary string, the existence of a universal complexity measure together with the application of Occam's razor seems to solve the problem of defining the prior probabilities for the concepts in $\mathcal{C}$. This is not the case because a direct use of the Kolmogorov complexity as the guiding principle is not possible since it is a non-computable function. Nonetheless, the theory opens the way to approaches that approximate this universal probability distribution.

Let M be a Turing machine with a one way output tape, a one way input tape and a two-way work-tape and let the input alphabet for this machine contain only zeros and ones (no blanks). These are the self-delimiting Turing machines, because the set of inputs for which each machine stops if prefix-free, no string being a prefix of any other string. A string p is called a program for M if M stops exactly after scanning the last bit of p. The *self-delimiting Kolmogorov complexity*, (hereby abbreviated to *Kolmogorov complexity*[4]) of a binary string s with respect to machine M, $K_M(s)$ is given by the length of the smallest input string that will cause M to write string s in the output tape. Now, if M is chosen to be a universal Turing machine the Kolmogorov complexity with respect to M is called simply the Kolmogorov complexity and is defined as $K(s)$. Kolmogorov [51], Solomonoff [95] and Chaitin [20] proved that, for any Turing machine M

$$K(s) \leq K_M(s) + k \text{ for all finite strings } s \tag{2.9}$$

where k is a constant that depends only on M. This result is important because it shows that each finite string has a complexity that does not depend (except for the constant factor) on the particular approach used to describe it. Using the above definitions, it is possible

[3] Some authors have proposed the use of the somewhat awkward but precise *Kolmogorov-Solomonoff-Chaitin complexity*, but this is the most commonly accepted terminology.

[4] There are two Kolmogorov complexity measures that differ by a logarithmic factor. The one used here has some properties that are required in the sequence.

to define a probability distribution that attributes a certain probability to any string s, the (non-computable) Solomonoff-Levin distribution:

$$P_{\mathrm{SL}}(s) = \sum_{p:M(p)=s} 2^{-|p|} \tag{2.10}$$

where the sum is over all programs p that are an encoding for the string s and $|p|$ is the length of program p. It can be shown that this expression properly defines a probability distribution[5] over the set of all finite strings. Furthermore, this probability distribution is universal in the sense that, for any *computable* probability distribution $\mu(s)$, there exists a positive constant c such that $P_{\mathrm{SL}}(s) \geq c\mu(s)$. This means that, given enough data, the application of Bayes law using the Solomonoff-Levin distribution as the prior distribution is guaranteed to converge to the right solution. Naturally, a larger training set may be needed than in the case where the exact distribution is available. However, the fact that $P_{\mathrm{SL}}(s)$ is off by, at most, a constant factor, guarantees that the extra amount of data required is not large. In fact, not only is this approach guaranteed to converge to the true solution, but it will converge faster than any other method up to a constant multiplicative factor. Levin has shown that the Solomonoff-Levin distribution and the Kolmogorov complexity are related by

$$P_{\mathrm{SL}}(s) = e^{-K(s)+O(1)} \tag{2.11}$$

Therefore, using the Solomonoff-Levin distribution as the prior probability is equivalent to the assumption that more complex hypothesis (as measured by the Kolmogorov complexity of the corresponding encodings) are less likely to be the right answer. This is a very general and elegant justification of the Occam's razor approach. Regrettably, using directly this probability distribution as the prior distribution is not a feasible approach because of several problems. First and foremost, the Solomonoff-Levin distribution is non-computable, because its computation implies the solution of the halting problem which is undecidable. It can be approximated from below, but there is no way to know how close the approximation is. Furthermore, even though the convergence results are extremely important in the limit of long strings, for simpler problems the constant factors involved may make it a sub-optimal solution if there is any other way to better approximate the true probability distribution, $P_C(H)$.

[5] More precisely, $P_{\mathrm{SL}}(s)$ defines a semi-measure over the space of strings and can be used to define a probability distribution after appropriate normalization. However, it can used directly in (2.8) because all hypotheses are discounted by the same factor.

The underlying ideas can, however, be applied in more restricted contexts, with very good results. The *Minimum Description Length Principle* can be viewed as a way to choose between alternative hypotheses using an approximation to the Solomonoff-Levin distribution.

2.7 The Minimum Description Length Principle

Rissanen's *Minimum Description Length* (MDL) Principle [80, 81] can be viewed as a way to select hypotheses in a way that approximates the results obtained using Bayes law and the Solomonoff-Levin distribution. If we take the negative logarithms of expression (2.8) we get

$$-\log_2 P_P(H|T) = -\log_2 P_L^m(T|H) - log_2 P_C(H) + \log_2 \sum_H P_L^m(T|H) P_C(H) \qquad (2.12)$$

Since T, the training set, is fixed, and the last term does not depend on H, the maximum of $P_P(H|T)$ is obtained by minimizing $\log_2 P_L^m(T|H) + \log_2 P_C(H)$. Now, if the hypotheses are encoded using an efficient self-delimiting code, the Kolmogorov complexity can be approximated by the length, in bits, of the string that describes the encoded hypothesis, $K_{\text{hyp}}(s)$. The Solomonoff-Levin distribution is now approximated (up to an irrelevant constant factor) by choosing $P_C(H) = 2^{-K_{\text{hyp}}(H)}$ and the term $-\log_2 P_C(H)$ is simply the length of the description of H using this encoding scheme.

In general, a given hypothesis will not match exactly the training set data. Let $E(T, H)$ be a string that describes the exceptions to this hypothesis present in the training set. $E(T, H)$ can also be encoded using some efficient self-delimiting encoding scheme. If this is the case, the same reasoning can be applied and $P_L^m(T|H)$ will be given by $2^{-K_{\text{exc}}(E(T|H))}$. The maximum value of $P_P(H|T)$ is therefore obtained by selecting the hypothesis H that minimizes

$$K_{\text{hyp}}(H) + K_{\text{exc}}(E(T, H)) \qquad (2.13)$$

This is the minimum description length (MDL) principle of Rissanen. The MDL principle can be viewed as a way to replace the need to estimate the a priori probability of an hypothesis by the somewhat more tractable problem of selecting the hypothesis that minimizes the total code length of expression (2.13). Naturally, one has to provide algorithms to encode both hypotheses and strings that describe exceptions. To keep expression (2.13)

computable, the encoding schemes cannot be based on Turing machines but on techniques that are well adapted to the particular representations used. In any case, for the approach to be justifiable by the arguments described above, the algorithms have to generate codes that are compact, and, in some sense, close to the optimal.

If the labeling is deterministic, $P_L^m(T|H)$ is either 1 or 0 and the maximization of (2.8) is equivalent to the selection of the consistent hypothesis with higher a-priori probability, i.e., the hypothesis H that minimizes

$$K_{\text{hyp}}(H) \tag{2.14}$$

In practice, it turns out that in many problems the labeling is almost deterministic but not exactly. Furthermore, even in deterministic problems, the representation scheme used may not be powerful enough to represent the exact hypothesis that maximizes (2.8). In these cases, it may happen that an expression that is somewhere between expressions (2.13) and (2.14) will actually give the best results. The most flexible algorithms will therefore select the hypothesis that minimizes

$$\alpha K_{\text{hyp}}(H) + K_{\text{exc}}(E(H)) \tag{2.15}$$

where α is a parameter between 0 and 1 that can be adjusted to maximize the performance. Believers in the MDL principle may find that the introduction of this extra parameter goes against the philosophy underlying the MDL principle but, in practice, this can be viewed as a way to compensate for inefficiencies of the coding scheme. It can also be viewed as a way to customize the algorithm for a particular application where considerations other than overall performance may be at play. For instance, if accurate performance in the training set is highly desirable, even at the expense of some deterioration in the overall accuracy, then α should be chosen smaller than 1. In the limit, when $1 >> \alpha$ the hypotheses chosen will perform perfectly in the training set (because of the high relative weight of the second term) but may be overly complex to perform well in the test set.

2.7.1 A Prefix Free Encoding Scheme for Graphs

The computation of $K_{\text{hyp}}(H)$ depends on the representation selected and the encoding scheme used. However, even though different representations will use different encoding schemes, most of them share the need to describe a graph. The encoding proposed

here for graphs is inspired on the encoding techniques proposed by Quinlan and Rivest [79] to encode decision trees.

The simple version presented here assumes that each node has a known number of outgoing edges. A variation that can handle the more general case will be presented in the chapters that require it.

Let G be a directed graph with one distinguished node. Assume that the outgoing edges of each node are ordered in some arbitrary, but fixed, way. For example, if each node has two outgoing edges, one labeled with a 0 and the other labeled with a 1, the edge labeled with 0 will always be considered before the edge labeled with 1. The structure of the connected part of this graph (i.e., the set of nodes that can be reached from this special node) can be encoded using the following encoding scheme:

- A node that was never visited before is encoded starting with 1 followed by an encoding of the nodes reached by following each of the outgoing edges.
- A node that was visited before is encoded starting with 0 followed by a reference to the (already described, at least partially) node.

Following Quinlan and Rivest, the issues related with the use of non-integral numbers of bits are ignored in the computation of encoding length because only a measure of complexity is desired and an actual transmission of the code does not have to be accomplished. As an example, the graph of figure 2.1 is encoded using the following encoding:
`(1 (1 (1 (1 0 00 0 11) 0 00) 0 01) 0 10)`

The total length of this encoding is 19 bits. Parentheses and spaces were used only to increase readability and do not belong to the actual encoding. A simple binary code was used to make references to nodes described previously. The number of bits required for each reference is known in advance because the number of nodes already visited is known at each point in time.

2.7.2 A Prefix Free Encoding Scheme for Exceptions

The computation of $K_{\text{exc}}(E)$ where E is a string that describes the exceptions is, on the other hand, essentially independent of the representation used.

The exceptions will be encoded using strings of 0's and 1's. The encoding of this type of strings follows closely the encoding used by Quinlan and Rivest [79] where the 1's

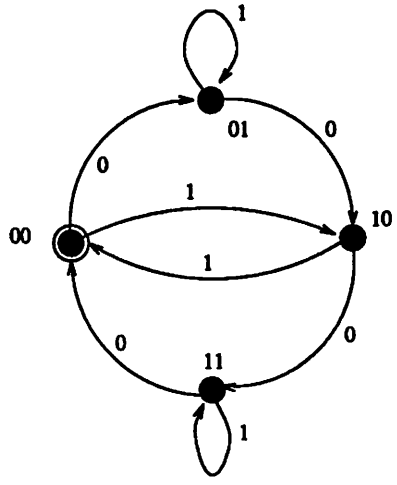


Figure 2.1: A 4 node graph with two outgoing edges for each node

will indicate the location of the exceptions. In general, these strings will have many more 0's than 1's. Assume that the strings are of length m, known and that there are $k \leq m$ 1's in the strings.

The string can be encoded by first describing the value of k, which requires $\log_2(m)$ bits and then describing which of the strings with k 1's describes the exceptions. Since there are $\binom{m}{k}$ such strings, the description length of the description will be given by

$$K_{\text{exc}}(E) = \log_2(m) + \log_2 \binom{m}{k} \tag{2.16}$$

Using Stirling's formula to approximate the second term in (2.16)

$$K_{\text{exc}}(E) = mH(\frac{k}{m}) + \frac{\log_2(m)}{2} - \frac{\log_2(k)}{2} - \frac{\log_2(m-k)}{2} - \frac{\log_2(2\pi)}{2} + \log_2(m) + O(\frac{1}{m}) \tag{2.17}$$

where $H(p)$ is the usual entropy function

$$H(p) = -p\log_2(p) - (1-p)\log_2(1-p) \tag{2.18}$$

2.8 Computational Complexity of Hypothesis Selection

The previous section described the theoretical results that support the selection of the hypothesis of minimal complexity as the most promising one. However, these results

do not address the problem of how the learner actual selects such an hypothesis. So, even if the learner is using the MDL principle as the guiding criterion for selecting one of the possible hypotheses, the task of designing an algorithm that actually performs the selection of the hypothesis that has minimal total encoding length is still open.

For many problems of interest, this procedure is of high computational complexity. In fact, for almost all the representation schemes addressed in this work, the problem of selecting an hypothesis of minimum complexity belongs to a class of problems that are believed to be very difficult to solve efficiently, the NP-complete class. A complete explanation of the concepts involved in the definition of this complexity class is outside the scope of this work and the reader is referred to [27] for an excellent review of the subject.

The following lemmas show that the computational complexity of the problems addressed in this dissertation is high. Since these results are not critical for the exposition that follows, only brief references are made to the respective proofs.

Lemma 1 *It is NP-complete to determine if a two-level threshold gate network consistent with a given training set and with less than k gates exists.*

Proof: this problem is one of the versions of the loading problem proved NP-complete in [45] and [11].

Lemma 2 *For a fixed ordering of the variables, it is NP-complete to determine if a reduced ordered decision graph that is consistent with a given training set and has less than k nodes exists.*

Proof: obtained by a reduction from graph K-colorability in [97].

Lemma 3 *It is NP-complete to determine if a finite state machine that has less than k states and is consistent with a given training set*[6] *exists.*

Proof: made by a reduction from the satisfiability problem with 3 variable clauses (3-SAT) in [32].

The problem of selecting the minimum multi-level Boolean network that is consistent with a given training set is also of high complexity. It is known to be in NP, because a solution can easily be checked for correctness in polynomial time. However, no published proof that it is NP-hard is known to the author, although it looks unlikely that a deterministic polynomial time algorithm exists for this problem unless P=NP.

[6] In this case, the training set consists of input/output sequences

Chapter 3

Two-Level Threshold Gate Networks

3.1 Related Work

Threshold gate networks have been the focus of an increasing interest in the research community in the last few years. In part, this interest is due to theoretical work that shows that threshold gates are more powerful than simple Boolean gates, in the sense that polynomial size, bounded level, networks of threshold gates can implement functions (and therefore represent concepts) that require unbounded level networks of purely logic gates. For example, it has been shown that functions like multiple-addition, multiplication, division and sorting can be implemented by polynomial-size threshold circuits of small constant depth [93, 94]. In particular, compact two-level threshold gate networks can represent many interesting concepts that require exponentially large two-level Boolean networks.

Extensive research in the field of neural networks has also created interest in algorithms for the synthesis and optimization of threshold gate networks. The more popular neural network architectures use threshold gate models that differ substantially from the ones studied here. In the first place, they assume the weights and the threshold value for each gate are defined by real numbers of arbitrary precision. In the second place, the most commonly used neural network learning algorithms assume that the function implemented by a threshold gate is continuous and differentiable. In the most commonly used models,

the output of each gate is given by

$$f_{\text{act}}\Big(\sum_i (w_i x_i) - w_0\Big) \tag{3.1}$$

where f_{act}, the activation function, is continuous and differentiable. A commonly used activation function is given by

$$f_{\text{act}}(x) = \frac{1}{1+e^{-x}} \tag{3.2}$$

plotted in figure 3.1 The most popular leaning algorithms derive the input (or connection)

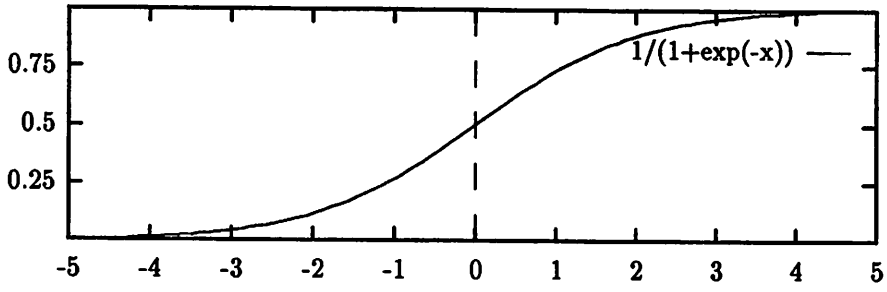


Figure 3.1: A typical activation function

weights for each of the gates in the network by minimizing, with respect to the connection weights, an error function defined over the weight space. For a single output network a typical error function is given by

$$\epsilon = \sum_j (t_j - o_j)^2 \tag{3.3}$$

where t_j is the desired value for the output when instance j is presented (given by the j label in the training set) and o_j is the value computed by the network. The partial derivatives of expression (3.3) with respect to each one of the weights in the network can be easily obtained and gradient descent techniques can be used to obtain a local minimum of the error in weight space. Many algorithms based on this technique have been proposed [8, 41, 52, 84] and used with varying degrees of success in a variety of problems. These algorithms, however, do not derive the architecture of the network, but only the connection weights. Constructive algorithms like the cascade-correlation [24] and others [25, 5] derive the architecture by modifying the error minimization strategy [39] and allowing for new units to be created if the current solution is not satisfactory. Another possibility is to prune the network obtained in an effort to minimize the overall complexity of the classifier [55].

In all these approaches, the selection of a minimal complexity network remains secondary to the minimization of (3.3).

All these algorithms use standard optimization techniques to select a connection weights combination that is a local minimum of (3.3). In many cases, these techniques are unable to find a solution that performs adequately in the training set, specially if the number of gates in the network is close to the minimum. This is a serious limitation of neural network training algorithms and limits the accuracy of the generalization performed by neural network classifiers in some problems.

This chapter describes a new formulation for the problem of selecting a minimal two-level network of threshold gates that matches exactly the data in the training set. The threshold gate model used is more restrictive than the one used in neural networks, in that both the weights and the threshold value in expression (3.1) are forced to be integer valued and the activation function is given by:

$$f_{\text{act}}(x) = \begin{cases} 0 & \text{if} \quad x < 0 \\ 1 & \text{if} \quad x \geq 0 \end{cases} \tag{3.4}$$

Expression (3.4) is equivalent to (1.1) and defines the function implement by a threshold gate in this model. The algorithm described here assumes that the labeling of the examples is deterministic, i.e., that no classification noise is present. The class of networks generated by the algorithms is much more restricted than what can be obtained by using neural network training algorithms. However, by using this restricted representation and a different approach, the solution is guaranteed to match the training set data, a problem that is, in itself, NP-complete to solve using neural network type algorithms [11, 45].

The selection of the minimal complexity solution is still an NP-complete problem, but any solution generated by the algorithm classifies correctly, by construction, all the instances in the training set. The networks generated are, in many cases, much smaller than the ones that can be obtained using any of the neural network algorithms described above. This approach is therefore an interesting alternative in problems that do not require real valued weights and map, in a natural way, into a two-level threshold gate network representation.

Previous work closely related to the approach described here is limited. Muroga [61] makes a good exposition of the properties of functions implemented by threshold gate networks and proposes algorithms for their synthesis but his approach is limited to functions

of a very small number of variables and is heavily based on the use of pre-computed tables for this type of functions.

3.2 Problem Formulation

The formulation developed for the problem of selecting a minimal complexity two-level threshold gate network is inspired on concepts and techniques developed for the synthesis of minimal two-level Boolean networks. Concepts like cubes and covers are generalized to reflect the function implemented by threshold gates. For the benefit of the reader unfamiliar with these concepts, these definitions and concepts are introduced before the corresponding generalizations. For clarity, the concepts and algorithms involved are first defined for the case where the threshold gates are restricted to have weights in the set $\{-1, +1\}$. Section 3.3.3 describes how the formulation can also be applied to the general case by applying a simple problem transformation.

3.2.1 Cubes and Pyramids

A literal is defined to be either a variable or its negation. A cube is a conjunction of k literals ($1 \leq k \leq N$), where no two literals corresponding to the same variable appear. A cube with N literals corresponds to a minterm, a point in the input space. A cube c_1 is said to contain another cube c_2 if $c_2 \Rightarrow c_1$, i.e., if the truth values defined in c_2 make c_1 true. This is equivalent to stating that all points in the input space contained in c_2 are also contained in c_1. If $c_1 \neq c_2$ such a containment is proper. The dimension of a cube c is the number of variables not present in c. The distance between two cubes, c_1 and c_2, $\delta(c_1, c_2)$ is the number of variables that appear negated in one cube and non-negated in the other. For example, $x_1\overline{x_3}x_4$ is at distance 2 from $\overline{x_1}x_2\overline{x_4}$.

A cube is identified with the boolean function implemented by an *and* gate. Consider now a threshold gate with a function defined by (3.1) and (3.4), with weights in the set $\{-1, +1\}$. Assume, without loss of generality, that the gate implements a function of the first k variables in an N-dimensional input space. Let V_{max} be the maximum value that the sum $\sum_i w_i x_i$ can take for a fixed value of the weights and cube c be defined by the literals $c_1 \ldots c_k$ where $c_i = x_i$ if $w_i = +1$ and $c_i = \overline{x_i}$ if $w_i = -1$. Consider now a minterm $z = z_1 \ldots z_k \ldots z_N$. This minterm will turn the gate *on* iff no more that $V_{max} - w_0$ literals in z are different from the corresponding literals in c, i.e., if $\delta(c, m) \leq V_{max} - w_0$.

Let $h = V_{max} - w_0$. A pyramid is defined as a pair $(c : h)$, where c is a the apex cube and h is a non-negative integer value, the height. The minterms contained by a pyramid are at distance h or less from the apex cube. Figure 3.2 shows a graphical representation of the minterms covered by a cube and a pyramid.

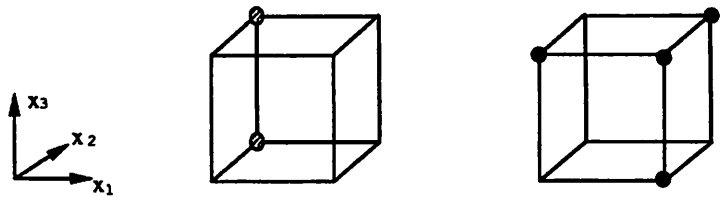


Figure 3.2: Cube $\overline{x_1}x_2$ and pyramid $(x_1\overline{x_2}x_3 : 1)$

3.2.2 Properties of Pyramids

The distance between a cube c and a pyramid $p = (c_p : h)$ is given by $\max(\delta(c, c_p) - h, 0)$. This distance measure corresponds to the intuitive notion of distance in Boolean spaces and is equivalent to the minimum distance between a minterm in p and cube c. For example, cube $\overline{x_1}x_2$ and pyramid $(x_1\overline{x_2}x_3 : 1)$ in figure 3.2 are at distance 1.

A pyramid p is a prime pyramid, relatively to some boolean function f, iff there is no other pyramid contained in the $f_{on} \bigcup f_{dc}$ set that properly contains pyramid p. Formally, $p = (c : h) \subseteq (f_{on} \cup f_{dc})$ is prime with respect to f iff

$$\forall (c' : h'), (c' : h') \supset (c : h) \Rightarrow (c' : h') \not\subseteq (f_{on} \cup f_{dc}) \tag{3.5}$$

Any pyramid $p = (c : h)$ has a complement pyramid, $\overline{p} = (\tilde{c} : k - h - 1)$, where k is the number of literals in c and $\tilde{c}$ is the cube obtained by complementing all such literals. For any pyramid p, $p\overline{p} = \emptyset$ and $p \cup \overline{p} = \{0,1\}^N$.

3.2.3 Covers and M-covers

A set of cubes S is a cover for a Boolean function f if all points in f_{on} are covered by at least one cube in S and no point in f_{off} is covered by a cube in S. A two-level Boolean network that implements a function compatible with f can be obtained directly from a cube cover. Each cube corresponds to an *and* gate and the outputs of the *and* gate feed the second level *or* gate. The concept of cube cover of a boolean function can be generalized in

a way that preserves the relation between a cover and a two-level implementation of f but allows for the use of general threshold gates in both the first and second levels.

A set of pyramids is a pyramid cover for a function f if every minterm in f_{on} is contained in at least one pyramid and no minterm in the f_{off} set is contained in any pyramid. This concept of cover can be further extended to a more general one that leads to implementations where the second level gate is also a general threshold gate instead of an *or* gate. A bag of pyramids, B, is an M-cover ($M \geq 1$) for a function f iff all minterms in the f_{on} set are covered by at least M pyramids and all minterms in the f_{off} set are covered by at most $M-1$ pyramids in B.

Figure 3.3 shows how pyramids $p_1 = (\overline{x_1}\,\overline{x_2}\,\overline{x_3} : 2)$ and $p_2 = (x_1x_2x_3 : 1)$ are a 2-cover for the function f defined by $f_{on} = \{x_1x_2\overline{x_3}, x_1\overline{x_2}x_3, \overline{x_1}x_2x_3\}$, $f_{off} = \{0,1\}^3 \backslash f_{on}$ and $f_{dc} = \emptyset$.

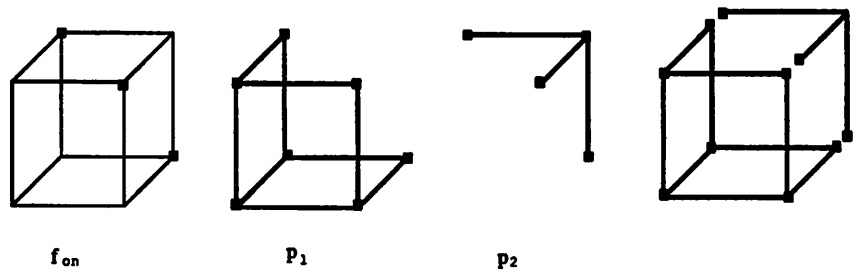


Figure 3.3: 2-cover for function f.

A two-level network of threshold gates that implements f can be obtained directly from an M-cover by simply allocating one threshold gate for every pyramid in B and connecting them to a gate in the second level with all weights equal to 1 and a threshold value of M.

3.2.4 Expanding and Reducing Pyramids

Algorithms for the synthesis of two-level Boolean networks make extensive use of the ability to incrementally change a cube c into a cube c' in such a way that c' properly contains c. This operation, the *expand* operation, is performed by simply dropping one of the literals present in cube c. The opposite operation, the *reduce* operation, is also important and is performed by adding one new literal to the set of literals in the original cube.

The definition of operations with similar properties but that manipulate pyramids

instead of cubes is important because it facilitates the use of approaches based on Boolean networks minimization techniques. The expand operation, applied to pyramids, can be performed by applying one of the two following changes to the pyramid $p = (c : h)$:

1. Expand the apex cube, by dropping one literal from c. For example, the pyramid $(x_1x_2x_3 : 1)$ can be expanded to pyramid $(x_1x_3 : 1)$:

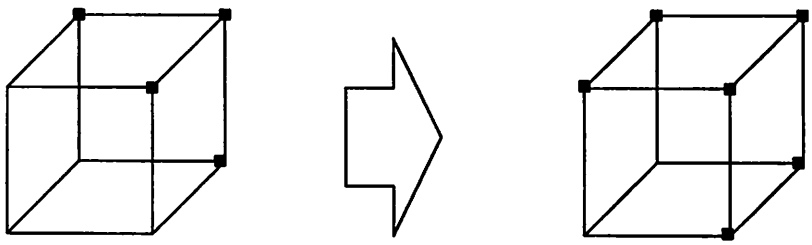


Figure 3.4: Expansion of a pyramid by expansion of the apex cube

2. Reduce the apex cube, by adding one literal to c and increase the pyramid height, h. For example, pyramid $(x_1x_3 : 1)$ can be expanded to pyramid $(x_1x_2x_3 : 2)$:

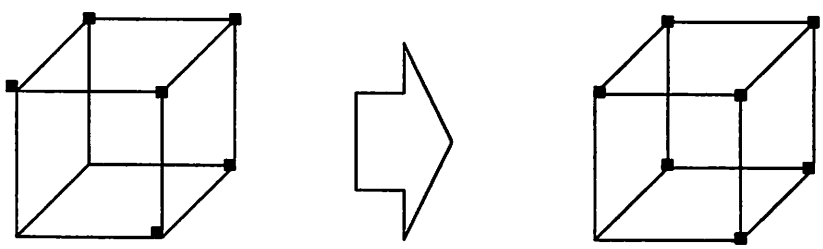


Figure 3.5: Expansion of a pyramid by increasing the pyramid height

The reduction operation can be performed by either applying the reverse operations or by expanding its complement pyramid and complementing the result. The reader may easily

verify that the expand (reduce) operation does indeed generate a pyramid that properly contains (is contained) in the original one.

The expand operation can be used to obtain a prime pyramid (with respect to some function f) by expanding a pyramid until no further expansion operations can be applied without causing some minterm in f_{off} to be covered by the pyramid.

3.3 The Search Algorithm

3.3.1 The Encoding Scheme

Given the correspondence between M-covers and the two-level networks with unit weights discussed above, it is natural to chose an encoding scheme that simply lists the pyramids contained in a given M-cover. An M-cover is therefore encoded in the following way:

- List each pyramid in the M-cover by encoding the value of h followed by a bit string with $N\lceil log_2 3 \rceil$ bits that describes the apex cube.
- Terminate the list with the encoding of an integer that is an impossible value for h (e.g., N) followed by an encoding of the value of M.

A description of an M-cover with k pyramids using this encoding scheme takes $k(\log_2 N + N\lceil log_2 3 \rceil) + \log_2 N + \log_2 k$ bits. Furthermore, this encoding has the characteristic that its length increases linearly with the number of pyramids in the cover. The minimization of expression (2.14) using this encoding scheme is equivalent to the selection of an M-cover of minimal cardinality for the incompletely specified function f defined by the training set.

3.3.2 A Local Search Algorithm

Algorithms for the minimization of two-level Boolean gate networks like, for instance, `espresso` [14], perform the search for a cover of minimal cardinality by expanding the cubes in a solution and then solving a set covering problem using standard techniques. A similar approach could be used here if the objective was to select a pyramid cover (i.e., a 1-cover) for a given function. In fact, the expand operations described above can be used to first expand maximally all pyramids in a given cover. Such an approach was actually implemented [65] but is restricted to the selection of 1-covers. When the objective is to

select an M-cover, $M > 1$, these techniques are no longer useful, because the solution does not, in general, consist of prime pyramids.

The algorithm proposed here to perform this selection is, instead, based on a tree search algorithm that searches for a better solution by performing incremental changes in an existing one. The algorithm is started with a 1-cover that is obtained by simply listing all the minterms that correspond to the positive instances in the training set, the f_{on} set. The algorithm described in figure 3.6 then searches for an M-cover of minimal description length. The CHANGECOVER procedure in figure 3.7 changes the pyramids in S in such

```
FINDCOVER()
   M = 1
   M_saved = 1
   REMOVEPYRAMIDFROMCOVER()
   repeat
      if CHANGECOVER(M) = FALSE                    Didn't find solution
         if M_saved ≠ M
            return                                 No solution found for this value of M
         M := M + 1
      else
         SAVESOLUTION()                            Best solution found so far
         REMOVEPYRAMIDFROMCOVER()
         M_saved := M
   until FALSE
```

Figure 3.6: Searching for a small M-cover.

a way that the reduced bag of pyramids represents a new M-cover. This is accomplished using a search tree. To each node n_i in this tree corresponds one bag of pyramids, b_i. Each node in the search tree is obtained by changing one pyramid from the bag that corresponds to the parent node. Nodes that have a smaller number of f_{on} points covered less than M times are explored first by the CHANGECOVER procedure until the maximum tree size is reached or a bag of pyramids that is an M-cover for f is obtained.

The expansion of a node in the search tree is performed by the algorithm described

```
CHANGECOVER(M)
  n_0 := BUILDTREEROOT()                          Create root with existing bag
  EXPANDTREENODE(M,n_0)
  for j := 1 to MAXTREESIZE
    n_i := PICKBESTNODE()  Select node with smaller number of non-covered minterms
    if UNCOVEREDMINTERMS(n_i) = 0                          An M-cover was found
      return TRUE
    EXPANDTREENODE(M, n_i)
  return FALSE
```

Figure 3.7: Transforming a bag of pyramids into an M-cover

in figure 3.8. After selecting one of the f_{on} minterms covered less than M times, the algorithm selects the k_{max} pyramids closer to it as the candidates, where k_{max} is a parameter that controls the branching factor of the search tree. Function BUILDSETS creates the s_{on} set by adding minterm z to the list of f_{on} minterms covered by p and not covered more than M times. The s_{off} set consists of all the minterms in the f_{off} set that are already covered by $M-1$ pyramids but not by p.

Function FINDCOVERINGPYR derives a pyramid covering all minterms in the s_{on} set and none in the s_{off} set. This function uses the expand and reduce operations described in section 3.2.4 to select the pyramid that satisfies this definition. The algorithm starts by identifying s_{red}, the set of minterms in s_{on} already covered by the pyramid. The pyramid is first reduced with respect to this set, i.e., it is reduced as much as possible while still covering all the minterms in s_{red}. It is then expanded until either all minterms in the s_{on} set are covered or a prime is obtained. If the second condition holds and the first doesn't, it reports failure. Otherwise, it returns the expanded pyramid.

Figure 3.9 illustrates, in an hypothetical two-dimensional projection of the space, how a search for a 1-cover takes place. The CHANGECOVER procedure receives a bag of pyramids that does not cover some of the minterms in f_{on}. At each node in the search tree, one of these minterms is selected and the nearest pyramids are expanded in such a way that the selected minterm is now covered.

```
EXPANDTREENODE(M, n_i)
  z := PICKONEUNCOVEREDMINTERM(n_i)
  while k < k_max
    p_old := CHOOSENEXTPYR(z)                          Select closest pyramid
    (s_on, s_off) := BUILDSETS(M, n_i, p_old)          Build on and off sets
    p :=FINDCOVERINGPYR(s_on, s_off, p_old)
    if p ≠ ∅                                           Pyramid exists
      n_j := CREATECHILDOFNODE(n_i, b_i ∪ {p} \ {p_old}) Create child with modified bag
```

Figure 3.8: Expanding a node in the search tree.

3.3.3 Threshold Gates With Larger Weights

All the concepts, encodings and algorithms described in the previous sections assume that the input weights for the threshold gates in the solution are either +1 or -1. This section describes how the search for a solution with larger integer weights is equivalent to a search for a pyramid in a space of higher dimensionality. Consider a function f of N variables, $\{x_1, x_2..., x_N\}$ defined by its f_{on} and f_{off} sets. Let v be an integer and f^v be a function of vN variables defined by its f^v_{on} and f^v_{off} sets. Every minterm in f^v_{on} (f^v_{off}) is obtained by replicating v times every literal in the corresponding minterm of f_{on} (f_{off}). For example, if $v = 2$ and $N = 3$ the minterm $x_1x_2\overline{x_3}$ is converted to $x_{1_1}x_{1_2}x_{2_1}x_{2_2}\overline{x_{3_1}}\,\overline{x_{3_2}}$. Now, let f be a boolean function of N variables such that f is implementable by a single threshold gate with integer input weights $\{w_1, w_2...w_k\}$. Let v be the maximum value of $\mid w_1 \mid, \mid w_2 \mid ... \mid w_n \mid$. Then there exists a pyramid in the space of vN dimensions that covers all the minterms in the f^v_{on} set and none in the f^v_{off}. This fact implies that a restriction of the input weights of the first level threshold gates to the range $[-v, +v]$ in a space of N variables, can be obtained by applying the algorithms described above to the problem defined in a new space obtained by replicating v times each input variable.

The encoding scheme described in section 3.3.1 can be changed to encode this type of solutions. The description of the literals in each cube will now require $N\lceil\log_2(2v+1)\rceil$ bits. Clearly, this approach is not economical for large values of the weights. However, large values of the weights also make for expensive implementations of the solutions obtained and

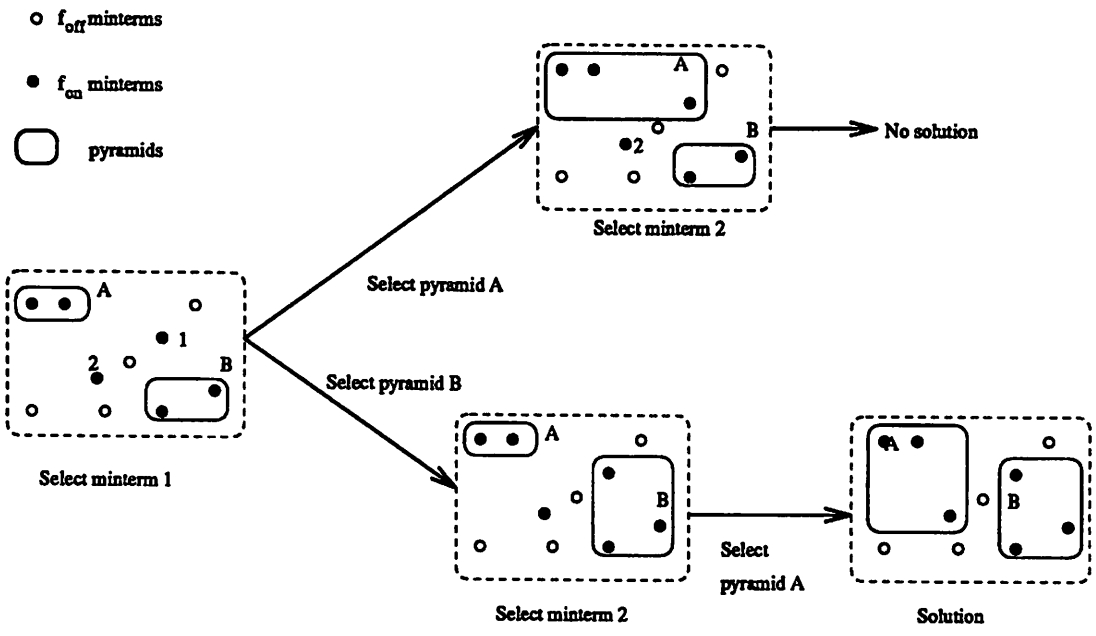


Figure 3.9: A schematic view of the CHANGECOVER procedure

for hypotheses with higher complexity.

3.4 Experimental results

3.4.1 Problems Requiring a Known Minimum Number of Threshold Gates

The results obtained by the algorithm were compared with the theoretical minimum cost realizations. All the problems require the use of non-degenerate threshold gates (i.e., threshold gates different from either *and* and *or* gates) to achieve the minimum realization. The asymmetry problems also require weights larger than 1, and this is reflected in the number of variables (N) for each problem. A detailed description of the functions used for these tests can be found in appendix B. The results are shown in table 3.1. As before, m is the size of the training set. These results show that the minimum size realization was obtained for all but the two larger parity problems. The sharp increase in CPU time for the larger asymmetry problems is due, in part, to the large weights needed. This represents a serious limitation if networks with large weights represent the optimum solutions. All the results presented were obtained in a DEC-station 3100 and all CPU times are in seconds.

Problem	m	N	*Theoretical*	Experimental	T_{cpu}
6-parity	64	6	6	6	4.3
7-parity	128	7	7	7	10.7
8-parity	256	8	8	8	61.9
9-parity	512	9	9	11	878.1
10-parity	1024	10	10	24	1759.8
6-asymmetry	64	24	2	2	1.5
8-asymmetry	256	64	2	2	23.2
10-asymmetry	1024	160	2	2	408.6
12-asymmetry	4096	384	2	2	11752.9

Table 3.1: Experiments using threshold gates.

3.4.2 Comparison With Standard Two-Level Minimizers

In the second set of problems, the performance of the algorithm was compared with the performance of a popular two-level minimizer, `espresso` [14]. This comparison was performed by constraining `lsat` to use only *and* gates in the first level and *or* gates in the second level. This can be easily done by not allowing the second type of expand operations to be performed. For each function, two randomly generated training sets of sizes 200 and 600 were generated. According to the generation procedure, all instances of the problems should accept a solution with no more cubes than the upper bound shown in table 3.2. This upper bound is given by the size of the two-level realization of the original concept. This table shows that although no specific code optimization was performed for the special case when a cube cover is to be found, the performance of the algorithm still compares well with a classic two-level optimizer. In particular, it obtains results that are either similar or better than `espresso` and much faster. Moreover, the speed gain increases with the size of the problem. This is due, in part, to the fact that `lsat` does not require an explicit cover for the f_{dc} set while `espresso` does. It is also clear from these results that both programs obtain results very far from the minimum in a large number of cases.

Problem				espresso		lsat	
Name	m	N	Upper bound	Experimental	T_{cpu}	Experimental	T_{cpu}
dnf1	200	80	6	6	144	6	87
	600	80	6	14	840	15	173
dnf2	200	40	8	8	84	9	21
	600	40	8	10	236	8	161
dnf3	200	32	6	7	19	6	15
	600	32	6	6	110	6	54
dnf4	200	64	10	13	183	9	80
	600	64	10	25	1997	10	506
mux11	200	32	8	14	39	8	33
	600	32	8	20	208	8	99
par5_32	200	32	16	15	49	15	21
	600	32	16	40	363	41	111

Table 3.2: Experiments using logic gates.

Chapter 4

Multi-Level Boolean Networks

4.1 Related Work

Although the use of threshold gates instead of basic Boolean gates extends the number of concepts that can be efficiently represented by compact two-level networks, many concepts of interest remain unrepresentable in compact form. Many of these concepts can be represented by polynomial size multi-level Boolean networks.

The choice of multi-level Boolean networks instead of threshold gate networks as the hypothesis representation scheme has other advantages. Although threshold gate networks can be implemented using standard digital technologies, for many applications this approach is expensive and inefficient. Pulse stream modulation [63] is one possible alternative, but is limited to a relatively small number of neurons and becomes slow if high precision is required. Dedicated boards based on DSP processors can achieve very high performance and are very flexible but may be too expensive for some applications. Applications that require high speed and/or compact hardware implementations can benefit from an approach based on Boolean networks because the speed and compactness of digital implementations is still unmatched by its analog counterparts [12, 86]. Additionally, many alternatives are available to designers that want to implement Boolean networks, from full-custom design to field programmable gate arrays. This makes the digital alternative more cost effective than solutions based on analog designs.

The number of practical algorithms that can be used to perform synthesis of multi-level Boolean networks from examples is limited. Conceptually, gradient-descent neural network algorithms like the ones described in the previous chapter could be adapted for

this task by including penalty terms that would force the functions implemented by each gate to be simple Boolean functions. In practice, such an approach is of limited interest because the additional penalty terms make the optimization problem more difficult and the utility of these algorithms would still be restricted to the selection of node functions in a fixed architecture.

On the other hand, extensive work has been done on logic synthesis algorithms for multi-level Boolean networks [16]. This algorithms can be used, in principle, to select a multi-level Boolean network that is of minimal size and is consistent with the incompletely specified function defined by the training set. However, many of the techniques developed for multi-level logic synthesis make little use of the extra degrees of freedom allowed by the presence of very large don't care sets and are therefore not applicable to this problem. In fact, in inductive inference, the value assigned by the minimum realization to points in the don't care set of the target function is the most important result of the algorithm. Techniques that allow multi-level logic synthesis systems to make a better use of the don't care set were studied and implemented in systems like SIS [7, 87] but they require too much computation time for this application and are not always effective. The results and computation times obtained using these algorithms are shown in section 4.5 and compared with the ones obtained using the greedy approach described in this chapter.

The algorithms for the adaption of decision trees by Armstrong and Gecsei [4] are also very limited in the type of transformations they can perform in the structure of the network and will work only if the structure is well tuned to the problem at hand.

The algorithms described in this chapter derive both the architecture of the network and the functions implemented by each node. Furthermore, they can be applied to the minimization of (2.15) and can therefore handle problems where the labeling of the instances is made in a non-deterministic way.

4.2 Encoding Multi-Level Boolean Networks

Recall that a single output Boolean network can be represented by a directed acyclic graph where each node implements a simple Boolean function. The node with zero out-degree is the output node and the nodes with zero in-degree are the input nodes.

Given these constraints, it is possible to use the general graph encoding scheme described in section 2.7.1 to encode a general Boolean network. However, this encoding has

to be modified to take into account the three following particular characteristics of these graphs.

1. The graph has to be traversed in the reverse direction of that used in the encoding scheme of section 2.7.1 because otherwise not all nodes will be reached.

2. The description of each node needs to include the number of input variables to that node.

3. The description of each node needs to include the function implemented by that node.

The encoding scheme used assumes that a maximum value of the support of the functions is known. This is known in advance because it is an input parameter for the algorithm and can therefore be considered common knowledge. The encoding scheme for Boolean networks is, therefore, the following:

- A node that was never visited before is encoded starting with 1 followed by an encoding of k, the number of inputs to the node, followed by 2^k bits that describe the function implemented by the node, followed by the encodings of the k input nodes.

- A node that was visited before is encoded starting with 0 followed by a reference to the already described node. The N input nodes are assigned the first N binary codes and are considered visited from the start.

4.3 Global Optimization Using Local Modifications

The algorithm used belongs to a class of algorithms that use local transformations to change the Boolean network in such a way that a minimum of the cost function is reached. Algorithms like the metropolis algorithm [58] or simulated annealing [49] use such an approach. In this case, evaluating moves by computing directly the net change that a move causes on expression (2.15) does not work well and a related variable, the mutual information between network output and the label vector is used as a proxy for the total description length. Before a discussion of why this approach is required, a definition of the local changes that can be applied is needed. Such local changes are usually called moves, as they move the solution to a neighbor point in solution space.

4.3.1 Applying Incremental Changes to a Boolean Network

A list of nodes in the Boolean network is identified as the active list, $L_{\text{act}} = (n_{a_1}, \ldots n_{a_l})$. The l nodes in L_{act} can (potentially) assume any one of the 2^l possible input combinations and each one of these combinations corresponds to one possible value of a multi-valued variable. The objective is to obtain a network with a single output that minimizes (2.15).

Given an active list L_{act}, and a maximum value for the number of variables in the support of newly created functions, M_{sup}, the following three types of local modifications (or moves) can be applied to the Boolean network. In the following illustrations, nodes in the active list are marked with an arrow:

1. Replace k nodes in the active list, $n_{a_1}, \ldots, n_{a_k}$ by $k-1$ new nodes that correspond to $k-1$ new functions of k variables, $n'_{a_i} = f'_i(n_{a_1}, \ldots, n_{a_k})$. This move decreases the number of nodes in the active list by one.

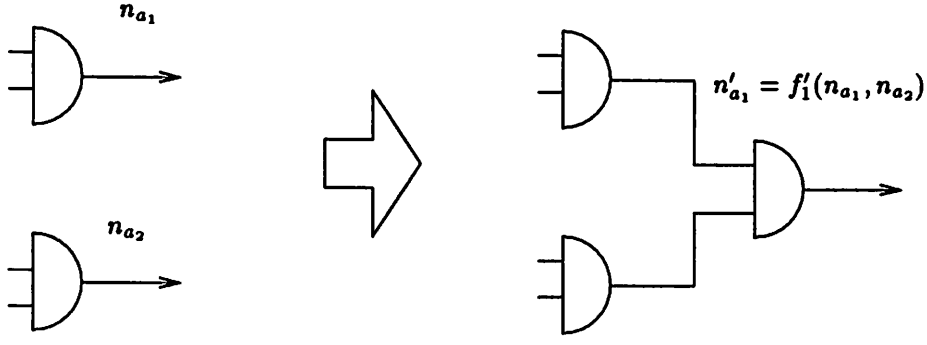


Figure 4.1: Merging two nodes in the active list

2. Replace one node in the active list, n_{a_i}, by a new node that corresponds to a function of s variables $n'_{a_i} = f(n_{a_i}, n_{j_1}, \ldots n_{j_{s-1}})$. This move does not change the number of nodes in the active list.

3. Add one node to the active list increasing the number of nodes in the active list.

Before an algorithm that uses these local changes to obtain a Boolean network that minimizes (2.15) is described, a detour is needed to introduce the concept of mutual information between two variables.

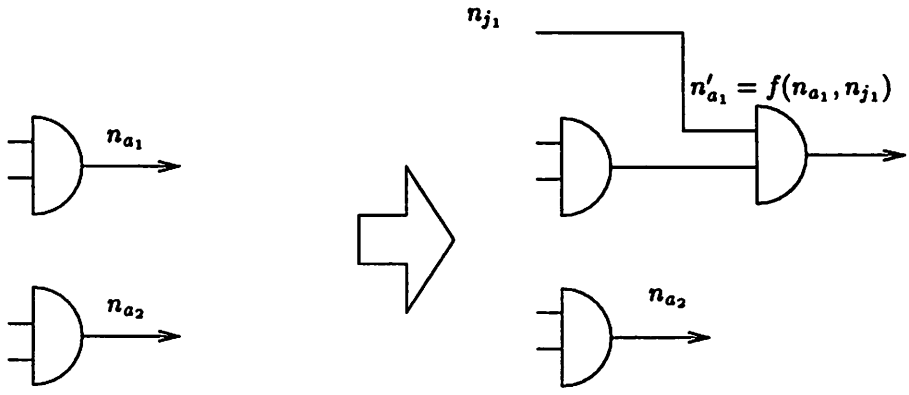


Figure 4.2: Replacing an existing node by a new function

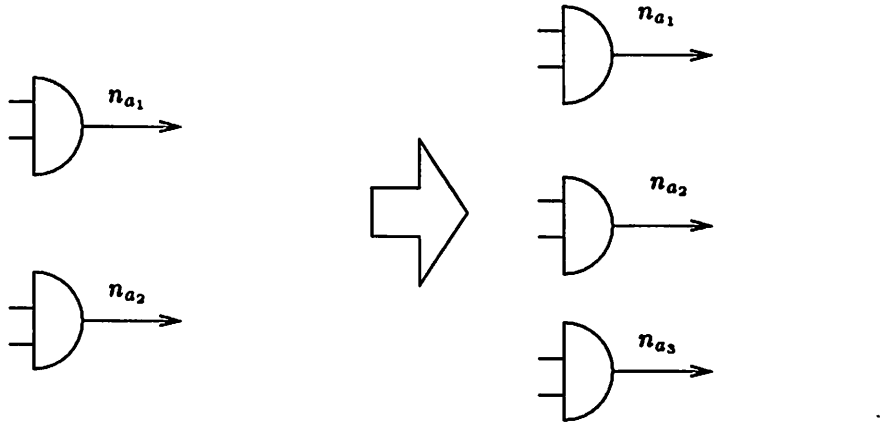


Figure 4.3: Adding a new node to the active list

4.3.2 Entropy and Mutual Information

Let variable V take the values $\{v_1, v_2, ..., v_n\}$ with probabilities $p(v_1), p(v_2), ..., p(v_n)$. The entropy of V is given by

$$H(X) = -\sum_j p(v_j) \log p(v_j) \tag{4.1}$$

and is a measure of the uncertainty about the value of V. The uncertainty about the value of V when the value of another variable Y is known is given by

$$H(V|Y) = -\sum_i p(y_i) \sum_j p(v_j|y_i) \log p(v_j|y_i) \tag{4.2}$$

The amount by which the uncertainty of V is reduced when the value of variable Y is known,

$$\mathcal{I}(Y,V) = H(V) - H(V|Y) \tag{4.3}$$

is called the mutual information between Y and V.

In this case, the objective is to compute $\mathcal{I}(L_{\text{act}}, t)$, the mutual information between the nodes in L_{act}, viewed as a multi-valued variable, and the value of the labels in the training set t. Each combination of the values of the nodes in L_{act} defines a subset of the instances in the training set. For each of these subsets, the rightmost sum in (4.2) is given by:

$$\frac{n}{n+p} \log(\frac{n}{n+p}) + \frac{p}{n+p} \log(\frac{p}{n+p}) \tag{4.4}$$

where n and p are the numbers of negative and positive instances observed in each of these sets. The weighted sum of these quantities is then easily computed to obtain the conditional entropy and the corresponding mutual information.

4.3.3 Hill Climbing on Mutual Information

It is now possible to analyze why direct hill climbing[1] on the value of (2.15) cannot be performed using the set of moves defined in section 4.3.1. Suppose the target concept is defined by the function $f = x_1x_2x_3x_4$, the training set is generated in accordance with the uniform probability distribution in $\{0,1\}^4$ and assume that the active list consists of only

[1]This designation is somewhat misleading since a minimum of the function is desired. However, the term hill climbing is commonly used independently of whether a maximum or a minimum of the function is desired.

one node, $L_{\text{act}} = \{x_1\}$. Replacing the active node x_1 by $n_5 = x_1 x_2$ (using operation 2) is a move in the right direction. Clearly, one of the effects of this move is to increase the first term in (2.15), the hypothesis complexity. However, even though n_5 implements a function that is more similar to the target function, it does not reduce the number of exceptions, and therefore the second term in (2.15) does not decrease. This happens because even if an instance causes the value of n_5 to be 1, it is still more likely to be a negative instance than a positive one and the additional information cannot be used to reduce the complexity of the description of the exceptions.

For this reason, the algorithm performs hill-climbing on the value of the function $\mathcal{I}(L_{\text{act}}, t)$. The algorithm incrementally changes the network in a greedy way, trying, at each iteration, to decrease the number of nodes in L_{act} and to increase the value of $\mathcal{I}(L_{\text{act}}, t)$. Every time a network with a single output is obtained, it is checked to see if it corresponds to a smaller value of (2.15) than the previous solution. The function $\mathcal{I}(F_{\text{act}}, t)$ is therefore used only as a proxy for the actual function that is to be minimized because it represents a finer estimate in the space of solutions defined by the moves used.

4.4 An Algorithm for Hill-Climbing on Mutual Information

4.4.1 Selecting the Best Move

The objective is to obtain a network with a single output that minimizes (2.15) by performing hill-climbing on the mutual information. These two objectives are conflicting because changes of type 3 increase $\mathcal{I}(L_{\text{act}}, t)$ but also increase the number of nodes in L_{act} while changes of type 1 decrease the number of nodes in L_{act} but also decrease $\mathcal{I}(L_{\text{act}}, t)$.

The solution adopted is to apply changes of type 1 only if the decrease in $\mathcal{I}(L_{\text{act}}, t)$ is not statistically significant, to apply changes of type 2 if any increase in $\mathcal{I}(L_{\text{act}}, t)$ takes place and, finally, to apply changes of type 3 if these two options fail.

These operations are all performed for a fixed value of M_{sup}, the number of variables in the support of newly created functions. At some point, however, the number of variables in L_{act} may be to large to obtain statistically significant tests. In these case, the only solution is to increase the value of M_{sup}, the maximum number of variables in newly created functions and restart the process. The pseudo-code in figure 4.4 describes the algorithm. The algorithm takes as input two constants, M_{L} and M_{max} that define respectively

the maximum number of nodes allowed in the active list and an upper limit on the maximum number of nodes allowed in the support of a newly created function. The choice of these constants is based only on runtime considerations. Typically, no more than 4 nodes are required in the active list and the search for functions with more than 3 variables is too expensive to perform.

```
MinNetwork(M_L, M_max)
  M_sup := 2
  repeat
    L'_act := ApplyChangeType1(L_act, M_sup)
    if LossIsNotSignificant(L'_act, L_act) Information loss is not statistically significant
      L_act := L'_act                                              Accept move
      M_sup := 2
      continue
    L'_act := ApplyChangeType2(L_act, M_sup)
    if I(L'_act, t) > I(L_act, t)          There is an increase in mutual information
      L_act := L'_act                                              Accept move
      M_sup := 2
      continue
    if |L_act| < M_L          The number of variables in L_act does not exceed maximum
      L'_act := ApplyChangeType3(L_act, M_sup)
      L_act := L'_act                                              Accept move
      M_sup := 2
      continue
    M_sup := M_sup + 1
  until M_sup = M_max ∨ |L_act| = 1
```

Figure 4.4: Minimal network search algorithm.

4.4.2 Selecting a Discriminating Function

Moves of type 3 require only the addition of a new variable to the active list and this search can be performed in a relatively efficient way. On the other hand, the application of a type 1 or type 2 move require the search for a new function. This procedure can be time consuming. In fact, not only all combinations of existing variables need to be examined but the best function of a given combination of variables needs to be selected as the candidate one.

More specifically, the application of moves of type 1 and type 2 requires the solution of the following problem: given two sets of nodes, $S_1 = \{n_{i_1}, \cdots, n_{i_s}\}$ and $S_2 = \{n_{j_1}, \cdots, n_{j_r}\}$, select a set of functions $\{f_1 \ldots f_{s-1}\}$ that takes as inputs the variables in S_1 and maximizes

$$\mathcal{I}(\{f_1 \ldots f_{s-1}\} \cup S_2, t) \tag{4.5}$$

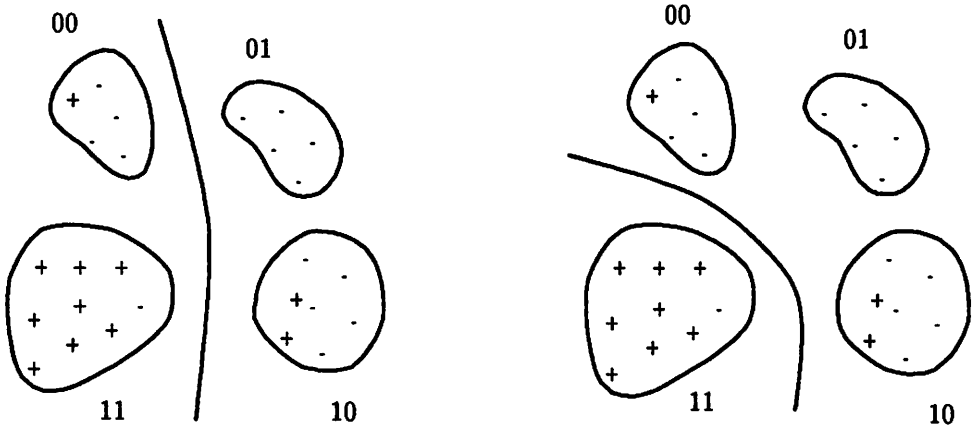
Any Boolean function f can be viewed as a partition of the input space. In this case, the function should depend only on the variables $\{n_{i_1}, \cdots, n_{i_s}\}$. Furthermore, the value of this function is important only for the points in the input space present in the training set. To obtain this function, the points in the training set that share the same values of the variables in S_1 are joined into the same cluster. There will be 2^s such clusters, although some of them may be empty. Any partition of these clusters into 2^{s-1} sets corresponds to a given value of (4.5) and to a specific choice of $(f_1, \cdots, f_{s-1})$. The computation of the value of expression (4.5) is fast because there are only 2^s such clusters and s is limited to be no larger than a user imposed constant, M_{sup}. A partition of these clusters that maximizes locally (4.5) is then obtained using the Kerninghan-Lin partitioning algorithm [48]. This algorithm selects a good partition with respect to some cost function by swapping objects (in this case by swapping clusters) between the two sides of the partition and evaluating the net effect of such swaps in the target cost function. This procedure will be illustrated with a concrete example.

Assume that the training set defined over a set of variables $\{x_1, x_2, x_3 \ldots\}$ has the distribution of positive and negative examples shown in table 4.1 when the training set is projected on the subspace $\{x_1, x_2\}$. This projection is obtained by simply ignoring the values of the remaining variables. The original entropy of t, the variable that defines the label in the training set is obtained applying expression (4.1) and is equal, for this example, to 0.983. Now, assume that the active variables are $L_{\text{act}} = \{x_1, x_2\}$. The mutual

Table 4.1: Distribution of positive and negative examples

x_1	x_2	Pos	Neg
0	0	1	4
0	1	0	5
1	0	2	5
1	1	8	1

information between $L_{act} = \{x_1, x_2\}$ and t, $\mathcal{I}(L_{act}, t) = 0.438$. Now a move of type 1 is to be performed and the possibility under consideration is to replace x_1 and x_2 for the best function $f(x_1, x_2)$. This corresponds to the selection of sets S_1 and S_2 as $S_1 = \{x_1, x_2\}$ and $S_2 = \emptyset$. This is equivalent to the selection of a partition of the 4 clusters in figure 4.5 into 2 classes. Figure 4.5 shows two possible partitions and the corresponding values of mutual information. Clearly, the selection of the function $f_2 = x_1 x_2$ provides the solution that



$f_1 = x_1 \oplus x_2$ $\qquad$ $f_2 = x_1 \wedge x_2$

$\mathcal{I}(f_1, t) = 0.177$ $\qquad$ $\mathcal{I}(f_2, t) = 0.369$

Figure 4.5: Partitions and corresponding values of mutual information

maximizes (4.5). The selection of such a partition when the set S_2 is non void is similar, although the computation of the value of (4.5) for each possible partition is slightly more elaborate. When S_1 has $k > 2$ variables the computation of $k - 1$ functions needed to

perform local changes of type 1 is equivalent to the selection of a 2^{k-1}-way partition of the clusters.

4.4.3 Evaluating the Statistical Significance of Information Loss

The algorithm in figure 4.4 only applies moves of type 1 like the one used in the previous example if the loss in mutual information is not statistically significant. This is necessary because every time k variables in the active list are merged into $k-1$ variables, it is likely that there will be an information loss. This will only be false if the proportions of positive to negative instances in each of the merged bins are exactly the same.

This information loss may not be statistically significant because it may be caused by fluctuations introduced by the sampling process that generated the training set. In the example in table 4.1 the characteristic function of the target concept may be equal to $f = x_1x_2$ but the presence of noise may have corrupted some labels. Alternatively, the target function may have a weak dependence on other variables (e.g., $f = (x_1x_2) \oplus g(x_7x_8x_9x_{10})$).

In both cases, a practical approach is to perform the merging by applying the move of type 1 only if the proportions of positive and negative instances in the merged bins are not significantly different. This condition is tested using a chi-squared test.

The application of the chi-squared test is also illustrated using the example presented in the previous section. Assume that the merging selected by the function f_2 in figure 4.5 is to be tested for statistical significance. If the real function is given by $f = x_1x_2$, (possibly with other dependences on other variables) then the different proportions of negative and positive examples observed in the bins labeled 00, 01 and 10 are only due to random fluctuations. If this is the case, the expected proportion of positive and negative instances in each of these bins is the same and is possible to compute the deviations $o_i^+ - e_i^+$ on the number of positive instances observed for each bin, where o_i^+ is the observed number of positive instances and e_i^+ is the expected value. The same is true for the negative instances. The statistic

$$\chi^2 = \sum_i \frac{(o_i^+ - e_i^+)^2}{e_i^+} + \frac{(o_i^- - e_i^-)^2}{e_i^-} \tag{4.6}$$

has, under reasonable assumptions, a chi-squared distribution with $k = 2$ degrees of freedom. (The number of degrees of freedom is given by the number of clusters merged minus 1).

The probability of the deviations observed observed under the assumption that they are due only to random fluctuations can then be extracted from the tables of the chi-

squared distribution or computed directly by evaluating $\Gamma(k/2, \chi^2/2)$ where $\Gamma(x, y)$ is the gamma function.

Table 4.2: Computation of the chi-squared statistic

Bins	1	2	3	Total
Observed				
o_i^+	1.00	0.00	2.00	3.00
o_i^-	4.00	5.00	5.00	14.00
Total	5.00	5.00	7.00	17.00
Expected				
e_i^+	0.88	0.88	1.24	
e_i^-	4.12	4.12	5.76	

The computation of the values required for the evaluation of the chi-squared statistic given by expression (4.6) is exemplified in table 4.2. The computed value, $\chi = 1.67$ means that the merging should be accepted at a significance level β of 0.05 because it is smaller than the tabulated value for $\chi_{2,1-0.05}$, 5.991.

4.4.4 The Hill-Climbing Algorithm Illustrated

Figures 4.6 and 4.7 exemplify how the algorithm works when learning the simple Boolean function $f = x_1x_2 + x_3x_4x_5$ assuming that a complete training set is given. In this

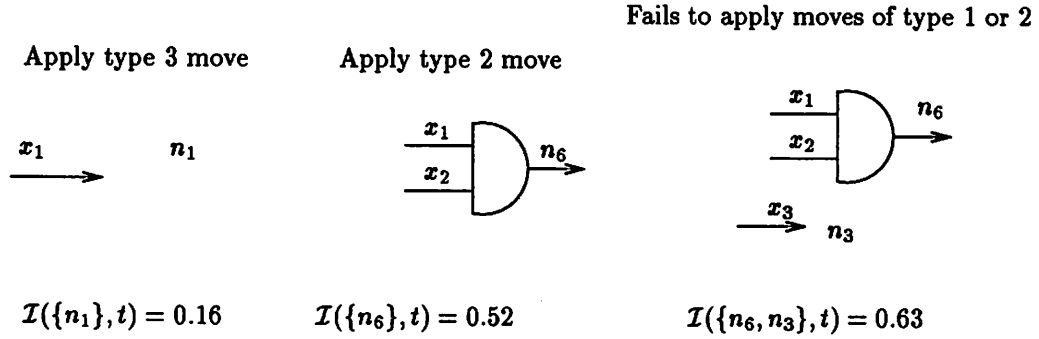


Figure 4.6: Example of a run of the algorithm, part 1

example, the value of M_{sup} is always at 2 and, therefore, only 2 input Boolean functions are generated. In figure 4.6, the algorithm generates the partial function $g_1 = x_1x_2$ and reaches a point where it cannot improve the mutual information by applying changes of type 2. It then adds a new node to the active list and builds an auxiliary function, $g_2 = x_3x_4x_5$. It is implicit that many moves of other type (like, for instance, merging nodes n_7 and n_6) were considered but did not pass the chi-squared test described above. An operation of type 1 can only be performed with success when the function $g_3 = x_3x_4x_5$ is obtained. The mergings implied by a move of type 1 now pass the statistical significance test and are applied to yield the final result. It is important to note that, in general, the algorithm

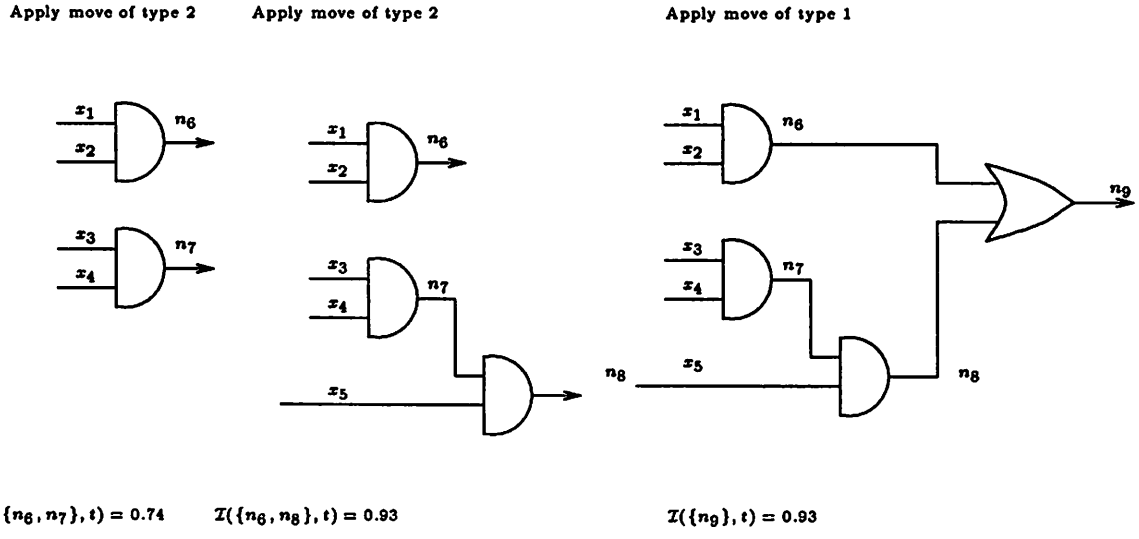


Figure 4.7: Example of a run of the algorithm, part 2

will not aim straight for the right solution. Instead, it will build functions that are closer and closer to the target function and are increasingly more complex. In particular, the statistical significance test is bound to fail in both directions, once in a while. This means that good mergings will be sometimes rejected and bad ones accepted. This, however, does not block the algorithm from selecting a solution that is a local minimum, under the set of moves selected, to expression 2.15.

4.5 Experimental Evaluation

An objective evaluation of the merits of this algorithm is hard to accomplish because no other algorithms developed specifically for this purpose have been proposed to date.

However, if the algorithm is run with the restriction that it should minimize expression (2.15) with a very small value of α, then it will obtain a network with an output that will always coincide with the labels present in the training set. The implementation made of these algorithms in the form of a C++ program, `muesli`, can be used in this exact mode and was used in the comparisons described in this section.

In these conditions, it is possible to compare the performance of the algorithm with standard logic synthesis techniques, and, in particular, multi-level logic synthesis algorithms. The two algorithms selected are both based on the SIS system [15] developed at Berkeley but use two different scripts that give very different results. The *rugged* script uses the computation of local don't care sets for each node to achieve a simpler final network. This script takes usually longer than the other script used, the *algebraic* script because the computation of the local don't care sets is computationally intensive. The *algebraic* script, on the other hand, uses mostly faster algebraic manipulations to minimize the resulting network and sacrifices, in general, solution quality for speed.

4.5.1 Experimental Setup

To evaluate the relative performance of the algorithms, a set of 12 functions was selected, all of them known to accept compact multi-level implementations. These functions are described in appendix B.

The first 4 functions were designed to be specially simple and are defined over Boolean spaces with a small number of variables. The last 8 functions have been proposed in the machine learning literature [69] and are relatively more complex. For each of these functions, training sets of increasing sizes were randomly generated and labeled in accordance with the function being learned.

The three algorithms were run on each problem for a maximum of one hour of CPU time in a DEC/Alpha machine and a memory usage limited to 140 Megabytes.

4.5.2 Results Analysis

Tables 4.3, 4.4 and 4.5 describe the results obtained for each of the functions and training set sizes.

There are some points worth noting in these results. First, the *muesli* algorithm obtained results that were always competitive with the best alternatives but using an amount of CPU time much smaller. In fact, for the larger problems, the algorithms based on the use of logic synthesis techniques were unable to complete in the time and memory allocated for the task while the algorithms described in this chapter faced no serious problems deriving relatively compact representations. The *rugged* script always gave better results than the *algebraic* script, both in terms of the number of literals in the final result and the total amount of CPU time. However, even this script is unable to solve the larger problems listed in tables 4.4 and 4.5. The superior speed of the *rugged* script in these problems came as a surprise and deserves further study.

These results show that logic synthesis algorithms that are highly effective in the type of problems commonly encountered in the logic synthesis field are not well adapted to the case where the target functions are specified by a sample of the *on* and *off* sets. This is probably due to the fact that the computation of the don't care sets for each node becomes too expensive to perform and this don't care set is hard to use effectively.

The amount of CPU time used by the logic synthesis algorithms makes them unusable for all but the simplest learning problems. On the other hand, the algorithms described in this chapter were applied with success to problems much larger than the ones studied here. For example, in chapter 7 they were used to learn several functions of 256 variables with training set sizes with more than 50000 instances. Even if the logic synthesis approach could be modified by careful tuning of the scripts to handle the problems studied in this chapter, it is unlikely that they can be applied with any success to problems of that order of magnitude.

Concept	Training Set		# literals			Cpu time		
	# inputs	Size	muesli	rugged	algebraic	muesli	rugged	algebraic
heel9	9	50	26	43	64	0.8	0.7	0.5
		100	23	14	80	0.5	1.2	0.8
		150	18	13	110	0.6	1.9	1.3
		200	18	14	106	0.5	2.0	1.2
		250	13	13	116	0.4	1.7	1.5
		300	13	13	121	0.3	2.2	1.4
		350	13	13	122	0.3	2.4	1.3
		400	13	14	131	0.3	2.2	1.2
		450	13	13	120	0.4	2.4	1.2
		500	13	14	117	0.3	2.3	1.2
sm12	12	100	75	86	175	3.1	6.2	4.2
		200	114	139	301	6.3	24.5	23.8
		300	90	79	401	5.5	59.6	75.2
		400	41	76	458	1.5	66.4	199.9
		500	57	38	505	2.1	71.2	269.0
		600	41	37	537	1.6	66.9	435.6
		700	43	36	600	2.3	88.8	696.3
		800	40	35	655	1.6	167.3	1006.2
		900	51	43	653	3.0	185.1	1174.6
		1000	70	38	681	2.5	193.5	1128.4
heel18	18	100	50	91	-	3.3	53.6	-
		200	92	134	-	6.4	185.4	-
		300	25	164	-	1.3	494.0	-
		400	25	80	-	1.3	1429.2	-
		500	25	47	-	1.5	1092.7	-
		600	25	119	-	1.8	2725.7	-
		700	25	62	-	1.5	3343.8	-
		800	25	-	-	1.5	-	-
		900	25	-	-	1.8	-	-
		1000	25	-	-	1.8	-	-
str18	18	100	23	34	86	1.1	9.3	0.9
		200	65	84	165	6.6	42.7	4.5
		300	130	104	245	12.4	1082.4	14.6
		400	119	-	314	11.6	-	34.0
		500	157	117	-	12.9	2602.9	-
		600	137	187	-	14.8	544.4	-
		700	29	150	-	2.6	779.7	-
		800	58	234	-	4.5	1003.8	-
		900	90	115	-	8.0	1263.4	-
		1000	126	236	-	11.5	1898.1	-

Table 4.3: Minimal Boolean networks obtained by different techniques, part 1

Concept	Training Set		# literals			Cpu time		
	# inputs	Size	muesli	rugged	algebraic	muesli	rugged	algebraic
mux6	16	100	43	42	350	2.5	14.9	31.4
		200	28	19	-	1.4	90.4	-
		300	61	31	-	2.3	181.7	-
		400	38	44	-	1.6	370.4	-
		500	35	23	-	1.4	964.3	-
		600	32	30	-	1.7	813.4	-
		700	42	77	-	2.5	2765.6	-
		800	34	15	-	1.8	2185.6	-
		900	50	21	-	2.4	2931.3	-
		1000	51	-	-	3.4	-	-
mux11	32	100	65	91	-	4.8	135.2	-
		200	149	243	-	15.5	1070.2	-
		300	303	420	-	50.6	2407.5	-
		400	227	-	-	43.2	-	-
		500	482	-	-	109.8	-	-
		600	522	-	-	203.5	-	-
		700	162	-	-	24.1	-	-
		800	111	-	-	16.8	-	-
		900	114	-	-	12.6	-	-
		1000	194	-	-	34.7	-	-
par4_16	16	100	107	106	371	13.5	366.3	44.5
		200	242	171	-	45.3	89.9	-
		300	431	39	-	96.4	219.4	-
		400	662	463	-	263.4	531.5	-
		500	12	316	-	17.2	1153.3	-
		600	121	71	-	31.9	1137.1	-
		700	12	24	-	14.0	1782.6	-
		800	12	29	-	1.2	2761.6	-
		900	-	36	-	-	3342.4	-
		1000	12	-	-	5.9	-	-
par5_32	32	100	85	121	-	11.1	119.5	-
		200	168	248	-	34.6	1260.2	-
		300	313	439	-	68.9	3168.7	-
		400	-	-	-	-	-	-
		500	610	-	-	197.8	-	-
		600	712	-	-	269.4	-	-
		700	847	-	-	453.1	-	-
		800	-	-	-	-	-	-
		900	-	-	-	-	-	-
		1000	-	-	-	-	-	-

Table 4.4: Minimal Boolean networks obtained by different techniques, part 2

Concept	Training Set		# literals			Cpu time		
	# inputs	Size	muesli	rugged	algebraic	muesli	rugged	algebraic
dnf1	80	100	27	-	-	3.4	-	-
		200	64	-	-	6.8	-	-
		300	94	-	-	13.8	-	-
		400	156	-	-	24.4	-	-
		500	235	-	-	42.5	-	-
		600	198	-	-	41.1	-	-
		700	342	-	-	87.1	-	-
		800	389	-	-	98.6	-	-
		900	413	-	-	145.0	-	-
		1000	415	-	-	138.1	-	-
dnf2	40	100	34	-	-	2.7	-	-
		200	120	154	-	9.0	1410.8	-
		300	109	-	-	7.0	-	-
		400	140	-	-	12.4	-	-
		500	101	-	-	9.4	-	-
		600	73	-	-	6.3	-	-
		700	125	-	-	10.3	-	-
		800	107	-	-	7.6	-	-
		900	111	-	-	8.7	-	-
		1000	79	-	-	7.6	-	-
dnf3	32	100	67	52	320	2.8	139.9	40.3
		200	81	124	-	5.9	399.2	-
		300	44	111	-	2.9	1352.6	-
		400	100	-	-	7.6	-	-
		500	142	-	-	15.8	-	-
		600	57	-	-	3.4	-	-
		700	53	-	-	2.8	-	-
		800	73	-	-	6.0	-	-
		900	85	-	-	6.6	-	-
		1000	76	-	-	4.8	-	-
dnf4	64	100	55	122	-	5.4	1307.4	-
		200	120	-	-	19.1	-	-
		300	156	-	-	28.1	-	-
		400	165	-	-	28.9	-	-
		500	429	-	-	91.5	-	-
		600	129	-	-	18.3	-	-
		700	148	-	-	23.5	-	-
		800	87	-	-	16.5	-	-
		900	87	-	-	12.6	-	-
		1000	52	-	-	7.1	-	-

Table 4.5: Minimal Boolean networks obtained by different techniques, part 3

Chapter 5

Reduced Ordered Decision Graphs

5.1 Introduction

This chapter describes both exact and heuristic algorithms for the induction of minimal complexity reduced ordered decision graphs from training set data. Decision graphs can be viewed as a generalization of decision trees, a very successful approach for the inference of classification rules [17, 76, 79]. The selection of decision graphs instead of decision trees as the hypothesis representation scheme is important because, even though decision trees can represent any concept, they are not concise representations for some concepts of interest. In particular, the quality of the generalization performed by a decision tree induced from data suffers because of two well known problems: the replication of subtrees required to represent some concepts and the rapid fragmentation of the training set data when attributes that can take a high number of values are tested at a node. Oliver [67] describes in some detail these limitations.

Decision graphs have been proposed as one way to alleviate these problems, but the algorithms proposed to date for the construction of these graphs suffer from serious limitations. Mahoney and Mooney [57] proposed to identify related subtrees in a decision tree obtained using standard methods but reported limited success in the sense that they observed no improvement in the quality of the generalization performed. They used a non-canonical representation of Boolean functions (DNF expressions) to represent the functions

implemented by these subtrees. The non-canonicity of this representation makes is a non-trivial process to identify identical subtrees and that renders this approach impracticable for large decision trees. Oliver [67] proposed a greedy algorithm that performs either a join or a split operation, depending on which one reduces more the description length. He reported improvements over the use of decision trees in relatively simple problems but our experiments using a similar approach failed in more complex test cases because of the greedy nature of the algorithm.

This chapter describes two different approaches for the problem of selecting the RODG of minimal description length. Since the problem is NP-complete [97] both an exact and an heuristic approach are described for this problem.

The exact approach described in section 5.3 works only in the noise free case and selects the RODG that minimizes (2.14) under a fixed ordering of the variables. Although the domain of applicability of the exact solver is limited, this approach is interesting because it is the first formulation of this optimization problem that does not require an explicit search of the solution space as previous approaches do [91].

Section 5.4 describes an heuristic approach that obtains an RODG that minimizes (2.15) and is much faster than the exact one for large examples. Furthermore, it derives a good ordering of the variables together with the RODG structure. The approach differs from the one proposed by Kohavi [50] that also uses RODGs. Although his approach performs well for small problems, it requires far too much computation to be applicable to any problems of reasonable size. Other heuristic algorithms that can be used for the selection of compact RODGs compatible with the training set have been proposed before in the logic synthesis literature [92] but the quality of the results obtained makes them ineffective for the type of problems found in machine learning. They are, however, useful as a generator of the initial RODG, as described in section 5.4.1.

This work draws heavily on techniques developed by other authors in the machine learning and logic synthesis fields. From machine learning, I use many of the techniques developed for the induction of decision trees described in [76] and [79]. Also used are the constructive induction algorithms proposed by Pagallo and Haussler [69] and further developed in [70] and [101]. From the logic synthesis field, the use of the vast array of techniques developed for the manipulation of RODGs as canonical representations for Boolean functions [18, 13] and the variable reordering algorithms studied in [26] and [83] are critically important. For the benefit of readers not familiar with the use of RODGs as a tool for

the manipulation of Boolean functions, appendix A gives an overview of the techniques available and their relation to this work.

5.2 Definitions

5.2.1 Decision Graph Nodes and Functions

Recall that an decision graph is a rooted, directed, acyclic graph where each node is labeled with the name of one variable. and every non-terminal node n_i has one *else* and one *then* edge that point to the children nodes, n_i^{else} and n_i^{then}, respectively.

Any minterm z in the input space induces a unique path in an RODG defined in the following way: start at the root and take, at each node, the *else* or the *then* edge according to the value assigned by minterm z to the variable that is the label of the current node until a terminal node is reached. An RODG corresponds to the completely specified Boolean function f that has all the minterms in f_{on} (and only these) inducing paths in the RODG that terminate in n_o. For a fixed ordering of the variables, the RODG for a given Boolean function is unique. This implies that RODGs are a canonical representation of Boolean functions and the notation n_i will be used to denote both the node in the RODG and the Boolean function it corresponds to.

The level of a node n_i, $\mathcal{L}(n_i)$ is the index of the variable tested at that node under the specific ordering used. The level of the terminal nodes is defined as $N+1$, where N is, as before, the number of input variables. The maximum level of a set s of nodes, $\mathcal{L}_{\max}(s)$ is the maximum level of all the nodes in s. A decision graph is called complete if all edges starting at level i terminate in a node at level $i+1$ or in a terminal node.[1] The level of a function h, $\mathcal{L}(h)$, is defined as the level of a RODG node that implements h. If n_i is a node in the RODG and z a minterm, $n_i(z)$ will be used to denote both the value of function n_i for minterm z and the terminal node that z reaches when starting at n_i. This notation is consistent because the two terminal nodes stand for the constant functions 0 and 1.

A 3 Terminal RODG (3TRODG) is defined in the same way as an RODG in all respects except that it has three terminal nodes : n_z, n_o and n_x. A 3TRODG F corresponds to the incompletely specified function f that has all minterms in f_{off},f_{dc} and f_{on} terminate in n_z, n_x and n_o, respectively.

[1] A complete decision graph will not, in general, be reduced.

5.2.2 Decision Trees

A decision tree is a special case of an unordered decision graph where the underlying graph structure is restricted to be a tree, i.e., has no re-converging paths, and there are more than two terminal nodes. The terminal nodes in a decision tree belong two one of two sets: S_z or S_o. A decision tree corresponds to the completely specified Boolean function f that has all the minterms in f_{on} (and only these) inducing paths in the RODG that terminate in terminal nodes that belong to S_o.[2]

Consider now a decision tree or a decision graphs defined over the domain D. Let $n_0, n_1, ..., n_s$ be the nodes in the graph or tree, v_i denote the variable tested in node n_i of the graph (or tree) , n_i^{then} (the *then* node) denote the node pointed to by the arc leaving node n_i when attribute v_i is 1 and n_i^{else} (the *else* node) denote the node pointed to by the arc leaving node n_i when attribute v_i is 0. Finally, let node n_0 be the root of the graph.

5.2.3 Encoding Decision Graphs

The encoding scheme used is again a variation of the generic graph encoding scheme described in section 2.7.1. That scheme has to be modified only slightly to take into account the fact that, for each node, the index of the variable tested at that node also has to be described. The encoding scheme is, therefore, the following:

- A node that was never visited before is encoded starting with 1 followed by an encoding of the variable tested at that node, followed by the encoding of the node pointed to by the *else* edge, followed by an encoding of the node pointed to by the *then* edge.
- A node that was visited before is encoded starting with 0 followed by a reference to the already described node.

The first node to be described is the root of the graph, and the two terminal nodes are considered visited from the beginning and assigned reference numbers 0 and 1.

For the exact algorithm, it is assumed that each reference uses a fixed number of bits, no matter how many nodes were described at the point that reference is used. This makes the encoding length monotonic in the number of nodes in the graph and makes the

[2]Equivalently, a decision tree can be defined has having only two terminal nodes and have no re-converging paths *except* at these two terminal nodes.

minimization of expression (2.14) equivalent to the minimization of the number of nodes in the graph.

For the heuristic algorithm, the slightly more efficient encoding that only uses $\log_2(r)$ bits for each reference where r is the number of nodes described up to that point is used. This is important because the heuristic algorithm minimizes expression (2.15) and it is more critical to use encodings that are close to the optimum.

5.3 An Exact Minimization Algorithm

Given a fixed ordering, the objective is to derive an RODG that is consistent with the incompletely specified function f defined by the training set and minimizes (2.14) using the encoding scheme described in above. Assume, without loss of generality, that the ordering selected is $(x_1, x_2, ..., x_n)$. Any other ordering can easily be handled by performing a simple variable transformation.

From the 3TRODG that is obtained from the incompletely specified function defined by the (noiseless) training set, a compatibility graph is extracted that describes which nodes in this 3TRODG can be merged. The desired result is obtained by selecting a minimum cardinality closed clique cover for the compatibility graph. This approach is inspired in the standard algorithms used for the minimization of incompletely specified state machines and uses some of the concepts developed for that purpose.

5.3.1 The Compatibility Graph

Previous algorithms [91] for this problem used directly the RODG representation of f_{on} and f_{off}. The exact approach described in this paper works with the 3TRODG F that corresponds to f. F is assumed to be complete. If necessary, F is made complete by adding extra nodes that have the *then* and *else* edges pointing to the same node. The resulting 3TRODG is no longer reduced, but this transformation is required to warrant that the minimum solution is found. Obtaining F from the training set is a simple procedure [18] that can be performed using a standard RODG package like, for instance, the one described in [13].

Definition 2 *Two nodes n_i and n_j in F are compatible*[3] *($n_i \sim n_j$) iff no minterm z exists that satisfies $n_i(z) = n_z \wedge n_j(z) = n_o$ or $n_i(z) = n_o \wedge n_j(z) = n_z$.*

Definition 3 *Two nodes n_i and n_j in F are common support compatible ($n_i \approx n_j$) iff there exists a function h such that $h \sim n_i$ and $h \sim n_j$ and $\mathcal{L}(h) \geq \max(\mathcal{L}(n_i), \mathcal{L}(n_j))$.*

By definition $n_z \not\sim n_o$ and $n_x \approx n_i$, for any node n_i.

The compatibility graph, G, is an undirected graph that contains the information about which nodes in F can be merged. Except for the terminal node n_x, each node in F will correspond to one node in G with the same index. The level of a node in G is the same as the level of the corresponding node in F. Similarly, g_i^{else} and g_i^{then} are the nodes that correspond to n_i^{else} and n_i^{then}.

Graph G is built in such a way that if nodes n_i and n_j are common support compatible then there exists an edge between g_i and g_j. An edge may have labels. A label is a set of nodes that expresses the following requirement: if nodes g_i and g_j are to be merged, then the nodes in the label also need to be merged. There are three types of labels: e, t and l labels. The following two lemmas justify the algorithm by which graph G is built:

Lemma 4 *If $\mathcal{L}(n_i) = \mathcal{L}(n_j)$ then $n_i \approx n_j \Rightarrow n_i^{else} \approx n_j^{else} \wedge n_i^{then} \approx n_j^{then}$.*

Proof : Since F is complete, either the successors are at the same level or at least one of them is a terminal. Therefore, $n_i^{else} \not\approx n_j^{else} \Rightarrow n_i^{else} \not\sim n_j^{else}$ and a minterm z can be selected in such a way that $n_i^{else}(z) \not\sim n_j^{else}(z)$ and $z_{\mathcal{L}(n_i)} = 0$. The existence of this minterm shows that $n_i \not\approx n_j$. A similar argument is true for the *then* branch. Therefore, $n_i^{else} \not\approx n_j^{else} \vee n_i^{then} \not\approx n_j^{then} \Rightarrow n_i \not\approx n_j$.

Lemma 5 *If $\mathcal{L}(n_i) < \mathcal{L}(n_j)$ then $n_i \approx n_j \Rightarrow n_i^{else} \approx n_j \wedge n_i^{then} \approx n_j \wedge n_i^{else} \approx n_i^{then}$*

Proof : If $n_i^{else} \not\approx n_j$ then, for any functions h at level $\mathcal{L}(n_j)$ or higher a minterm z can be selected in such a way that $n_i^{else}(z) \not\sim h(z)$ and $z_{\mathcal{L}(n_i)} = 0$. This minterm shows that $n_i \not\sim h$, thereby showing that $n_i \not\approx n_j$. Identically for the *then* branch. If $n_i^{else} \not\approx n_i^{then}$ then there are minterms w and z such that $n_i^{else}(w) \not\sim n_i^{then}(w) \wedge w_{\mathcal{L}(n_i)} = 0$ and $n_i^{else}(z) \not\sim n_i^{then}(z) \wedge z_{\mathcal{L}(n_i)} = 1$. This minterms can be chosen to differ only in the value of the variable $x_{\mathcal{L}(n_i)}$ and lead to incompatible terminal nodes. Therefore, n_i cannot be equivalent to any function h such that $\mathcal{L}(h) >= \mathcal{L}(n_j)$. These two lemmas justify the creation of G as follows:

[3]This is simply a restatement of the compatibility between functions as defined in section 5.2.

1. Initialize G with a complete graph except for edge (g_z,g_o) that is removed.

2. If $\mathcal{L}(g_i) = \mathcal{L}(g_j)$ then the edge between g_i and g_j has two labels: an e label with $\{g_i^{else}, g_j^{else}\}$ and a t label with $\{g_i^{then}, g_j^{then}\}$. (By lemma 4.)

3. If $\mathcal{L}(g_i) \neq \mathcal{L}(g_j)$ edge (g_i,g_j) has an l label with $\{g_k^{else}, g_k^{then}, g_m\}$ where $\mathcal{L}(g_k) = \min(\mathcal{L}(g_i), \mathcal{L}(g_j))$ and $\mathcal{L}(g_m) = \max(\mathcal{L}(g_i), \mathcal{L}(g_j))$. (By lemma 5.)

4. For all pairs of nodes (g_i, g_j) check if the edge between nodes g_i and g_j has a label that contains $\{g_a, g_b\}$ and there is no edge between g_a and g_b. If so, remove the edge between g_i and g_j. Repeat this step until no more changes take place.

Figure 5.1 shows an example of the 3TRODG F obtained from f defined by the following sets: $f_{\text{on}} = \{011, 111\}$, $f_{\text{off}} = \{010, 110, 101\}$ and the corresponding compatibility graph.

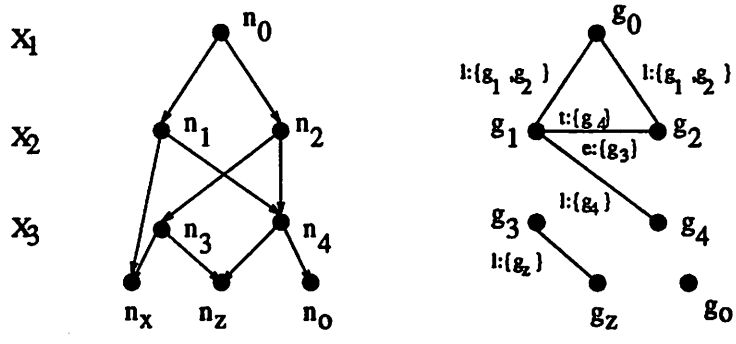


Figure 5.1: The 3TRODG F and the compatibility graph G.

Lemma 6 *If n_i and n_j are common support compatible then G has an edge between nodes g_i and g_j.*

Proof : Follows directly from lemmas 4 and 5 and the algorithm definition.

5.3.2 Closed Clique Covers

To any set s of nodes that is a clique of G there are associated class sets. If the nodes in s are to be merged into one, the nodes in its class sets are also required to be in the same set. Let $s_i = \{g_{i_1}, g_{i_2} \ldots g_{i_w}\}$ be a set of nodes that form a clique in G. The following are the definitions of the e, t and l classes of s_i.

Definition 4 *The e class of s_i, $C_e(s_i)$ is the set of nodes that are in some e label of an edge between a node g_j and g_k in s_i with $\mathcal{L}(n_k) = \mathcal{L}(n_j) = \mathcal{L}_{\max}(s_i)$.*

Definition 5 *The t class of s_i, $C_t(s_i)$ is the set of nodes that are in some t label of an edge between a node g_j and g_k in s_i with $\mathcal{L}(n_k) = \mathcal{L}(n_j) = \mathcal{L}_{\max}(s_i)$.*

Definition 6 *The l class of s_i, $C_l(s_i)$ is the set of nodes that are in some l label of an edge between a node g_j and g_k in s_i with $\mathcal{L}(g_j) \neq \mathcal{L}(g_k)$*

The algorithm that selects the minimum RODG compatible with the original function works by selecting nodes of G that can be merged into one node in the final RODG. If a set s of nodes in G is to be merged into one, they have to be pairwise common support compatible. Therefore, they have to be a clique of G. The objective is to find a set of cliques such that every node in G is covered by at least one clique. However, given the properties of a valid solution, some extra conditions need to be imposed.

Definition 7 *A set $S = \{s_1, s_2 ... s_n\}$ of sets of nodes in G is called a closed clique cover for G if the following conditions are satisfied:*

1. *S covers G : $\forall g_i \in G \, \exists s_j \in S \; : g_i \in s_j$*
2. *All s_k are cliques of G : $\forall g_i, g_j \in s_k \; : (g_i, g_j) \in \text{edges}(G)$*
3. *S is closed wrt the e and t labels :*
   $\forall s_i \in S \, \exists s_j \in S \; : C_e(s_i) \subseteq s_j \; \bigwedge \; \forall s_i \in S \, \exists s_j \in S \; : C_t(s_i) \subseteq s_j$
4. *All sets in S are closed wrt the l labels : $\forall s_i \in S \; : C_l(s_i) \subseteq s_i$*

5.3.3 Generating the Minimum RODG

From a closed clique cover for G, a reduced RODG R is obtained by the following algorithm:

1. For each s_i in S, create an RODG node in R, r_i, at level $\mathcal{L}_{\max}(s_i)$.
2. Let the nodes in R that correspond to sets s_i containing nodes that correspond to terminal nodes in F be the new corresponding terminal nodes of R.

3. Let the *else* edge of the node r_i go to the node r_j that corresponds to a set s_j such that $C_e(s_i) \subseteq s_j$.

4. Let the *then* edge of the node r_i go to the node r_j that corresponds to a set s_j such that $C_t(s_i) \subseteq s_j$.

Lemma 7 *R is an Ordered RODG compatible with F.*

Proof : Since the cover is closed, steps 3 and 4 four are always feasible. Any path in F that leads to a 1 or a 0 will lead to the corresponding terminal node in R. Finally, there will never be edges going upward in R because the node that results from a set s_i is at the lowest level of all the nodes in s_i.

Now, the main result follows. Let $\mathcal{B}$ be the set of all RODGs that represent functions compatible with the incompletely specified function f. Then, the following result applies:

Theorem 1 *The RODG induced by a minimum closed cover for G is the RODG in $\mathcal{B}$ with minimum number of nodes.*

Proof : Given the result in lemma 7 it is sufficient to prove that there exists at least one closed cover of cardinality equal to the size of the minimum RODG in B.

Let U be an RODG in $\mathcal{B}$ with minimum number of nodes k. For each node in U, u_i, create a set s_i such that g_j is in u_i iff $n_j \sim u_i$ and $\mathcal{L}(n_j) \leq \mathcal{L}(u_i)$. Let $S = \{s_1, s_2 ... s_k\}$. We will show that S satisfies all the conditions in definition 7:

1. (S covers G) We show that the assumption that some g_i at level l is not in some set of S leads to a contradiction: let z be one minterm that defines a path in F that goes through n_i and terminates in n_o or n_z. Let Z be the set of all minterms that have the same values as z for $x_1 ... x_{l-1}$. Either one of these minterms will define a path in U that goes through some node u_j in U at a level equal or higher than l or all minterms in Z terminate in some terminal node before reaching any node at level l. In the first case, since $n_i \not\sim u_j$ (by the hypothesis) there exists a minterm in M that will lead to incompatible terminal nodes in U and F, thereby contradicting the assumption that U is consistent with F. In the second case, all such minterms lead to the same terminal node in U. Since g_i is not equivalent to neither terminal node (by the hypothesis), this also implies that U and F are not compatible.

2. (All $s_i \in S$ are cliques) Since each node in s_i is compatible with a completely specified function (r_i) they are all pairwise common support compatible between them and therefore the nodes in s_i are a clique of G.

3. (S is closed wrt the e and t labels) Let u_i be a node in U, $u_a = u_i^{else}$ and $u_b = u_i^{then}$. Let $b_i = \{g_j \in s_i : \mathcal{L}(g_j) = \mathcal{L}_{\max}(s_i)\}$. For each node $g_j \in b_i$ $n_j \sim u_i$ implies $u_a \sim n_j^{else}$ and $u_b \sim n_j^{then}$. Therefore, $C_e(s_i) \subseteq s_a$ and $C_t(s_i) \subseteq s_b$.

4. (S is closed wrt the l labels) Suppose $C_l(s_i) \not\subseteq s_i$. Then, there must be a node n_w such that $g_w \in s_i$ at level $l < \mathcal{L}(r_i)$ such that $n_w^{else} \not\sim u_i$ or $n_w^{then} \not\sim u_i$. Assume the first is true and let $n_w^{else} = n_a$. n_a is not compatible with u_i (or else it would be in s_i) and depends only in $\{x_{l+1}...x_n\}$. Therefore, there exists a minterm m such that $u_i(m) \not\sim n_a(m)$ and $m_l = 0$. This minterm shows that $n_w \not\sim u_i$ which contradicts the hypothesis that g_w is in s_i.

Therefore, S is a closed clique cover for G and it has cardinality k □.

As an example, $S = \{\{n_0, n_1, n_2\}, \{g_4\}, \{g_3, g_z\}, \{g_o\}\}$ is a closed cover for the example depicted in figure 5.1 and induces the RODG R shown on the right side of figure 5.2.

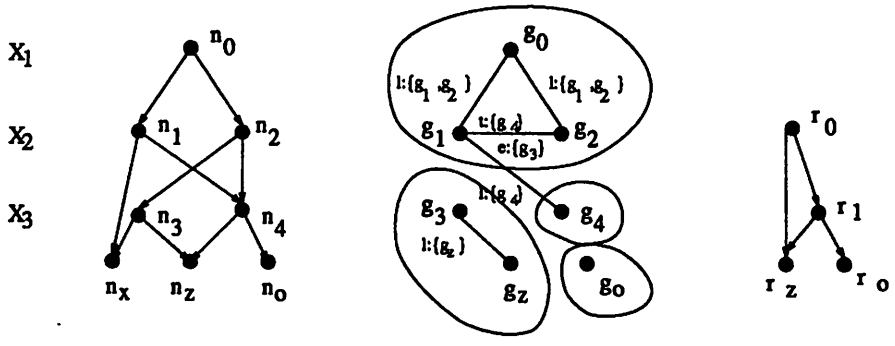


Figure 5.2: The 3TRODG F, the compatibility graph G and a solution R.

Given the result in theorem 1, a minimum RODG can be found by selecting a minimum closed clique cover. This problem is very similar to other problems that have been extensively studied in the logic synthesis community. For example, minimization of incompletely specified finite state machines leads to a similar optimization problem where the selection of a minimum closed cover yields the desired solution.

5.3.4 Direct Solution of the Covering Problem Using Compatibles

Definition 8 *A set s of nodes in G is a compatible set iff s is a clique of G and $C_l(s) \in s$.*

Only compatible sets may be present in a solution to the covering problem. However, one does not need to consider all the compatible sets since a solution consisting only of prime compatible sets is bound to exist.

Definition 9 *A set s_i of nodes in G is a prime compatible set iff s_i is a compatible set of G and there is no other s_j that is a compatible set of G and satisfies: $s_i \subset s_j \ \wedge \ C_e(s_j) \subseteq C_e(s_i) \ \wedge \ C_i(s_j) \subseteq C_i(s_i)$*

Since there exists one minimum solution that consists entirely of prime compatible sets (see [36]) an exact algorithm for solving the minimum closed cover is the following:

1. Generate all prime compatible sets of G.
2. Formulate the closed covering problem as the satisfiability of a Boolean expression and solve it.

The satisfiability problem is NP-complete and can be solved, for some problems, using one the methods presented in the literature. A detailed analysis of these algorithms is outside the scope of this work. The implementation of this algorithm uses the routines described in [38] to solve the minimum covering problem.

5.4 An Heuristic Minimization Algorithm

The heuristic minimization algorithm described in this section derives an RODG that corresponds to a local minimum of (2.15) under the encoding scheme described in section 5.2.3.

The algorithm initializes the RODG using the techniques described in section 5.4.1 and then applies local changes to obtain an RODG of smaller description length.

5.4.1 Generating the Initial RODG

There are several possible ways to generate an RODG that can be used as the starting point for the local optimization algorithm. Experiments have shown that three of

them are particularly effective. The RODG selected as the initial solution is the smaller one of the following three:

- The RODG that realizes the function implemented by a decision tree derived from the training set data using standard decision tree algorithms.
- The RODG that realizes the function implemented by a decision tree defined over a new set of variables obtained using constructive induction techniques.
- The RODG obtained by applying the restrict heuristic to the function obtained by listing all positive instances in the training set.

Each of these three approaches can and do generate sometimes RODGs that are several orders of magnitude larger than the minimum possible. However, by selecting the smaller of the three, it is possible, in many of the problems tried, to obtain a final solution that is reasonably close to the minimum possible. How these three techniques can be used to generate the original RODG is the subject of the next three subsections.

5.4.1.1 Initialization Using Decision Trees

One possible way to initialize the RODG is to obtain a decision tree from the data and to convert the function obtained by the decision tree to RODG form. Several efficient algorithms for the induction of decision trees from data have been proposed in the literature. Since all the attributes are Boolean and we are not concerned with algorithms for pruning[4] the tree, a relatively straightforward algorithm can be used to generate the decision tree. The algorithm used is the same as the one proposed in [76] and uses the concepts of entropy and mutual information as defined in section 4.3.2.

At each point, the decision tree algorithm has to select one variable to be tested at a given node. The variable selected is the one that provides the maximum amount of information about the target class for the examples that reached that node.

After a decision tree is derived, the transformation to a decision graph representation is trivially performed using the facilities provided by the RODG package. Consider a particular node n_i of a decision tree and the subtree rooted at that node (see figure 5.3).

[4]The graph minimization algorithm takes care of the trade-off between training set fit and model complexity.

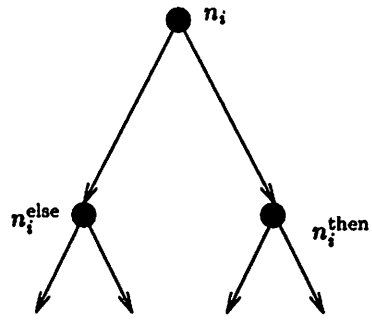


Figure 5.3: A subtree rooted at node n_i

The function implemented by non-terminal node n_i is given by

$$n_i = \begin{cases} n_i^{\text{then}} & \text{if } v_i = 1 \\ n_i^{\text{else}} & \text{if } v_i = 0 \end{cases} \tag{5.1}$$

Expression (5.1) is a recursive definition of the function implemented by any node in the decision tree. The recursion stops when n_i^{else} (or n_i^{then}) is a terminal node. In this case n_i^{else} (or n_i^{then}) corresponds to the constant 0 function or the constant 1 function, depending on the type of the terminal node.

It must be pointed out that the decision graph that corresponds to a particular decision tree is not isomorphic to the tree that served as the starting point. Figure 5.4 shows a decision tree for the function $f = x_1x_2 + x_3x_4$ and the graph that results from applying the above definition, assuming the ordering used is (x_1, x_2, x_3, x_4).

5.4.1.2 Initialization Using a Constructive Induction Algorithm

Constructive induction algorithms create new complex attributes by combining existing attributes in ways that make the description of the concept easier. The algorithm used in *fulfringe* [66], identifies patterns near the fringes of the decision tree and uses them to build new attributes. The idea was first proposed in [69] and further developed in [70]. Another algorithm of this family, *dcfringe* [101] identifies the patterns shown in the first two rows of figure 5.5. These patterns correspond to 8 Boolean functions of 2 variables. Since there are only 10 distinct Boolean functions that depend on two variables[5], it is natural to

[5]The remaining 6 functions of 2 variables depend on only one or none of the variables.

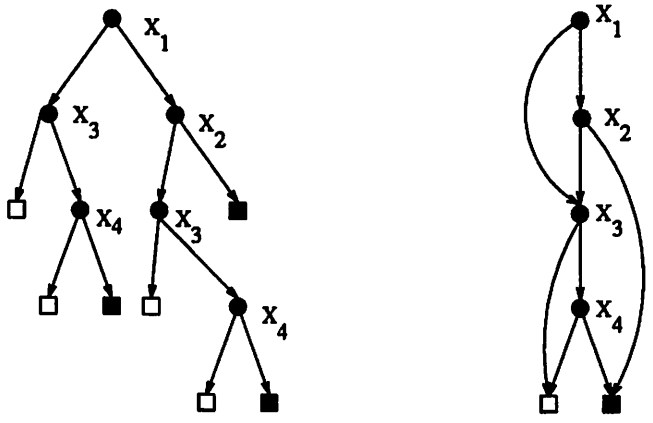


Figure 5.4: A decision tree and the corresponding decision graph

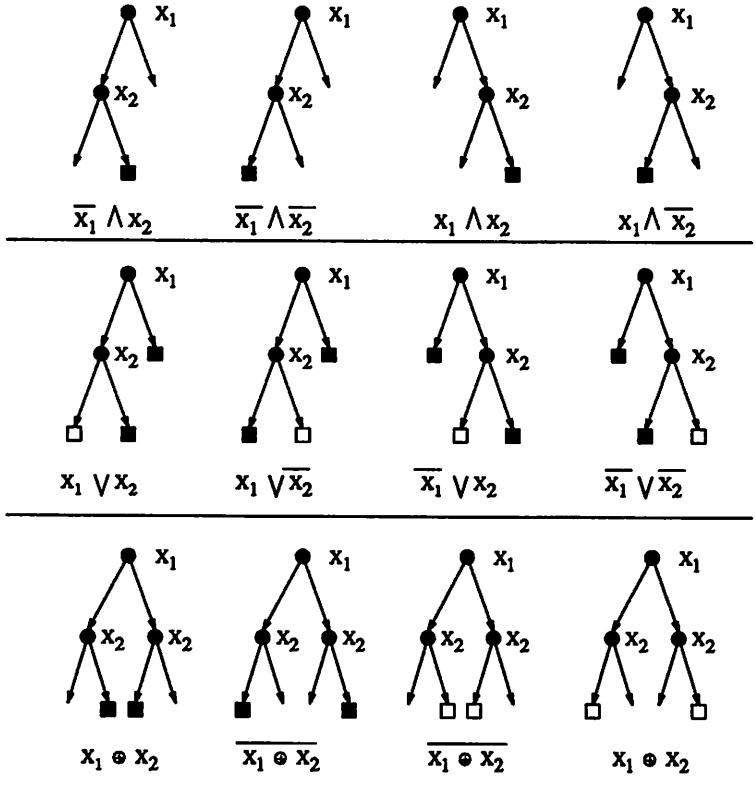


Figure 5.5: Fringe patterns identified by *fulfringe*

add the patterns in the third row and identify all possible functions of 2 variables. As in *dcfringe* and *fringe*, these new composite attributes are added (if they have not yet been generated) to the list of existing attributes and a new decision tree is built. The process is iterated until no further reduction in the decision tree size takes place or a decision tree with only one decision node is built.[6]

The composite attributes are Boolean combinations of existing attributes and, therefore, the RODGs for them are created in a straightforward way using the Boolean operators between existing functions provided by the RODG package. Expression (5.1) can still be used to derive the RODG implemented by a decision tree defined over this extended set of variables. However, the control variable v_i in expression (5.1) is no longer a primitive variable but a composite function. This is not a problem for the Boolean function manipulation routines that treat a primitive and a composite variable in an uniform way.

It is important to note here that even though the successive decision trees are defined using composite attributes, the RODG that corresponds to any one of these trees is still defined over the original set of variables. The constructive induction algorithm is simply used to derive a simpler Boolean function to initialize the RODG reduction algorithm.

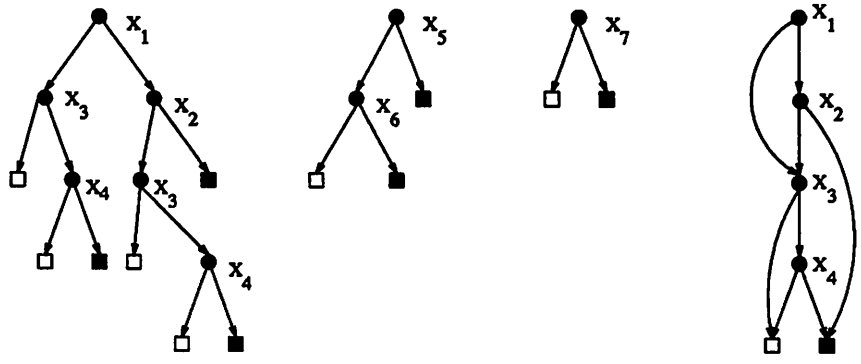


Figure 5.6: Decision trees created by *fulfringe*, after creating the composite attributes $x_5 = x_1 \wedge x_2$, $x_6 = x_3 \wedge x_4$ and $x_7 = x_5 \vee x_6$. The rightmost figure represents the resulting decision graph.

Figure 5.6 shows the successive decision trees obtained using this algorithm for the function used in figure 5.4. The first decision tree created is the same as before. Using the patterns listed in figure 5.5 the algorithm creates the two following attributes: $x_5 = x_1 \wedge x_2$ and $x_6 = x_3 \wedge x_4$. A smaller decision tree is then built using these attributes (together with

[6]The first condition is only necessary to ensure the algorithm will terminate in a reasonable time. In normal usage, a decision tree with a single node will always be obtained.

the primitive ones, in general) and the new attribute $x_7 = x_5 \vee x_6$ is then created. The final decision tree has a single node that tests attribute x_7. The RODG is then created by computing $f = \text{Ite}(x_7, 1, 0)$. In this case, the final RODG is the same as the one obtained using the initial decision tree although this is not always the case.

5.4.1.3 Initialization Using the Restrict Operator

The third way to initialize the algorithm is to use the restrict operator [21]. This RODG operator can be used to obtain a more compact RODG representation for a function defined by its *on* and *off* sets.

The restrict operator belongs to a family of heuristics [92] that generate a small RODG by merging, in a bottom up fashion, nodes in an RODG. The merging of nodes is performed in a way that keeps the function that corresponds to the generated RODG contained in the union of f_{dc} and f_{on}. The restrict heuristic is remarkably fast and obtains, in some cases, RODGs that are much better solutions that the ones obtained by the much slower decision tree algorithms. However, in many other cases, the solutions are much worst and totally useless for inductive inference applications. For this reason, this heuristic is valuable as a way to initialize the local optimization procedure but cannot be used to generate the final RODG.

The restrict operator also has the problem that it tends to generate RODGs that depend only on the variables that come first in the ordering selected, even if this is not the best choice. For problems with a large number of variables, the RODG initialized using this approach may not be a good starting point.

5.4.2 Reducing an RODG by Applying Local Transformations

Further reductions in the size of the RODG obtained using one of the initialization procedures described above are performed in steps. At each step, one or more nodes are removed from the RODG. To each node n_i in the RODG is associated a vector w_i that contains a 1 for the positions that correspond to instances in the training set that define paths in the RODG that go through node n_i and 0 otherwise. The j position of vector w_i is denoted by w_i^j and these vectors w_i can be computed recursively by applying the following expressions:

$$w_0^j = 1 \tag{5.2}$$

$$w_i = \Big(\bigvee_{n_j : n_j^{\text{else}} = n_i} \overline{v_j} \wedge w_j \Big) \vee \Big(\bigvee_{n_j : n_j^{\text{then}} = n_i} v_j \wedge w_j \Big) \tag{5.3}$$

The REMOVENODE procedure, described below, reduces the description length by making one of the nodes in the RODG redundant. This is done by redirecting all its incoming edges. When node n_i is under consideration, the algorithm goes through all incoming edges and tries to select, for each one of them, a different node n_k that implements a function as close to the target as possible (see figure 5.7).

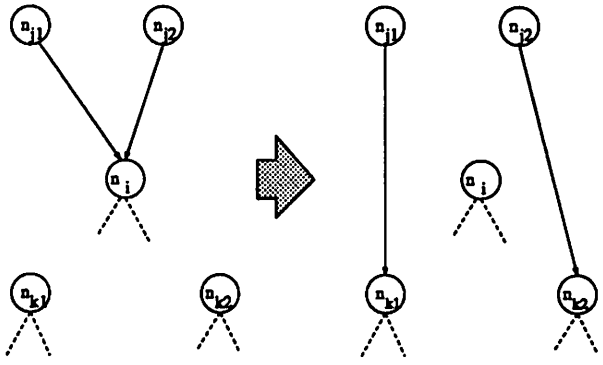


Figure 5.7: Removing one node from the RODG

The value of this function is only important for the examples that reach n_i through the edge that is being redirected. The pseudo-code in figure 5.4.2 describes how this modification is accomplished. This algorithm takes as input one copy of the current RODG and tries to redirect the incoming edges for each of this nodes. If the RODG that results from redirecting each one of these edges has a cost function smaller than the original one, the procedure returns the modified RODG.

If the above procedure fails to make one node redundant, procedure REPLACEPAIR is called. REPLACEPAIR removes a pair of nodes by creating another node that implements a function as close as possible to the functions implemented by the pair of nodes under consideration (see figure 5.9). The value of the new function is only relevant for the examples that reach these nodes.

5.4.3 Selecting the Best Ordering

The selection of a good ordering for the variables is of critical importance if the target is to obtain a compact RODG. Regrettably, selecting the optimal ordering for a given

```
REMOVENODE(R)
  foreach n_i
    foreach n_j s.t. n_j^else = n_i        For all nodes that have the else edge pointing to n_i
      Select n_k such that |(n_k ⊕ t) ∧ w_j ∧ v̄_j| is minimal
      Modify RODG such that n_j^else = n_k
    foreach n_j s.t. n_j^then = n_i        For all nodes that have the then edge pointing to n_i
      Select n_k such that |(n_k ⊕ t) ∧ w_j ∧ v_j| is minimal
      Modify RODG such that n_j^then = n_k
    if Modified RODG has smaller description length
      return (Modified RODG)
    else
      Undo changes
  return (Failure)
```

Figure 5.8: The REMOVENODE procedure.

function is NP-complete [85] and cannot be solved exactly in many cases. However, many heuristic algorithms have been proposed for this problem [26, 83].

In this setting, the problem is even more complex because the objective is to select an ordering that minimizes the final RODG and this ordering may not be the same as the one that minimizes the RODG obtained after the initialization step. The solution found is

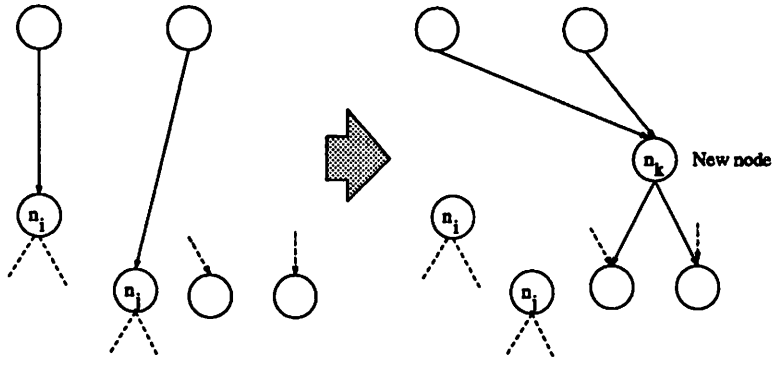


Figure 5.9: Replacing a pair of nodes by a new node.

```
REPLACEPAIR(R)
  foreach n_i
    foreach n_j                                   For each pair of nodes
      w := w_i ∨ w_j                  w is 1 for all objects that reach nodes n_i or n_j
      Create n_k = ITE(v_k, n_a, n_b) such that |(n_k ⊕ t) ∧ w| is minimal
      Modify RODG such that incoming edges into n_i and n_j point to n_k
      if Modified RODG has smaller cost function
        return (Modified RODG)
      else
        Undo changes
  return (Failure)
```

Figure 5.10: The REPLACEPAIR procedure

to use the *sift* algorithm for dynamic RODG ordering [83] after each local modification is performed.

The *sift* algorithm is based on the fact that swapping the order of two variables in the RODG ordering can be done very efficiently [13, 44] because only the nodes in these two levels are affected. The algorithm selects the best position in the ordering for a given variable by moving that variable up and down (using the inexpensive swap operation) and recording the smaller size observed. This procedure is applied once to all variables and can be, optionally, iterated to convergence. This algorithm is extremely efficient since it was designed to be applied to very large RODGs. Therefore, its repeated application to the relatively small RODG encountered in this problems is not a major limitation in the overall speed of the approach.

5.4.4 Efficiency Issues

The complexity of these algorithms depends strongly on the approach used to evaluate the complexity reduction achieved by each operation. Because the effect of each change can be estimated locally the overall description length of the RODG or the number

of exceptions created by a local modification doesn't need to be recomputed after each step.

With careful coding, the REMOVENODE procedure requires $O(s^2m)$ operations, where, as before, s is the number of nodes in the current RODG and m is the size of the training set. The REPLACEPAIR procedure is more expensive and requires $O(s^3m)$ operations. By using bit packing techniques [82], the algorithm can be applied to reduce large RODGs in reasonable times.

For large problems, the decision graph obtained from the initialization phase may be too large. In this case, the local optimization algorithm may take a large amount of time to reduce this graph. For these problems, the algorithm can be run on a fast mode that initializes the graph with a decision tree that is not fully consistent with the training set data. This is obtained by stopping the growth of the decision tree when the entropy of the samples that reach a particular node is inferior to a given value. The larger this value, the smaller the decision tree obtained and the simpler the initial graph. However, if this threshold is set to high, the local optimization algorithm will not be able to improve the solution and the generalization accuracy obtained by the decision graph will not be any better than the one obtained by the decision tree that was used in the initialization.

5.4.5 The Smog Algorithm

The algorithms described in sections 5.4.1 through 5.4.3 can now be combined in a straightforward way as shown in figure 5.4.5.. The main loop simply calls the above procedures until both return failure, while calling, at each iteration, the reordering procedure described in section 5.4.3.

The algorithms described in this section were implemented in a system called **smog** (*Selection of Minimal Ordered Graphs*) that uses the RODG package described in [13] to perform the standard RODG manipulations. After an extensive empirical evaluation of the algorithm performance with various values of the parameters, the default value for the parameter α in equation (2.15) was set equal to 0.5. However, for some problems, other values of this parameter lead to better results and the user should select the appropriate value of α for a particular application.

```
MAINLOOP()
  S :=INITRODG()
  repeat
    R := S                                   Store the current RODG
    R :=REORDER(R)                           Select best ordering for current RODG
    S :=REMOVENODE(R)
    if S =FAILURE                            RemoveNode operation failed
      S :=REPLACEPAIR(R)
  until S = Failure
  return (R)
```

Figure 5.11: The smog algorithm

5.5 Experimental Results

5.5.1 Experiments Using the Exact Algorithm

The applicability of the exact algorithm is known to be limited to small examples. It is, however, interesting to evaluate the limits of such an approach and to study the dependency of the number of compatibles and the number of primes on the problem size.

A number of simple functions was selected for this purpose. These functions are also described in appendix B. Table 5.1 summarizes some statistics about these functions. For each function, the RODG size required to represent the target concept represents an upper bound on the minimum size of a RODG consistent with the training set data. It is not a sharp bound because the training set is chosen randomly and there may exist other functions consistent with the target concept but with smaller RODGs. The value of this upper bound is listed in the last column of table 5.1. Training sets for each of these concepts were generated by randomly selecting half of the points in the domain and labeling them according to the target concept.

Table 5.1 also lists the sizes of the RODGs required to represent the *on*, *off* and *don't care* sets that were obtained from these training sets and the size of the 3TRODG that represents the training set, F.

The exact algorithm was run in these examples using a timeout of 1 hour (in a

Concept (g)	# inputs	RODG size				
		f_{on}	f_{dc}	f_{off}	F	Upper bound
xor5	5	14	17	15	22	11
xor6	6	23	27	24	38	13
xor7	7	36	46	38	59	15
xor8	8	56	78	62	97	17
xor9	9	97	129	92	158	19
xor10	10	160	227	160	273	21
dnfa	6	14	24	23	32	8
dnfb	7	24	43	42	53	14
dnfc	8	44	75	61	90	23
dnfd	9	45	127	124	150	23
carry4	8	51	75	61	97	32
monks1	10	79	124	81	142	10
monks2	10	89	134	124	171	25
heel6	6	14	24	22	33	8
heel9	9	74	127	101	151	11

Table 5.1: Test function statistics for the exact approach

DEC/alpha machine) and a memory usage limit of 140 Megabytes. This algorithm managed to compute the maximum compatible sets for each of the problems, but failed to compute the full set of primes in some of them. The results obtained, and the resulting RODG sizes are shown in table 5.2. These results seem to imply that the applicability of the exact minimization algorithm is limited to problems of no more than 8 variables. It is an open question whether or not this represents a clear advantage over the exact approach proposed in [91], although the algorithms presented here seem to be less limited by the number of points in the don't care set. There exists, however, the possibility of applying implicit enumeration techniques to the problem of generating and solving the covering problem that results from this formulation.

5.5.2 Experiments Using the Heuristic Algorithm

An objective evaluation of the performance of the heuristic algorithm in terms of how small are the RODGs obtained is hard to obtain because, in general, it is impossible to obtain the value for the exact solution. An upper bound can be computed by computing the size of the RODG needed to represent the target concept, but, in some cases, this bound

Concept	# maximals	# primes	RODG size
xor5	15	27	10
xor6	38	107	13
xor7	90	550	15
xor8	132	35601	17
xor9	521	-	-
xor10	1437	-	-
dnfa	13	42	6
dnfb	27	55	14
dnfc	156	-	-
dnfd	227	-	-
carry4	233	-	-
monks1	894	-	-
monks2	568	-	-
heel6	18	74	8
heel9	234	-	-

Table 5.2: Resulting RODG sizes

may be too lose. This is likely to happen if there are paths in this RODG that are exercised by a only a small number of minterms and none of these minterms is present in the training set.

Instead, the RODG sizes generated by the two most effective initialization procedures were used a standard of comparison for the quality of the final RODG generated. Since the heuristic approach can be initialized using either the *fulfringe* constructive induction approach or the *restrict* heuristic for RODG reduction, it is easy to obtain the sizes after the initialization step. This will also illustrate the need for the use of a variety of approaches to initialize the local optimization algorithm. In fact, each of the initialization strategies used can generate RODGs that are several orders of magnitude above the minimum result possible.

For a number of functions labeled training set sizes of increasing size were generated and used as the input for the smog algorithm. A fixed ordering of the variables was used in these comparisons.

Tables 5.3 and 5.4 shows the sizes of the RODGs obtained by the two initialization procedures referred above and the final size obtained by the smog algorithm. It is important to note that even relatively small improvements in the size of the RODG can have a large

impact on the quality of the generalization performed. For this reason, the differences observed in all these examples are very important and do, sometimes, render the RODGs obtained after the initialization algorithms uninteresting for inductive inference purposes.

It is interesting to note that, by fixing the ordering of the variables, some of the problems are made much more difficult. For example, the *mux11* problem that is easily solved by smog[7] if a reordering of the variables is allowed becomes very difficult under a fixed ordering because the RODG size, in this problem, is very sensitive to the order selected.

[7] As shown in the learning curves for this problem described in chapter 7

Concept	Training Set Size	smog	restrict	fulfringe
heel9	100	14	30	24
	200	18	45	18
	300	18	48	18
	400	18	44	18
	500	18	47	18
sm12	200	38	79	66
	400	54	133	66
	600	60	162	66
	800	64	183	66
	1000	65	191	66
str18	200	31	49	90
	400	51	92	192
	600	69	129	137
	800	65	172	118
	1000	104	210	212
heel18	200	52	91	103
	400	57	169	103
	600	65	231	103
	800	74	309	103
	1000	73	359	103
mux6	200	21	101	22
	400	22	183	22
	600	22	241	22
	800	22	303	22
	1000	22	357	22
mux11	200	64	104	2120
	400	99	188	765
	600	147	279	300
	800	116	363	282
	1000	130	444	282
par4_16	200	9	113	9
	400	9	206	9
	600	9	302	9
	800	9	385	9
	1000	9	450	9

Table 5.3: Initial and final sizes of reduced graphs, part 1

Concept	Training Set Size	smog	restrict	fulfringe
par5_32	200	67	105	5831
	400	116	200	46336
	600	153	294	281376
	800	242	416	537079
	1000	305	463	438587
dnf1	200	42	82	329
	400	81	146	5082
	600	102	213	25700
	800	140	256	134036
	1000	173	300	459327
dnf2	200	55	102	1083
	400	106	189	642
	600	79	261	217
	800	90	314	312
	1000	224	368	605
dnf3	200	50	90	337
	400	72	153	1722
	600	112	213	453
	800	143	280	648
	1000	170	328	665
dnf4	200	55	105	1160
	400	105	202	25362
	600	142	283	22236
	800	201	362	9597
	1000	227	430	9925

Table 5.4: Initial and final sizes of reduced graphs, part 2

Chapter 6

Finite State Machines

6.1 Introduction and Related Work

This chapter addresses a representation that is fundamentally different from the ones addressed previously in that the objects in the domain are defined by attribute sequences of variable length.

Finite state machines are a natural representation for hypotheses in this domain. Unlike the case in the previous chapters the selected representation cannot represent all possible concepts defined in the domain because only sets that correspond to regular languages can be represented by finite state machines [42]. This set is, however, rich enough to contain many concepts of interest and many researchers have addressed inductive inference problems using this representation.

With the particular encoding described in 6.2.1, the selection of a finite state machine that minimizes (2.15) is equivalent to the selection of a finite state machine that has minimal number of states and generates outputs consistent with the labels in the training set. There exists a trivial transformation between the finite state machine that satisfies these conditions and the minimal deterministic finite automaton (DFA) that accepts all positive instances in the training set and rejects all negative instances. Therefore, all the previous work done using the DFA formalism is relevant and closely related to the work discussed here.

The problem of selecting the minimum DFA consistent with a set of labeled strings is known to be NP-complete. Specifically, Gold [32] proved that given a finite alphabet Σ, two finite subsets $S, T \subseteq \Sigma^*$ and an integer k, determining if there is a k-state DFA that

recognizes L such that $S \subset L$ and $T \subset \Sigma^* - L$ is NP-complete.

If *all* strings of length n or less are given (a *uniform-complete* sample), then the problem can be solved in a time polynomial on the input size [100, 75, 37]. Note, however, that the size of the input is in itself exponential on the number of states in the resulting DFA. Angluin has shown that even if an arbitrarily small fixed fraction $(|\Sigma^{(n)}|)^{\epsilon}$, $\epsilon > 0$ is missing, the problem remains NP-complete [2].

The problem becomes easier if the algorithm is allowed to make queries or experiment with the unknown machine. Angluin [3] proposes an algorithm based on the approach described by Gold [31] that solves the problem in polynomial time by allowing the algorithm to ask membership queries. Schapire [89] proposes an interesting approach that does not require the availability of a reset signal to take the machine to a known state.

All these algorithms address simpler versions of the problem discussed here. In our case, the learner is given a set of labeled strings and is not allowed to make queries or experiment with the machine. The best algorithms known for the specific problem addressed here, where the learner has not control over the training set, remain the ones proposed by Bierman et Al. [9, 10]. These algorithms are based on an explicit search algorithm and are guaranteed to obtain the exact solution, although they require, in general, exponential time. These algorithms are described in some detail in section 6.4. Recently, connectionist approaches have been proposed that address the problem of learning from a given set of strings, but had limited success. Das and Mozer [22], Giles et al. [29] and Polack [74] propose different approaches based on gradient descent algorithms for neural network training, but their results show that this strategy does not have any important advantages over search-based methods like the ones proposed by Bierman. Not only they are not guaranteed to find the exact solution but they are also very limited in the size of problems they can handle. For example, they are not even able to solve some of the Tomita grammars [99], none of which requires more than 5-state DFAs. It must be pointed out, however, that the main purpose of the connectionist work was not to beat discrete search algorithms, but to evaluate the applicability of such an approach to problems of this type.

Lang [53] describes a promising heuristic approach that is also based on a discrete search strategy and is very efficient. He has shown that it can be applied to find approximate solutions for machines with several hundred states. Regrettably, in many cases the solutions obtained are very far from the minimum and there is no way to estimate how close a given solution is from the optimal one.

A different approach is to view the problem of selecting the minimum automaton consistent with a set of strings as equivalent to the problem of reducing an incompletely specified finite state machine.[1] This problem is more general than the one addressed here and was also proved to be NP-complete by Pfleeger [72]. However, since this problem is of great practical importance, many different algorithms have been proposed for its solution. Paull and Unger [71] were the first to propose a method based on the selection of compatibility classes, or compatibles. A compatible is a set of states that are equivalent in the sense that they can be merged without affecting the behavior of the machine. The minimal machine can be found by selecting a minimal set of compatibles that satisfies two simple requirements. This method was improved by Grasselli and Luccio [36] who showed that only a subset of the compatibles, the prime compatibles, need be considered. An efficient implementation of these algorithms was made available in the `stamina` program by Hachtel et al. [38]. This algorithm is still the state of the art in finite state machine reduction for the majority of the cases. Some problems, however, exhibit exponentially large numbers of compatibles, rendering an explicit enumeration approach such as `stamina`'s ineffective. In particular, incompletely-specified finite state machines obtained from training sets by the procedure described in the next section tend to have an extremely large number of compatibles. In this case, a version of Grasseli and Luccio's algorithm based on the implicit enumeration of the compatibles proposed by Kam et al. [47] is more efficient.

The approach described in this chapter represents joint work with Stephen Edwards and is also based on the use of implicit techniques to perform the search for the minimum consistent finite state machine. Like Bierman's algorithms, it can be viewed as an implementation of Gold's *identification in the limit* paradigm. Unlike Bierman's approach, however, the algorithms described in these chapter do not explicitly try all possible solutions. The selection of the minimal consistent finite state machine is done by keeping an implicit list of all consistent machines.

[1] The exact way in which this reduction is performed is section 6.2.

6.2 Definitions

6.2.1 Finite State Machines

The algorithms described in this chapter can be used with minor modifications to induce either Mealy or Moore machines and the differences required will be pointed out when needed.

Definition 10 (Mealy Machines) *A* Mealy Machine *is a 6-tuple* $M = (\Sigma, \Delta, Q, q_0, \delta, \lambda)$ *where*

$\Sigma \neq \emptyset$ *is a finite set of input symbols (The symbol a denotes a particular input symbol)*

$\Delta \neq \emptyset$ *is a finite set of output symbols (The symbol b denotes a particular output symbol)*

$Q \neq \emptyset$ *is a finite set of states (The symbol q denotes a particular state)*

$q_0 \in Q$ *is the initial "reset" state*

$\delta(q, a) : Q \times \Sigma \rightarrow Q \cup \{\phi\}$ *is the transition function*

$\lambda(q, a) : Q \times \Sigma \rightarrow \Delta \cup \{\epsilon\}$ *is the output function*

ϕ *denotes an unspecified transition.* ϵ *denotes an unspecified output.*

Definition 11 (Moore Machines) *A* Moore Machine *is a Mealy Machine where* $\lambda(q, a_1) = \lambda(q, a_2)$ *for all* $a_1, a_2 \in \Sigma$ *thereby implying that the output of a Moore Machine does not depend on the input, only the state.*

The domain of the second variable of functions λ and δ is extended to strings of any length in the usual way:

Definition 12 (Output of a Sequence) *If* $s = (a_1, \ldots, a_k)$ *the notation* $\lambda(q, s)$ *denotes the output of a Moore or Mealy machine after a sequence of inputs* $(a_1, \ldots, a_k)$*, is applied in state q. The output of such a sequence is defined to be*

$$\lambda(q, s) \equiv \lambda(\delta(\delta(\cdots\delta(q, a_1)\cdots), a_{k-1}), a_k) \tag{6.1}$$

By definition, if M is a Moore Machine, then $\lambda(q, (a_1, \ldots, a_k))$ *is independent of* a_k.

accept	reject
1	0
11	10
111	01
1111	00
11111	011
111111	110
1111111	11111110
11111111	10111111

Figure 6.1: A training set with variable number of attributes.

Definition 13 (Destination State of a Sequence) *If* $s = (a_1, \ldots, a_k)$ *the notation* $\delta(q, s)$ *denotes the final state reached by a Moore or Mealy machine after a sequence of inputs* $(a_1, \ldots, a_k)$*, is applied in state* q*. This state is defined to be*

$$\delta(q, s) \equiv \delta(\delta(\ldots\delta(\delta(q, a_1), a_2)\ldots), a_k) \tag{6.2}$$

Since the definitions above may require the computation of $\lambda(\phi, a)$, and to avoid unnecessary notational complexities, $\lambda(\phi, a)$ is defined to be equal to ϵ. This means that the output for sequences not present in the training set is defined to be ϵ.

The function $\delta(q, a)$ defines the structure of the state transition graph of the finite state machine while the function $\lambda(q, a)$ defines the labels present in each of the edges of that graph.

The exact algorithm described in this chapter minimizes expression (2.14) using an encoding scheme that is exactly the one described in section 2.7.1 except for the fact that the value of the output for each edge in the underlying graph is appended to the description of the state pointed to by that edge.

6.2.2 From Training Sets to Tree Finite State Machines

An example of a possible training set for this representation is given in figure 6.1.[2] Alternatively, the training set can be specified by one or more sequences where, at each time, the value of the input/output pair is known. Figure 6.2 shows an example of this alternative way to specify a training set. Both forms of training sets description are equivalent and

[2]This particular training set for this concept was proposed by Tomita [99].

Input: A A B B A B A B B B B A B A A A B A B B
Output: 0 1 0 0 0 1 0 1 1 1 1 1 0 1 0 1 0 0 1 0
⟶ time

Figure 6.2: Observed input/output sequences for some FSM.

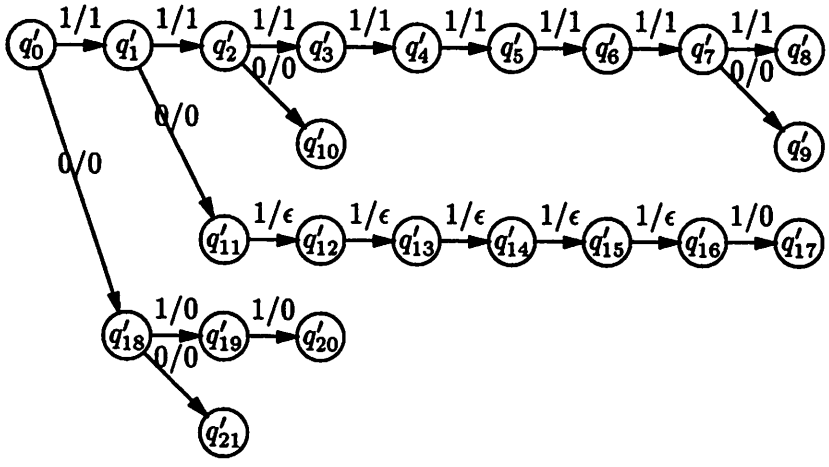


Figure 6.3: The TFSM that corresponds to the training set in figure 6.1

can be viewed as defining a particular type of incompletely specified finite state machine, a *Tree Finite State Machine* (TFSM). Figure 6.3 shows the TFSM that corresponds to the training set in figure 6.1.

Definition 14 *A Tree Finite State Machine is a finite state machine satisfying definition 10 and the following additional requirements:*

$$\forall q \in Q \setminus q_0,\ \exists^1 (q_i, a) \in Q \times \Sigma \ s.t.\ \delta(q_i, a) = q$$

$$\forall q \in Q, \forall a \in \Sigma\ \delta(q, a) \neq q_0$$

These requirements specify that the graph that describes the TFSM is a tree rooted at state q_0.

Definition 15 (Contained strings) *An input string $s = (a_1, \ldots, a_k)$, $a_i \in \Sigma$ is contained in a TFSM $T = (\Sigma, \Delta, Q, q_0, \delta, \lambda)$ iff there exists a sequence of states $(q_{r_0}, q_{r_1}, \ldots, q_{r_k})$ in Q such that for $i = 1, \ldots, k$ $\delta(q_{r_{i-1}}, a_i) = q_{r_i}$, $q_{r_0} = q_0$ and $\lambda(q_0, s) \neq \epsilon$.*

6.2.3 The Satisfying Criteria

A notion of "inequality" between outputs that takes into account the fact that some outputs can be unspecified is the following:

Definition 16 (Output incompatibility) *The output in a given transition, $b_i = \lambda(q_i, a_i)$ is said to be compatible with $b_j = \lambda(q_j, a_j)$ and denoted $b_i \equiv b_j$ in accordance with the following definition:*

$$b_i \equiv b_j \begin{cases} \text{true} & \text{if} \quad b_i = b_j \\ \text{true} & \text{if} \quad b_i = \epsilon \vee b_j = \epsilon \\ \text{false} & \text{otherwise} \end{cases} \tag{6.3}$$

The aim is to construct a machine M that exhibits a behavior equal to T for all strings contained in T. Assume that $M = (\Sigma, \Delta, Q, q_0, \delta, \lambda)$, $Q = \{q_0, \ldots q_k\}$ and $T = (\Sigma, \Delta, Q', q'_0, \delta', \lambda')$, $Q' = \{q'_0, \ldots q'_{k'}\}$ unless otherwise stated.

Definition 17 (Satisfying machine) *A machine M is consistent with a TFSM T if, for any input string $s = (a_1, \ldots, a_k)$ contained in T*

$$\lambda(q_0, s) \equiv \lambda'(q'_0, s) \tag{6.4}$$

A function $F : Q' \rightarrow Q$ is called a mapping function between TFSM T with set of states Q' and FSM M with set of states Q if $F(q'_0) = q_0$. A mapping function F satisfies the output requirement if

Definition 18 (Output requirement)

$$\forall q = F(q'),\ \lambda'(q', a) \equiv \lambda(q, a) \tag{6.5}$$

A mapping function F satisfies the transition requirement if

Definition 19 (Transition requirement)

$$\forall q = F(q'),\ F(\delta'(q', a)) = \delta(q, a) \tag{6.6}$$

Theorem 2 *For any machine $M = (\Sigma, \Delta, Q, q_0, \delta, \lambda)$ consistent with the tree finite state machine $T = (\Sigma, \Delta, Q', q'_0, \delta', \lambda')$ there exists a mapping function $F : Q' \rightarrow Q$, $F(q'_0) = q_0$, that satisfies the output and transition requirements.*

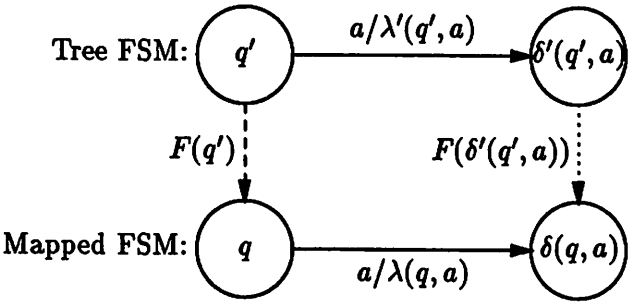


Figure 6.4: Output and transition requirements depicted graphically

Proof: let $s_k^i = (a_1^i, a_2^i \ldots a_k^i)$ be an arbitrary substring of some string s^i contained in T and let the mapping function F be defined by $F(\delta'(q_0', s_k^i)) = \delta(q_0, s_k^i)$.

Consider now all strings $s_{k+1}^i = (a_1^i, a_2^i \ldots a_{k+1}^i)$ in T. By the hypothesis, $\lambda(q_0, s_{k+1}^i) \equiv \lambda'(q_0', s_{k+1}^i)$ and therefore the output requirement has to be satisfied (simply make q' in expression (6.5) equal to $\delta'(q_0', s_k^i)$).

Furthermore, since the strings s_{k+1}^i are themselves substrings of some string contained in T (or else no s_{k+1}^i contained in T exists, in which case the requirement is automatically satisfied), $F(\delta'(q_0', s_{k+1}^i))$ equals $\delta(q_0, s_{k+1}^i)$ and therefore F also meets the transition requirement. □

The result in this theorem is important because it is not valid, in general, if the incompletely specified machine T is not a TFSM.

6.3 Compatible and Incompatible States

Theorem 2 shows that the selection of a machine that satisfies the training set is equivalent to the selection of an appropriate mapping function. The first step in selecting such a mapping function is the computation of the incompatibility graph.

6.3.1 The Incompatibility Graph

Two states q_i' and q_j' in a finite state machine T are incompatible if, for some input string s, $\lambda(q_i', s) \not\equiv \lambda(q_j', s)$. The *incompatibility graph* represents this information. The nodes in this graph are the states in Q', and there is an edge between state q_i' and q_j' if these states are incompatible.

The incompatibility graph is represented by a function $I : Q' \times Q' \rightarrow \{1, 0\}$.

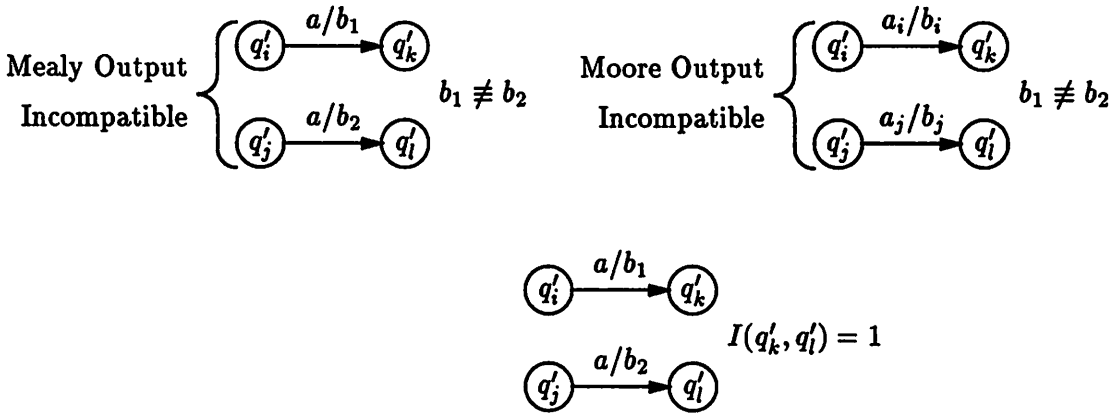


Figure 6.5: Conditions for output and transitive incompatibility.

$I(q'_i, q'_j)$ is 1 if and only if states q'_i and q'_j are incompatible. Since the incompatibility relation is symmetrical, $I(q'_i, q'_j) = I(q'_j, q'_i)$ for all $q'_i, q'_j \in Q'$. A state is never incompatible with itself, i.e., $I(q'_i, q'_i) = 0$. The computation of $I(q_i, q_j)$ uses the following two definitions:

- Output incompatibility: two states are output incompatible if, on some input, the two states produce a different output.
- Transitive incompatibility: two states are incompatible if, on some input, the respective next states are incompatible.

Definition 20 (Incompatibility Graph) *The incompatibility graph is*

$$I(q'_i, q'_j) = \begin{cases} 1 & \text{if } \exists a_i, a_j : \lambda(q'_i, a_i) \not\equiv \lambda(q'_j, a_j) \wedge a_i = a_j^{\dagger} \\ 1 & \text{if } \exists a : q'_k = \delta(q'_i, a) \wedge q'_l = \delta(q'_j, a) \wedge I(q'_k, q'_l) = 1 \\ 0 & \text{otherwise} \end{cases} \tag{6.7}$$

For Moore machines, the condition marked with a dagger (†), $a_i = a_j$, is omitted.

6.3.2 A Clique in the Incompatibility Graph

A clique in the incompatibility graph gives a lower bound on the size of the minimum machine. By definition, pairs of incompatible states cannot be mapped to the same state and therefore, a clique in this graph corresponds to a group of states that must map to different states in the resulting machine.

A large clique in the incompatibility graph is identified using a slightly modified version of an exact algorithm proposed by Carraghan and Pardalos [19]. Pseudo-code for

```
MaxClique(C = {c_1, ..., c_n} ⊆ Q', depth)
  for i := 1, ..., n − depth                          states being examined
    O := {c : c ∈ {c_{i+1}, ..., c_n}, I(c_i, c) = 1}   state to consider next
    if O ≠ ∅
      MaxClique(O, depth + 1)
    else if depth ≥ maxclique
      maxclique := depth + 1                          found a clique larger than any known
```

Figure 6.6: The clique-finding algorithm.

this algorithm is shown in Figure 6.6. The algorithm takes a set of states, forms subsets which are incompatible with another state from the set, and calls itself on these subsets. Each state from each subset is considered in turn, although only states which are "later" in the set (according to an ordering imposed at the beginning) are considered to be part of the new subset.

This algorithm was modified as to stop after a given amount of time has elapsed. Every time a clique is located, the algorithm allocates an amount of time that is a fixed multiple of the time is took to find that clique. This timeout scheme was developed after observing that in most cases, a clique of maximum cardinality was found fairly quickly but the algorithm spent a large amount of time ruling out larger cliques. Such a scheme can be used because the selection of the maximum clique is not critical for the success of the algorithm.

The size of the clique provides a lower bound on the number of states needed in the resulting machine. This lower bound is used as the starting point for the implicit enumeration algorithm.

6.4 The Explicit Search Algorithm

The explicit search algorithm implemented for the purpose of comparison is based on the algorithm proposed by Bierman et Al. [9, 10]. It builds a finite-state machine and a mapping function F by fitting transitions from the TFSM T into the machine M, one by one, forcing the transition (6.6) and output requirements (6.5) to be satisfied for all the

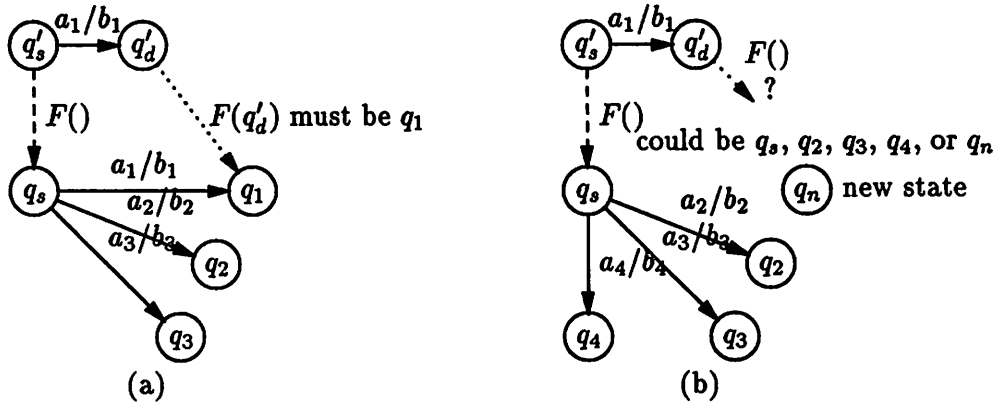


Figure 6.7: The two main cases of the explicit algorithm

transitions considered.

The algorithm is started with a machine containing only the reset state. At any time, the algorithm selects a transition in T and has to verify that transitions in M generate outputs consistent with the transitions in T. Let q'_s be the state where the transition under consideration origins and q_s be the state of M such that $F(q'_s) = q_s$ and let the transition under input a be the one under consideration. Two main cases should be considered, shown in figure 6.7:

- The choice of the mapping of the destination state is forced by an existing transition, labeled with a. If this is the case, two things may happen:
  - The output of the corresponding transition in M is consistent with the output of the transition in T. This means that the machine M is, so far, consistent with T.
  - The output of the corresponding transition is not consistent with the output of the transition in T. In this case, some transition in M (not necessarily this one) is wrong and the algorithm backtracks to the last point where it had a choice and tries another assignment of the destination state.
- There is no existing transition labeled with a, so any of the existing states or a new state is a possibility.

6.5 Solution Using an Implicit Enumeration Algorithm

The implicit approach described in this section avoids the need to explicitly search for the right mapping function. It does so by keeping an implicit description of all the mapping functions that satisfy the output and transition requirements.

The discrete function manipulation needed to keep this implicit list of possible mappings is performed by a multi-valued RODG package Discrete function manipulation using this type of representation is briefly described described in appendix A.

This approach makes the implicit algorithm very simple to describe, but incurs the overhead imposed by the use of discrete function manipulation routines. This overhead can be recovered if the regularities of the problem make the use of an implicit enumeration technique more efficient than an explicit one.

6.5.1 Implicit Enumeration of Solutions

An implicit list of the valid mapping functions $F : Q' \rightarrow Q$ can be directly manipulated using simple Boolean operations. This list is kept by considering a function $\mathcal{F} : Q^{|Q'|} \rightarrow \{0,1\}$ defined as follows:

Definition 21 *$\mathcal{F}(x_0, x_1, \ldots, x_{|Q'|-1}) = 1$ for the point $v_0, v_1, \ldots, v_{|Q'|-1}$ if the mapping function F defined by $F(q'_0) = v_0, F(q'_1) = v_1, \ldots, F(q'_{|Q'|-1}) = v_{|Q'|-1}$ produces a machine $|Q'|$ that satisfies the transition and output requirements (6.6) and (6.5).*

There is a one-to-one correspondence between each variable x_i in the support of $\mathcal{F}$ and each node $q'_i \in Q'$. Therefore, restrictions on valid mapping functions can be written as restrictions on the variables x_i. If two states in T, q'_i and q'_j, have to be mapped to different states, this is equivalent to the statement that $\mathcal{F}$ can only be true for points where $x_i \neq x_j$.

The transition and output requirements impose restrictions on the function $\mathcal{F}$. Let q'_i and q'_j be two states in Q'. For any two transitions out of these states that take place on the same input and have different outputs, the output requirement forces the source states of the transition to be mapped to to different states. Let $\lambda'(q'_i, a_i) = b_i$ and $\lambda'(q'_j, a_j) = b_j$. Then, for Mealy machines this requirement is:

$$(a_i = a_j) \wedge (b_i \not\equiv b_j) \Rightarrow x_i \neq x_j. \tag{6.8}$$

For Moore machines, different outputs imply different states even if the inputs are not equal:

$$b_i \not\equiv b_j \Rightarrow x_i \neq x_j. \tag{6.9}$$

Next-state determinism implies that, for any two transitions in the original machine that take place on the same input, the same assignment for the initial states implies the same assignment for the final states. Let $q'_k = \delta'(q'_i, a_i)$ and $q'_l = \delta'(q'_j, a_j)$. Then, this requirement translates into the following restriction:

$$(a_i = a_j \wedge x_i = x_j) \Rightarrow (x_k = x_l). \tag{6.10}$$

This can be rewritten as

$$(a_i = a_j) \Rightarrow (x_i \neq x_j \vee x_k = x_l). \tag{6.11}$$

For Mealy machines, (6.8) and (6.11) can be used to form $\mathcal{F}$ using the algorithm in figure 6.8. For Moore machines, the lines that impose the output restriction are changed to use (6.9) instead of (6.8).

```
MainLoop()
  F := 1
  R := ∅                                          Stores the processed states
  foreach q'_i ∈ Q'
    R := R ∪ q'_i                                 Add this state to the list
    foreach q'_j ∈ R
      foreach a ∈ Σ s.t. δ(q'_i, a) ≠ φ ∧ δ(q'_j, a) ≠ φ
        if λ'(q'_i, a) ≢ λ'(q'_j, a)              Output requirement
          F := F ∧ (x_i ≠ x_j)
        q'_k := δ(q'_i, a)
        q'_l := δ(q'_j, a)
        F := F ∧ ((x_i ≠ x_j) ∨ (x_k = x_l))      Transition requirement
  return F
```

Figure 6.8: The implicit algorithm basic loop

6.5.2 Using the Incompatibility Graph to Improve Performance

The above description of the implicit algorithm is very simple because all the complex manipulation of Boolean functions is performed by the decision graph package. However, for complex problems, the storage requirements of the RODG package limit the usability of the algorithm.

Although (6.8) and (6.10) contain all the information required to fully specify $\mathcal{F}$, the algorithm can be made more efficient by making use of the information contained in the incompatibility graph. In particular, if $I(q'_i, q'_j) = 1$, then $F(q'_i) \neq F(q'_j)$. This implies that (6.8) and (6.9) can be replaced by:

$$I(q'_i, q'_j) = 1 \Rightarrow x_i \neq x_j. \tag{6.12}$$

As described in section 6.5.1, the resulting function $\mathcal{F}$ is 1 for all points in $Q^{|Q'|}$ that represent a valid mapping. In general, many mappings exist that satisfy the output and transition requirements. In particular, if a mapping $F : Q' \rightarrow Q$ exists, at least $|Q|!$ mappings exist and can be obtained by renumbering the states in the resulting machine.

Since $\mathcal{F}$ implicitly keeps track of all these redundant mappings, it makes sense to preassign the mapping of some of the states. This can be done by observing that the states in a large clique in the incompatibility graph have to be mapped to different states and therefore pre-assigning the mapping of these states to arbitrary (but different) states in M does not discard any simpler solution and makes the computation of $\mathcal{F}$ much simpler.

Once the mapping of these states has been performed, some mappings for other states can be removed from consideration. In particular, let $C = \{q'_{c_0}, q'_{c_1}, \ldots, q'_{c_l}\}$ be a clique in the incompatibility graph. Then, the mapping of the states in this clique can be chosen arbitrarily to be $F(q'_{c_0}) = q_0, F(q'_{c_1}) = q_1, \ldots, F(q'_{c_l}) = q_l$. Furthermore, if q'_i is a node such that $I(q'_i, q'_{c_j}) = 1$, then $F(q'_i) \neq q_j$.

This information can be used by defining a family of functions $\mathcal{A}_i : Q^{|Q'|} \rightarrow \{0, 1\}$ that describe the values allowed for each of the variables x_i. These functions can be computed by the procedure shown in figure 6.9.

6.5.3 Ordering and Other Efficiency Issues

There are two important ordering problems to be addressed in the algorithm. The first one is the order in which states are included in the set R in the pseudo-code in figure 6.8.

```
COMPUTEALLOWED(C)
  foreach q'_i ∈ Q'
    A_i := 1
  foreach q'_{c_i} ∈ C
    A_{c_i} := (x_{c_i} = q_i)          States in the clique are assigned a unique state
  foreach q'_j ∈ Q' \ C
    foreach q'_{c_i} ∈ C
      if I(q'_{c_i}, q'_j) = 1          If a node is incompatible with a node in the clique
        A_j := A_j ∧ (x_j ≠ q_i)         it should be assigned a different value
```

Figure 6.9: Computation of the allowed mappings functions

The experiments have shown that no other ordering improved significantly the performance when compared with the ordering obtained by performing a breadth first search in the graph that represents T. This is the ordering used, by default.

The second ordering that deserves consideration is the ordering in which variables are stored internally in the RODG package. The best results were obtained by sorting the states according to the degree of the respective nodes in the incompatibility graph. More specifically, the states that correspond to nodes with higher degree in the incompatibility graph come first in the ordering. The intuitive justification for this ordering is that states that are incompatible with a larger number of other states have less degrees of freedom and restrict the branching of the RODG used to represent $\mathcal{F}$.

Dynamic reordering algorithms were also tried but, although this technique reduced somewhat the memory requirements of the algorithm, it also impacted the run-time in a very unfavorable way. It is, therefore, not used in any of the experiments described in section 6.6.

Using the techniques described in section 6.5.2, the main loop of the algorithm is shown in figure 6.10. A large clique of the incompatibility graph is selected and the family of functions $\mathcal{A}_i$ is computed. This algorithm was implement in the program `iasmin`.

```
MAINLOOP()
  F := 1
  R := ∅                                          Stores the processed states
  C := LargeClique()
  COMPUTEALLOWED(C)
  foreach q'_i ∈ Q'
    R := R ∪ q'_i                                 Add this state to the list
    F := F ∧ A_i
    foreach q'_j ∈ R
      foreach a ∈ Σ s.t. δ(q'_i, a) ≠ φ ∧ δ(q'_j, a) ≠ φ
        if I(q'_i, q'_j)                          Output requirement
          F := F ∧ (x_i ≠ x_j)
        q'_k := δ(q'_i, a)
        q'_l := δ(q'_j, a)
        F := F ∧ ((x_i ≠ x_j) ∨ (x_k = x_l))       Transition requirement
  return (F)
```

Figure 6.10: Optimized version of the implicit algorithm

6.6 Experimental Results

6.6.1 Comparison With Algorithms for IFSM Reduction

As described in section 6.2.2, the problem can be viewed as the minimization of the incompletely-specified finite state machine T. Algorithms for the minimization of incompletely specified finite state machines have been the subject of extensive research and several implementations of these algorithms are available. In this section, the performance of `iasmin`, the algorithm described in section 6.5 is compared with the performance of two algorithms that solve the problem using the FSM reduction paradigm: `stamina` [38] and `ism` [47].

To perform this comparison, three target machines, shown in Figure 6.11 were selected. For each machine, a number of training sets was generated, each training set consisting of a single random string of length between 10 and 65. For each time point, the value of the output was available, and, therefore, each training set was effectively equivalent to a set of labeled strings with a size comprised between these two limits. For each length

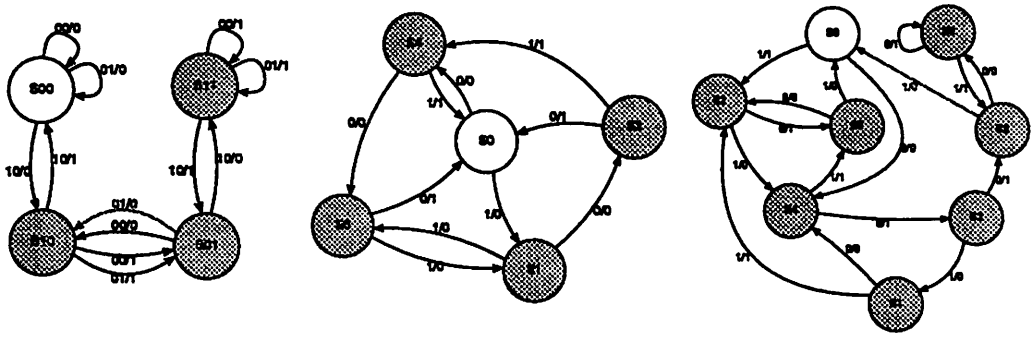


Figure 6.11: Machines used to generate training sets

considered, five training sets were generated. The various programs were then used to find the minimum machine consistent with each of the training sets.

Figures 6.12, 6.13, and 6.14 show the times required to find a solution. Each point represents the average over the five different training sets generated for each given length. These figures show that, in all cases, the state minimization algorithms require a time that increases exponentially in the length of the string while `iasmin` shows a less drastic increase. The different behavior observed illustrates well the distinct exponential dependences off the different approaches: traditional state minimization algorithms require time exponential in

the size of the original training set, while the algorithm described in this chapter requires time exponential in the size of the final machine.

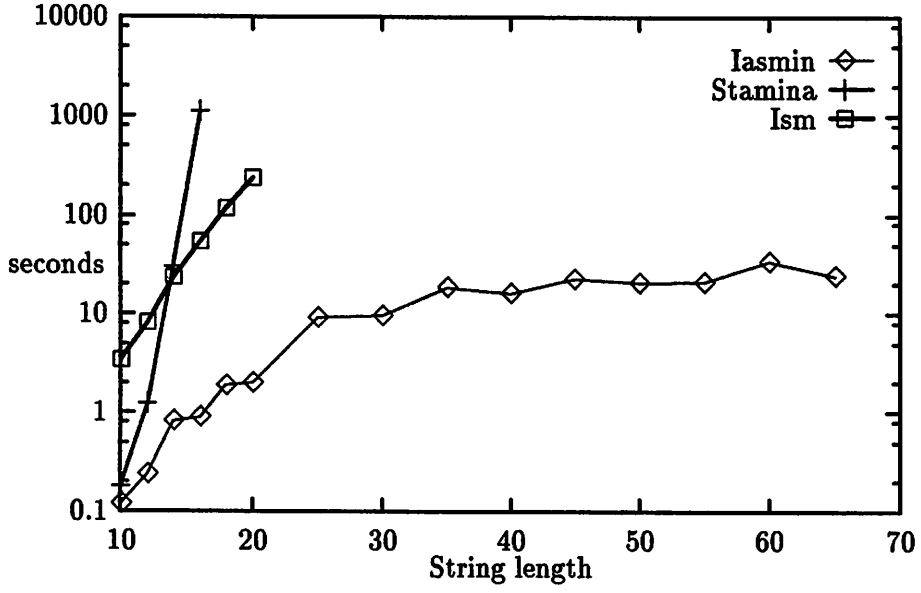


Figure 6.12: Run-time comparison for training sets generated with the first machine

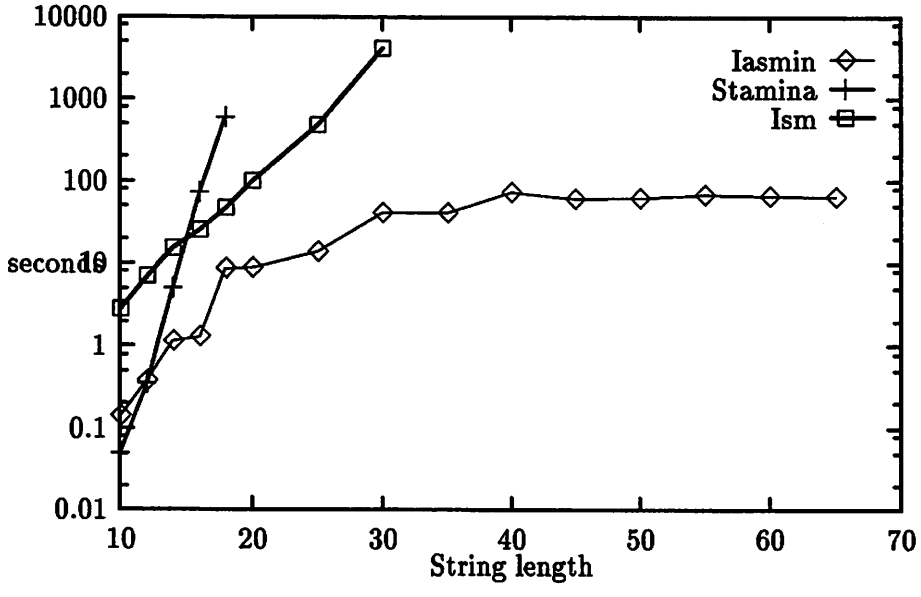


Figure 6.13: Run-time comparison for training sets generated with the second machine

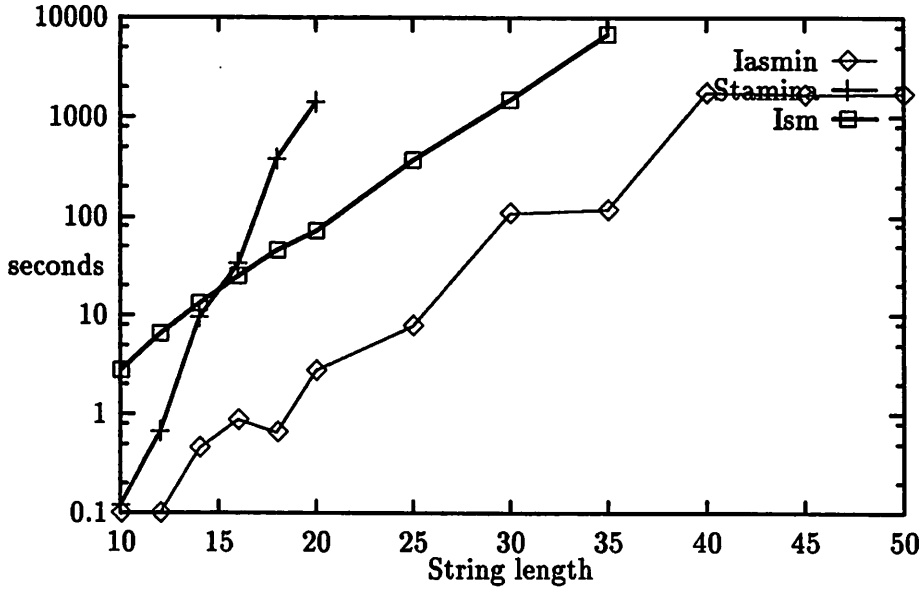


Figure 6.14: Run-time comparison for training sets generated with the third machine

6.6.2 Comparison With the Explicit Search Algorithm

6.6.2.1 Inference of Randomly Generated Machines

The previous section has shown that the algorithms for the minimization of incompletely specified finite state machines cannot be used effectively for the task at hand.

The algorithms based on explicit search presented on section 6.4 perform, however, much better and deserve a more careful comparison. This comparison was performed on a large set of randomly generated finite state machines. The random generation of finite state machines with a known number of minimum reachable states was performed using the following procedure. First, a random state transition graph with a number of states varying between 4 and 25 was generated. By random graph, is it meant that for each state and each possible input, a random output is chosen uniformly from all possible values and a random next state is chosen uniformly from all possible states.

This does not guarantee that all states are reachable or that the machine is irreducible. The number of states in the minimum consistent FSM is clearly bounded above by the number of states in the machine generated in this way. However, the number of states in the minimum consistent solution can be smaller when, for example, not every state in the generated machine is visited or not every transition is taken by the inputs present in

the training set. It is still possible that the generating machine is itself non-minimal.

To obtain a tighter bound on the number of states in the minimal consistent machine for a given training set, all states and transitions which were not visited are discarded. The resulting completely specified machine is sent through a traditional state minimizer, `stamina`, [38] and the number of states in this minimized machine is used as an estimate.

To examine the performance of the implicit and explicit search approaches with differently-sized minimum consistent machines, 575 training sets[3] were generated from 115 finite state machines generated in this fashion. Each program was given an hour and 150 Megabytes of memory to find the minimum consistent machine in a DEC/alpha workstation.

Figure 6.15 shows what fraction of the problems each algorithm was able to complete in the allotted time/space plotted as a function of the number of states in the minimum consistent machine.

These results show that the overall performance of the two algorithms is comparable and no clear advantage exists in favor of each one of the approaches.

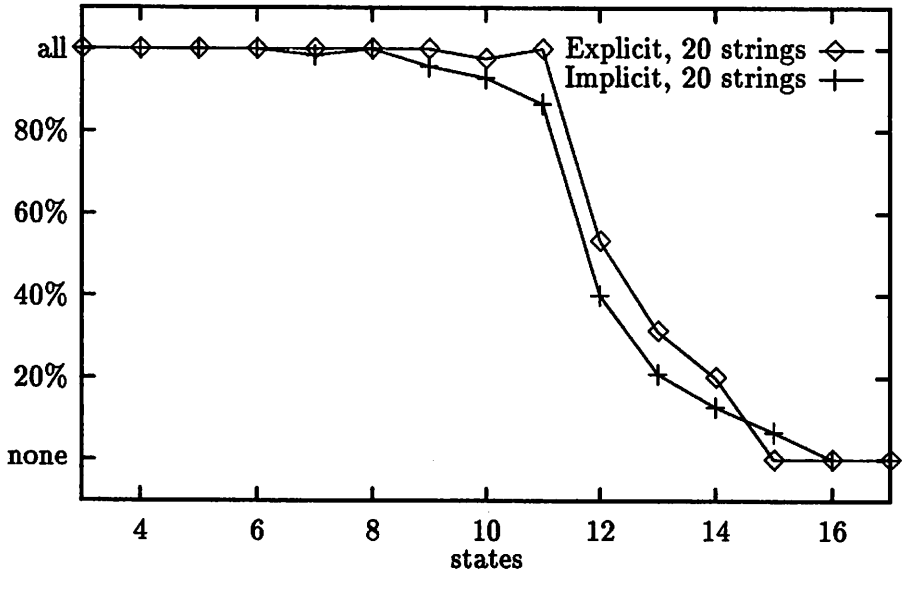


Figure 6.15: Fraction of runs completed

[3]Each training set contained twenty strings of thirty steps each. Each state machine had two inputs (0 and 1), and two outputs (0 and 1).

6.6.2.2 Inference of Machines from Structured Domains

Although the experiments with random machines performed in the previous section give a clear idea of the potential and limitations of the algorithms studied, it is also interesting to evaluate their performance on more structured problems. In fact, problems from structured domains tend to be more regular and exhibit a higher level of symmetry, thereby making them potentially more difficult to learn because it is harder to distinguish between different states.

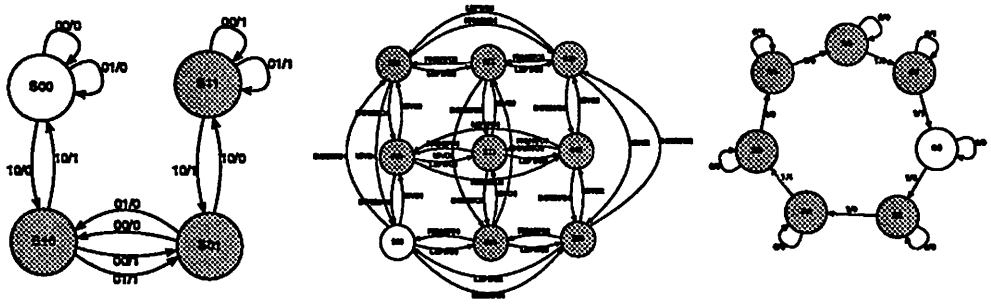
In this section, the target machines are the finite state machines that correspond to the following robot-worlds:

1. N-Rooms: The robot is in a circular house with N rooms. At each point in time, the robot has 3 possible actions (inputs to the finite state machine): toggle the light switch, move to the room on the right or move to the room on the left. The output is 1 if the light in the current room is on, 0 otherwise.

2. N×N Checkerboard: The robot is in a N×N checkerboard field that wraps around in torus-like fashion. There are 4 possible actions: move left, move right, move up or move down. The output is related to the square the robot is on: the white squares have the same output and each black square has a distinct output.

3. N-Counters: The robot is in a circular house with N rooms. There are two possible actions: move to the next room on the right or stay in the current room. The output is one only in the room immediately to the left of the starting room.

Figure 6.16 shows an example of one machine from each of the families listed above. For each problem in these three families, five experiments were performed. Each training set consisted of twenty strings, each of length thirty. Table 6.1 lists the number of successful runs using the same time and memory limits that were used in the previous section. Runs that failed to complete within the allotted time and memory requirements were considered failures. These results seem to imply that inferring machines that exhibit a high level of symmetry in the state transition graph may indeed be more difficult than inferring randomly generated machines. The data is, however, somewhat sparse and more experiments are required to establish a firm conclusion. In fact, some of these problems have multi-valued inputs or outputs, thereby making a direct comparison impossible. This



2-room 3x3 checkerboard 7-counter

Figure 6.16: Examples of the structured finite state machines used

Problem	Iasmin	Explicit
2-room	5	5
3-room	1	3
4-room	0	0
2x2 board	5	5
3x3 board	5	5
4x4 board	0	0
4-counter	5	5
5-counter	5	5
6-counter	5	5
7-counter	5	5
8-counter	0	0

Table 6.1: Number of successful runs.

is not true, however, for the 8-counter problem, a finite state machine with only 8 states with binary valued inputs and outputs. This example seems to show that these machines generate harder problems than the randomly-generated machines studied in the previous section. However, the increased difficulty may be related with other characteristics of the machine, such as the imbalance between the number of times the machine outputs a 1 versus the number of times the machine outputs a 0.

Chapter 7

Experimental Evaluation and Applications

7.1 Experimental Comparison of Generalization Accuracy

A full comparison of the relative merits of all algorithms described in the previous chapters with all other approaches available for the task of performing induction from examples is a very difficult task. Ultimately, each algorithm has its own strengths and weaknesses and will outperform other algorithms in a particular set of problems and be outperformed by them in another set. In fact, the conservation law presented in section 2.4 proves that this has to be the case, and that any learning algorithm will always outperform any other learning algorithm in some set of problems.

There is, however, a case to be made for algorithms that perform well in the majority of problems that are commonly encountered. The tests performed using problems extracted from the literature and from commonly available databases have shown that, of all the algorithms presented in this dissertation, the heuristic algorithms for the inference of reduced ordered decision graphs (`smog`) proved to be the most robust. The algorithms for the inference of Boolean networks (`muesli`) also perform well in many cases but are less robust in problems where the value of the label is poorly correlated with the value of input variables. This situation reduces the effectiveness of the heuristic based on the maximization of mutual information and renders the `muesli` algorithm relatively ineffective in these problems. In the next section, the quality of the inference performed by `smog` and

`muesli` is compared in a detailed way with the quality of the inference performed by `C4.5` [78], the most widely used algorithm for the inference of decision trees.

7.1.1 Experimental Setup

The preferred approach to perform the performance comparison between different algorithms is to use multiple runs with fixed training set sizes. This approach raises the problem of selecting the training set size small enough to make the problem non-trivial but large enough to permit meaningful induction. For instance, the size of the training set may be selected as a function of the complexity of the target concept when expressed using a given representation scheme. This was the approach used by Pagallo and Haussler [69] to derive appropriate training set sizes for the problems they consider.

Regrettably, this approach gives, in many cases, estimates of the required size that are too pessimistic because they are based on theoretical worst-case bounds [40]. Furthermore, the complexity of the representation of the target concept depends on the underlying representation used by the learning algorithms. This makes it difficult to use this approach when two very different representation schemes are used.

Instead of evaluating the performance of the algorithms using only training sets of one fixed size, the average accuracy for training sets of increasing size was computed and used as a measure of performance. For each problem, a test set was selected, containing either all the available data (for the problems for which a limited amount of data is available), or (for problems where larger amounts of data are available) a set of 5000 instances randomly generated and labeled in accordance with the target concept.

Ten different sizes for the training sets were then chosen. For problems with test sets larger than 1000, the training sets have sizes that are a multiple of 100. For problems with test sets of size $z < 1000$, the training set sizes are of sizes $i * z/10$, $i = 1 \ldots 10$.

Each *experiment* consists in evaluating the performance of the programs using 10 different training sets of increasing size, randomly generated in such a way that the larger training sets always include the smaller ones. The *experiment error* for each of the programs is obtained by averaging the generalization error in the test set over the 10 different training set sizes. The *experiment difference* is obtained by subtracting the *experiment errors* of the two algorithms under comparison.

All algorithms were run on exactly the same data by transforming the problems

that contain non-Boolean attributes into problems with only binary attributes using binary coding.

For each problem 10 experiments[1] were performed and the *average error* was computed by averaging the 10 *experiment errors.* The values of the 10 *experiment differences* were used to compute the *average difference* and the *difference variance.* This values were used to perform an analysis of the statistical significance of the observed differences in accuracy.

7.1.2 Evaluation of the Statistical Significance of Observed Differences

Let s_i and r_i denote the *experiment error* in experiment i for algorithm S and algorithm R, respectively. Under reasonable assumptions, the differences in performance observed for each run, $z_i = r_i - s_i$ can be viewed as samples of a random variable z with approximate Gaussian distribution of parameters (μ_z, σ_z^2). This assumption is justified if the number of actual runs performed is large enough to satisfy the requirements of the central limit theorem.

The parameters μ_z and σ_z are unknown and have to be estimated from the experimental data. The objective is to use the experimental results to test the hypothesis that the average generalization accuracy of algorithm S is superior to that of algorithm R.[2]

- H0 : $\mu_z > 0$

against the opposite hypothesis

- H1 : $\mu_z \leq 0$

This hypothesis testing is to be performed with a pre-specified significance level, β, that is defined as the probability that H0 is accepted as true when in fact it is false. A low significance level decreases the probability of accepting a false H0 as true, but increases the probability of rejecting a true H0 as false. Values commonly used for the significance level in the absence of other restrictions are $\beta = 0.1$ or $\beta = 0.05$.

[1] This corresponds to a total of 100 runs by each program.

[2] Note that the term hypothesis used in this section is unrelated with the hypotheses derived by the learning algorithms. Regrettably, hypothesis testing is the commonly accepted terminology in the statistical literature and this overloading of the word is unavoidable.

In order to obtain a test for the validity of H0, observe that the non-biased estimators for the unknown parameters are

$$\overline{z} = \frac{1}{n}\Sigma_i(z_i) \tag{7.1}$$

and

$$\hat{\sigma}_z^2 = \frac{1}{n-1}\Sigma_i(z_i - \overline{z})^2 \tag{7.2}$$

Under this conditions, it is known [59] that

$$t = \frac{(\overline{z} - \mu_z)\sqrt{n}}{\hat{\sigma}_z} \tag{7.3}$$

has a Student's t distribution with $n-1$ degrees of freedom. Therefore, hypothesis H0 should be accepted and the performance of algorithm S judged superior in a statistically significant way if

$$\overline{z} > \frac{\hat{\sigma}_z}{\sqrt{n}} t_{n-1,1-\beta} \tag{7.4}$$

where $t_{n-1,1-\beta}$ is taken from the tables of the Student's t distribution. This is the criterion that will be used in the next sections to evaluate the significance of observed differences in performance in the algorithms.

Expressions (7.1) and (7.2) are used to compute the *average difference* and the *difference variance* for the set of 10 experiments that are performed for each problem. In the sequence, all tests will be performed at a significance level of 0.05, meaning, for this particular test, that, in the average, an algorithm will be erroneously judged superior to another one no more than 5% of the cases.

7.1.3 Results in Problems From the Machine Learning Literature

Table 7.1 lists the average error for the `C4.5`, `smog` and `muesli` algorithms. For the `smog` and `muesli` algorithms, the *average difference* and the *difference variance* between the generalization accuracy obtained by them and the generalization accuracy obtained by `C4.5` is also shown. A positive difference means that `C4.5` showed a larger error than the algorithm in that column in a particular problem. These values were used to evaluate the statistical significance of the differences observed. `C4.5`, `smog` and `muesli` were run using the default parameters.

A circle in a given row marks the algorithm that obtained the lowest average error in the given problem. A filled circle means that the difference observed is statistically significant (when compared with `C4.5`).

Problem	C4.5	smog		muesli	
	Error	Error	Aver. diff. $\pm\sigma$	Error	Aver. diff. $\pm\sigma$
dnf1	21.13	20.10	1.04 ± 1.15	• 18.04	3.09 ± 0.97
dnf2	18.52	10.84	7.69 ± 1.28	• 10.50	8.03 ± 1.25
dnf3	12.70	• 8.14	4.56 ± 1.05	8.49	4.21 ± 1.32
dnf4	32.93	20.26	12.67 ± 2.10	• 18.55	14.38 ± 0.77
mux6	1.38	• 0.19	1.19 ± 0.89	1.95	-0.57 ± 1.19
mux11	18.52	• 4.99	13.53 ± 1.67	16.27	2.25 ± 2.15
par4_16	34.86	• 0.77	34.09 ± 5.79	24.50	10.36 ± 5.88
par5_32	45.10	• 20.60	24.50 ± 4.49	44.47	0.63 ± 0.46
monk1	6.03	• 0.56	5.48 ± 1.34	3.66	2.38 ± 1.96
monk2	29.37	• 12.82	16.55 ± 1.49	18.49	10.88 ± 1.43
monk3	0.96	• 0.44	0.53 ± 0.58	0.70	0.26 ± 0.85
vote	3.76	• 2.85	0.91 ± 0.33	3.08	0.68 ± 0.48
mushroom	1.68	1.41	0.27 ± 0.68	• 1.16	0.51 ± 0.41
splice	10.45	10.09	0.36 ± 0.83	• 6.93	3.52 ± 0.57
tictactoe	7.90	• 4.33	3.57 ± 0.60	7.22	0.68 ± 0.60
breast	3.76	3.62	0.14 ± 0.50	• 3.32	0.44 ± 0.50
kkp	9.81	• 1.44	8.37 ± 1.77	5.96	3.85 ± 2.14
krkp	2.88	○ 2.74	0.13 ± 0.30	2.96	-0.09 ± 0.46
heel9	7.78	• 1.60	6.17 ± 0.84	2.12	5.66 ± 1.39
heel	24.03	• 2.52	21.51 ± 1.30	4.31	19.73 ± 2.04
sm12	9.96	• 2.01	7.94 ± 0.67	3.72	6.23 ± 0.88
str18	10.95	• 5.95	5.01 ± 1.14	6.83	4.12 ± 1.33

Table 7.1: Average errors for C4.5, smog and muesli

Figures 7.1, 7.2 and 7.3 show the average learning curves observed for the problems *par5_32*, *mux11* and *kkp*, this last one being the problem used as an example in section 1.1.1. Each curve represents the average error observed for the 10 experiments performed. Some of these results are much better than any ones previously published. For example, the problem *par5_32* was solved exactly for all experiments when the training set size reached 800 while Pagallo and Haussler reported that they were unable to solve this problem using the constructive induction algorithm fringe even using training sets of size 4000 [69].

The superior performance of smog in problems like *mux11* is partly due to its ability to select the right ordering. For this problem, the value of the output is controlled by two sets of bits in the input: 3 control bits and 8 data bits. The 3 control bits select which one of the data bits defines the output value. Other authors have noted [77] that this problem

is very hard for decision tree algorithms because the mutual information heuristic tends to select the data bits first, while the minimal decision tree (or graph) should test the control bits first. The reordering heuristics used in smog correctly identify the right ordering even with very small training sets and account for the large differences in performance observed.

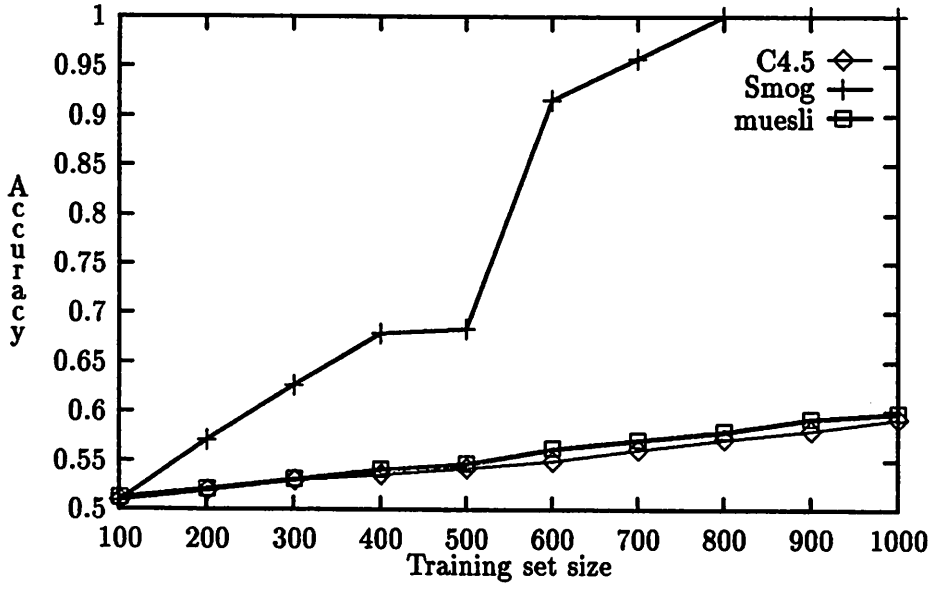


Figure 7.1: Learning curves for the par5_32 problem

7.1.4 Results in the Wright Laboratory Benchmark Set

The results obtained in this set of problems were obtained by Timothy Ross and the Pattern Theory Group at the Air Force Wright Laboratory. They kindly agreed to run smog in a benchmark his group assembled for the purpose of evaluating the efficacy of diverse learning algorithms.

Each one of the problems is defined over a space of 8 Boolean attributes. The setup is similar to the one used in section 7.1.3. One experiment consists of 10 independent runs with increasing training set sizes. The training set sizes selected are the multiples of 25 between 25 and 250. As before, 10 experiments were performed for each problem.

Tables 7.2 and 7.3 on pages 120 and 121 show the results obtained for the problems in this set. The meaning of the column labels is the same as before. The results listed for the program `C4.5` were obtained with the best combination found for the several options allowed by this program [34]. Since these problems are known to be noise free smog was

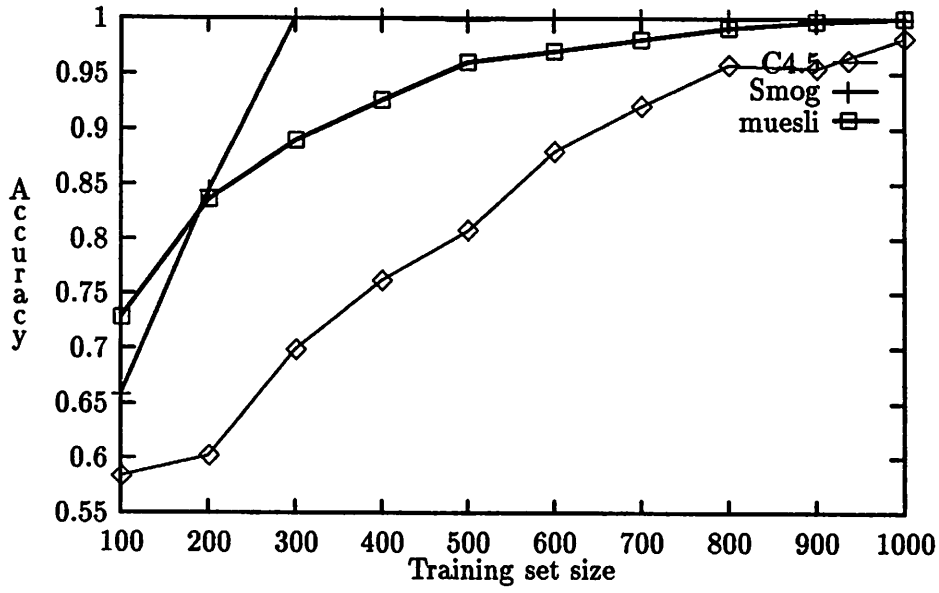


Figure 7.2: Learning curves for the mux11 problem

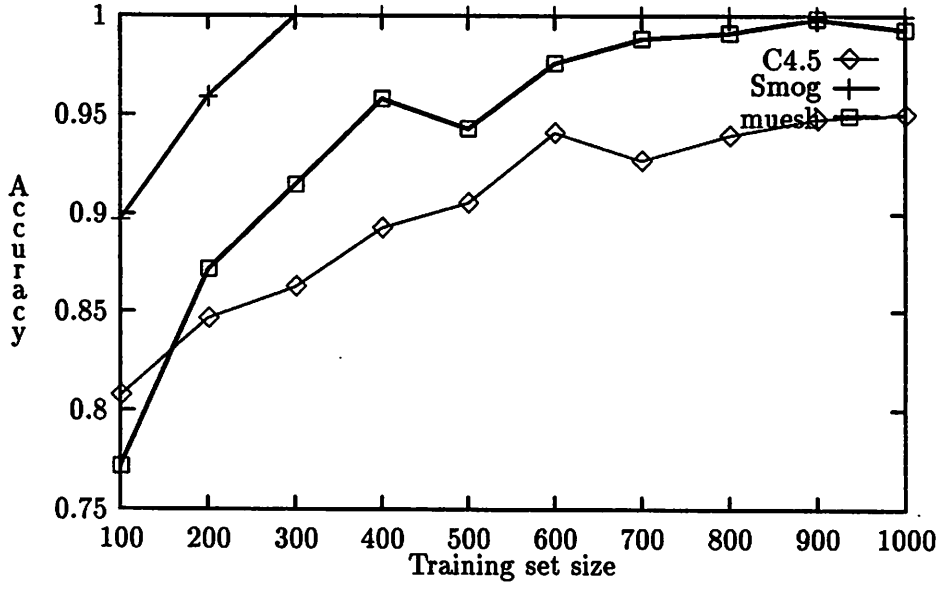


Figure 7.3: Learning curves for the king+pawn vs. king problem

run with the parameter α set to a value close to 0. Tables 7.2 and 7.3 show that the generalization error obtained with **smog** was smaller in a statistical significant way for a large number of problems. `C4.5` was significantly superior only for a set of 3 functions that are randomly generated but have a very different number of positive and negative examples (see appendix B). Apparently, with the parameters used, `C4.5` is better at detecting an unpredictable function and simply predicting the class of unclassified samples as the more commonly observed one.

This behavior can also be obtained with **smog** if the default value for α is used. The last column in tables 7.2 and 7.3 shows the results obtained with **smog** when the parameter α in equation (2.15) is set the default value. Setting a higher value for this parameter makes it easier for **smog** to generalize better in this type of random functions.

Other approaches were also tested in this benchmark. In particular, an inference algorithm based on the popular two-level minimizer **espresso** and the nearest-neighbor classification algorithm. This algorithm classifies unseen samples in the same class as the closest available sample in the training set and is commonly used as a standard of comparison. The graph in figure 7.4 shows a plot of the generalization error for each of the problems in tables 7.2 and 7.3. The problems were sorted in increasing order of generalization error for the **smog** algorithm to make the plot more readable. Algorithms with smaller errors exhibit a curve closer to the y axis in this graph. The plot shows that the **smog** algorithm shows a smaller generalization error for the majority of the problems in this benchmark.

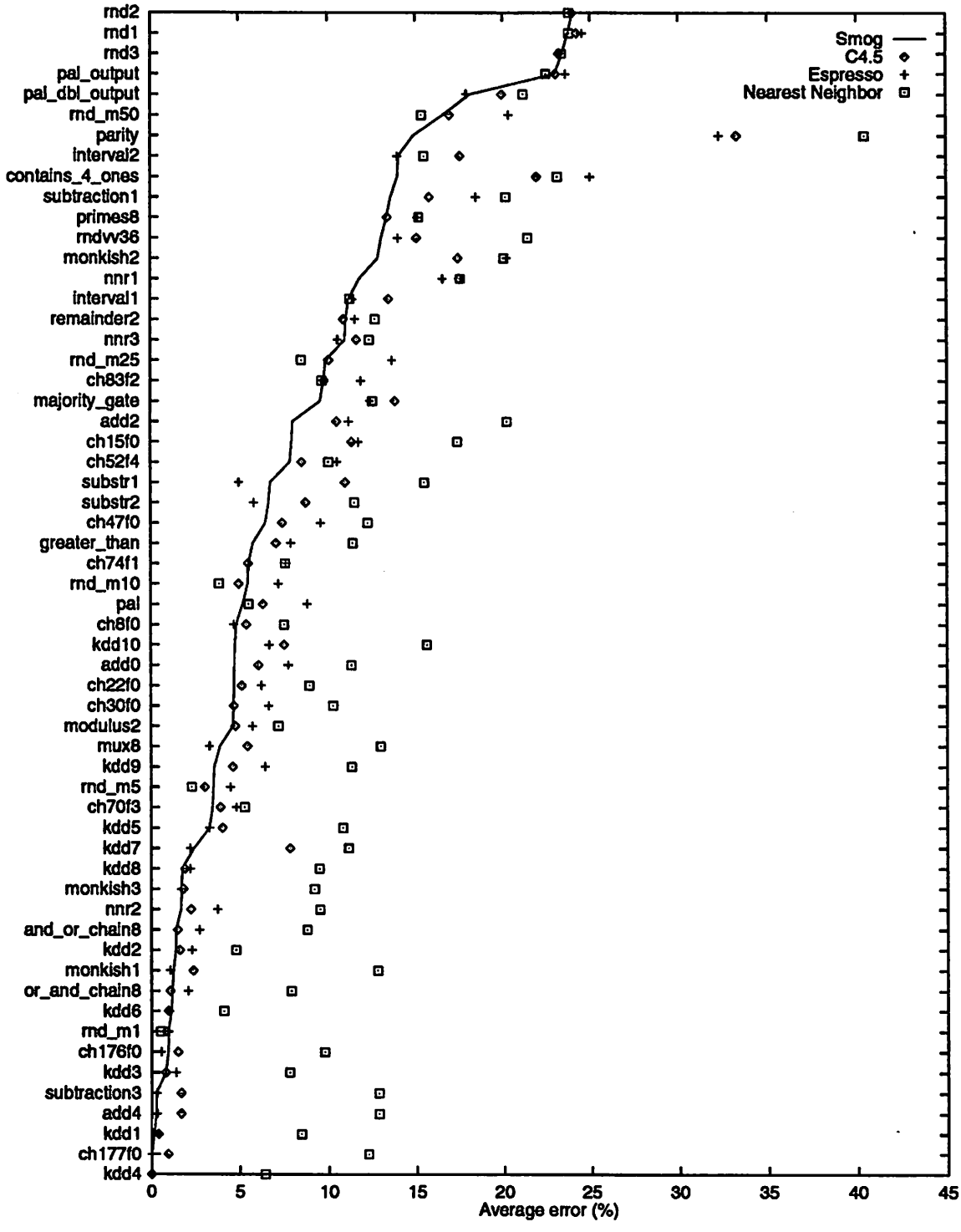


Figure 7.4: Generalization errors for `smog`, `c4.5`, `espresso` and *nearest neighbor*

Problem	C4.5	smog ($\alpha = 0.01$)		smog ($\alpha = 0.5$)
	Error	Error	Aver. diff. $\pm\sigma$	Error
add0	6.09	• 4.73	1.35 ± 0.80	4.95
add2	10.49	• 8.02	2.46 ± 2.12	8.03
add4	1.68	• 0.25	1.43 ± 1.67	0.25
and_or_chain8	1.50	○ 1.40	0.10 ± 0.35	1.59
ch15f0	11.34	• 7.96	3.38 ± 1.48	8.18
ch176f0	1.50	○ 0.94	0.56 ± 1.78	0.97
ch177f0	0.94	• 0.00	0.94 ± 1.33	0.00
ch22f0	5.15	○ 4.71	0.44 ± 1.00	4.34
ch30f0	4.70	○ 4.68	0.02 ± 0.89	4.57
ch47f0	7.43	• 6.48	0.94 ± 1.34	6.38
ch52f4	8.52	• 7.86	0.65 ± 0.87	8.04
ch70f3	3.94	• 3.48	0.46 ± 0.50	3.46
ch74f1	○ 5.52	5.53	-0.01 ± 0.78	5.57
ch83f2	9.79	○ 9.76	0.03 ± 0.73	9.07
ch8f0	5.41	• 4.83	0.58 ± 0.84	8.16
contains_4_ones	21.95	• 13.99	7.96 ± 0.79	13.19
greater_than	7.09	• 5.79	1.31 ± 0.83	5.99
interval1	13.46	• 11.18	2.29 ± 0.82	11.46
interval2	17.55	• 14.04	3.51 ± 0.81	14.45
kdd1	0.38	○ 0.13	0.25 ± 0.79	0.13
kdd2	1.61	○ 1.38	0.23 ± 0.42	1.69
kdd3	0.81	0.81	0.00 ± 0.00	0.88
kdd4	0.00	0.00	0.00 ± 0.00	0.00
kdd5	4.05	• 3.31	0.74 ± 0.47	3.51
kdd6	○ 0.97	1.13	-0.16 ± 0.47	5.34
kdd7	7.84	• 2.43	5.41 ± 0.78	2.43
kdd8	1.92	○ 1.75	0.17 ± 0.92	1.91
kdd9	4.64	• 3.59	1.05 ± 1.07	3.34
kdd10	7.52	• 4.77	2.75 ± 1.06	4.90

Table 7.2: Average errors for the Wright Labs benchmark, part 1

Problem	C4.5	smog ($\alpha = 0.01$)		smog ($\alpha = 0.5$)
	Error	Error	Aver. diff. $\pm\sigma$	Error
majority_gate	13.82	• 9.57	4.25 ± 0.66	9.84
modulus2	4.78	○ 4.64	0.14 ± 0.28	4.76
monkish1	2.38	• 1.25	1.12 ± 0.87	1.25
monkish2	17.43	• 12.85	4.57 ± 1.12	12.73
monkish3	1.84	○ 1.72	0.12 ± 0.34	1.59
mux8	5.47	• 3.91	1.56 ± 1.41	4.02
nnr1	17.50	• 11.80	5.70 ± 1.07	12.21
nnr2	2.25	○ 1.69	0.56 ± 1.19	1.75
nnr3	11.63	• 10.96	0.66 ± 0.77	11.15
or_and_chain8	○ 1.07	1.19	-0.12 ± 0.32	1.35
pal	6.34	• 5.21	1.13 ± 1.11	3.88
pal_dbl_output	19.95	• 18.13	1.82 ± 0.93	18.75
pal_output	23.04	○ 23.00	0.04 ± 0.50	23.60
parity	33.28	• 14.88	18.40 ± 3.26	13.88
primes8	13.39	○ 13.35	0.04 ± 0.89	13.52
remainder2	○ 10.88	11.01	-0.13 ± 0.85	10.71
rnd1	24.25	• 23.75	0.50 ± 0.48	24.42
rnd2	○ 23.99	24.02	-0.04 ± 0.61	24.71
rnd3	○ 23.25	23.43	-0.18 ± 0.55	24.21
rnd_m1	• 0.84	0.96	-0.11 ± 0.10	0.39
rnd_m5	• 3.05	3.54	-0.48 ± 0.42	2.05
rnd_m10	• 4.99	5.49	-0.50 ± 0.69	3.86
rnd_m25	10.05	○ 9.88	0.18 ± 0.63	9.27
rnd_m50	16.95	○ 16.58	0.37 ± 0.84	16.26
rndvv36	15.07	• 13.05	2.01 ± 1.17	13.24
substr1	10.96	• 6.76	4.20 ± 1.06	7.23
substr2	8.75	• 6.66	2.09 ± 1.10	6.87
subtraction1	15.80	• 13.59	2.21 ± 1.07	13.56
subtraction3	1.68	• 0.25	1.43 ± 1.67	0.25

Table 7.3: Average errors for the Wright Labs benchmark, part 2

7.1.5 Analysis

The results presented in the previous sections have shown that the algorithms described in chapter 5 and implemented in the program smog exhibit, for many problems of interest, a higher performance accuracy than standard decision tree algorithms like C4.5. The algorithms for the inference of Boolean networks also show good generalization accuracy, but are less robust when the value of the class label is poorly correlated with the attribute values.

It must be pointed out that, for problems that contain non Boolean attributes, the performance of C4.5 is not the same when it uses the original representation and the *binarized* version that contains only Boolean valued attributes. However, most problems did not require this transformation, and, in particular, none in the second benchmark set did. The setup used did not make use of the superior flexibility of C4.5 in the handling of multi-valued attributes. If this turns out to be important, it may be interesting to generalize smog to handle multi-valued attributes directly, or, at least, have the option to do so in specific circumstances.

7.2 Application to Handwritten Character Recognition

7.2.1 Problem Description

This section describes the application of the muesli and smog algorithms to a real world problem that is much more complex than any of the examples used in the comparisons performed in the previous section. The problem addressed is the recognition of handwritten digits introduced in section 1.1.1. The input data used was obtained from the National Institute of Standards and Technology (NIST) database of segmented handwritten characters [28] available in CDROM.

The handwritten characters in this database were obtained by scanning a large number of forms filled by volunteers and were assembled under the auspices of NIST. This database contains a total of 223,125 images of handwritten digits. This images have been segmented and labeled and each file contains the image of a single digit digitized in a 256 by 256 binary grid. Figure 1.4 in page 5, used to introduce this problem as a particular example of a inductive inference problem, shows an example of some characters as they are present in the database. The segmentation and the classification was manually checked and

corrected and the error rate of the segmented character files is less that 0.1%. However, the segmented images sometimes contain noise, like, for instance, the digit 1 in figure 1.4.

From this database, a training set of 53339 digits was selected together with an independent set of 52467 digits that was used to evaluate the accuracy obtained by the algorithms.

7.2.2 Pre-processing and Encoding

The images in the database were subject to a standard pre-processing procedure that is commonly used by the majority of the algorithms used for this problem [54]. First, they were de-skewed. This procedure identifies the angle of the principal component of the image and performs a transformation of the image using a linear operator that has the net effect of normalizing the characters with respect to the angle they were written initially.

The de-skewed characters were then normalized to a 16 by 16 grid. This normalization was performed by finding the bounding box of each image and performing a new linear transformation that reduced all characters to the same size.

Finally, the resulting images were discretized to binary values using a fixed threshold value. This last step is not commonly used because other algorithms like, for instance, neural networks, can handle continuous valued attributes directly. Examples of the data that result from this transformations are presented in figure 7.5.

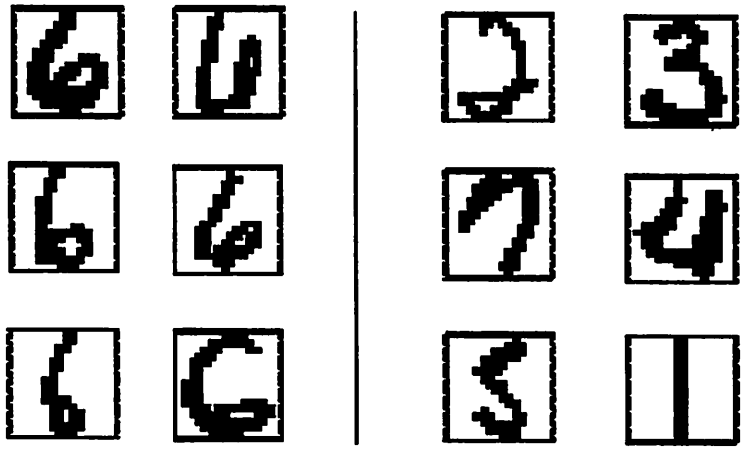


Figure 7.5: Discretized version of the digit recognition problem

The above discretization procedure has already mapped the original problem into

a 256 variable Boolean problem. However, since the problem is a multi-class classification, an appropriate encoding of the outputs has to be performed. The outputs were encoded using a 15 bit Hadamard error correcting code, the third option described in section 1.1.4. Table 7.4 shows the correspondence between the class defined by each digit and the desired values of the 15 outputs for this particular code. The reader may verify that this code

Digit	Code
0	000000000000000
1	101010101010101
2	011001100110011
3	110011001100110
4	000111100001111
5	101101001011010
6	011110000111100
7	110100101101001
8	000000011111111
9	101010110101010

Table 7.4: Digit encoding using a 15 bit error-correcting code

ensures a minimum Hamming distance of 8 between any two codewords.

7.2.3 Results

A 4 processor DEC/alpha machine was used to run these examples. The `muesli` program was run for a period of 24 hours and the best solution found at that time was used. The algorithm was still increasing the size of the network and better results might have been obtained with a longer execution time, but, for a problem of this size, the rate of the improvement becomes very slow. The smog algorithm was also run on this problem, using the fast mode described in section 5.4.4.

For this problem, there is an extra degree of freedom when generating the classification of a given input because the classifier may decide not to classify a character if it doesn't have enough confidence in the classification obtained. Such an action is termed a rejection. The ability to perform rejections is very important because, in real applications, the consequence of a rejection is usually less severe than the consequence of a wrong classification. For example, if used in automatic mail sorting, a classifier that cannot classify the

ZIP code of a given letter will cause such a letter to be sent to a human for classification. On the other hand, a classifier that mis-classifies the ZIP code will cause the letter to be sent to the wrong post office, at a much higher cost.

For this problem, the objective is to achieve the lowest possible rate of rejection with an error rate smaller than a given constant. Usually, the minimum rate of rejection for an error rate of 1% is the desired objective.

The use of an error correcting code allows for a tradeoff between the number of digits rejected and the number of errors committed. Specifically, the classifier may decide to reject a given digit if the output code obtained for that digit is too far away from the nearest valid codeword.

Table 7.5 and the graph in figure 7.6 show the error rates and the corresponding rejection rates in the training and test sets for the classifiers derived by `muesli` and `smog`. These rates are plotted against the maximum distance from a valid codeword allowed for non-rejection. If outputs that are at a larger distance from a valid codeword are accepted, less digits will be rejected but the classifier will have an higher error rate because classifications performed with a lower degree of certainty will be accepted as valid.

Distance from codeword	`muesli`		`smog`	
	Error (%)	Rejected (%)	Error (%)	Rejected (%)
0	0.1128	40.30	0.0381	38.95
1	0.4457	22.76	0.1334	20.74
2	1.2905	13.16	0.5127	11.58
3	3.2005	5.94	1.7840	5.42

Table 7.5: Error and rejection rates for the NIST database

The accuracy obtained by these approaches is very good although is doesn't match the best approaches proposed to date. These solutions [68] make heavy use of the characteristics of the domain and use the full set of 223,125 digits as the training set. Depending on the test set used, raw error values between 0.3% and 3.2% have been reported. Given this variations and the different sizes of the training sets used, a direct comparison is impossible. The classifier derived by the `muesli` algorithm has approximately a 13% rejection rate for a 1.3% error rate, obtained by rejecting any instances that cause an output at distance larger than 2 from a valid codeword. The classifier obtained using `smog` will reject

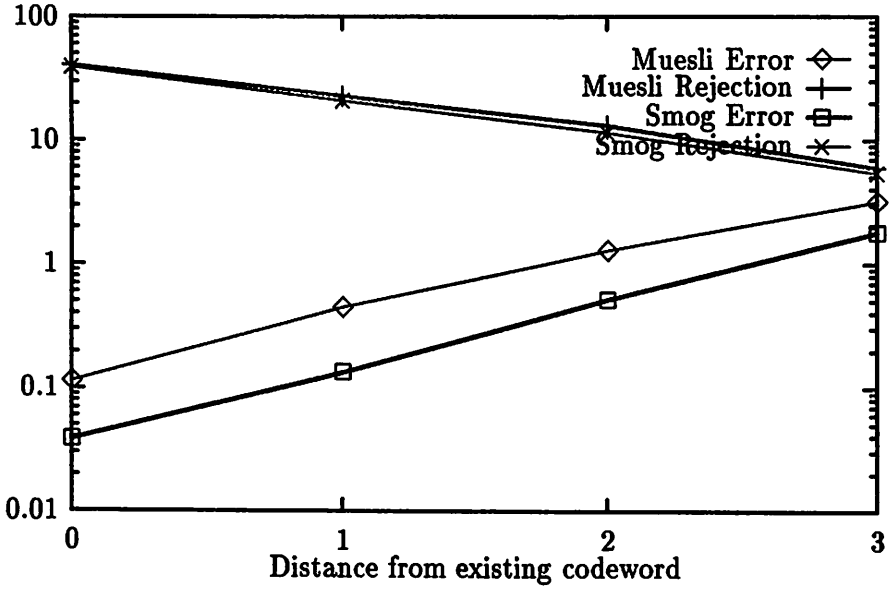


Figure 7.6: Error and rejection rates in test set vs. maximum distance from valid codeword

11.5% at an error rate of 0.51% and 5.42% at an error rate of 1.78%. These results can be further improved by using a variety of techniques that were proposed by other researchers in this problem. In fact, the performance of `smog` and `muesli` is severely hampered by the fact that no domain specific information or topological information of any sort was used. In contrast, all the approaches that obtain higher accuracy use large amounts of domain specific knowledge and, in some cases, hand-engineered features to improve generalization accuracy. None of these techniques were used because the main objective was not to obtain a competitive solution but merely to illustrate the applicability of the techniques developed to real world problems.

7.2.4 VLSI Implementation

One of the key characteristics of the majority of the approaches presented in this dissertation is that the classifiers obtained can be easily implemented using standard digital technologies. To illustrate this procedure, a VLSI implementation of the handwritten character recognizer was performed. The classifier obtained by `muesli` was selected because of its smaller size. In fact, the network obtained by mapping the classifier obtained using the `smog` algorithm into primitive Boolean functions is considerably larger than the one obtained directly by applying the `muesli` algorithm. While the combinational block obtained

using smog contained 36600 literals and used an area of 337 sq. mm[3] the combinational block obtained using muesli had only 11000 literals and used an area of 89 sq. mm.

The Boolean network obtained by `muesli` was optimized using standard logic synthesis optimization techniques and mapped into a standard cell structure using the `octtools` system [96], a package for the logic and physical design of VLSI circuits developed at Berkeley. The logic optimizations performed didn't change the logic functions implemented by the network because, in this step, the flexibility given by the use of don't care information was not used.

The complete system consists of the combinational block generated by `muesli` and some additional shift-registers that are required to store the data as it is scanned in and out. Figure 7.7 shows the final result obtained, a chip with 120 sq. mm. The large cell on the top is the combinational logic generated automatically by the learning algorithm. The small blocks in the bottom are the shift-registers used to store the data as it is shifted in. The chip has 15 data outputs, 16 data inputs, 2 control signals used to scan the data in and power supply pads. It is interesting to note that the routing and empty area of this chip represents 95% of the total chip area. This value is unusually high and accounts for the large size of the resulting circuit. This is due to the absence of locality in the network generated by the learning algorithm. The introduction of constraints that can restrict the type of connectivity allowed in the generated networks is an interesting direction for future research and has the potential to greatly reduce the overall size of the resulting system.

[3]Assuming 0.8 μ technology is used.

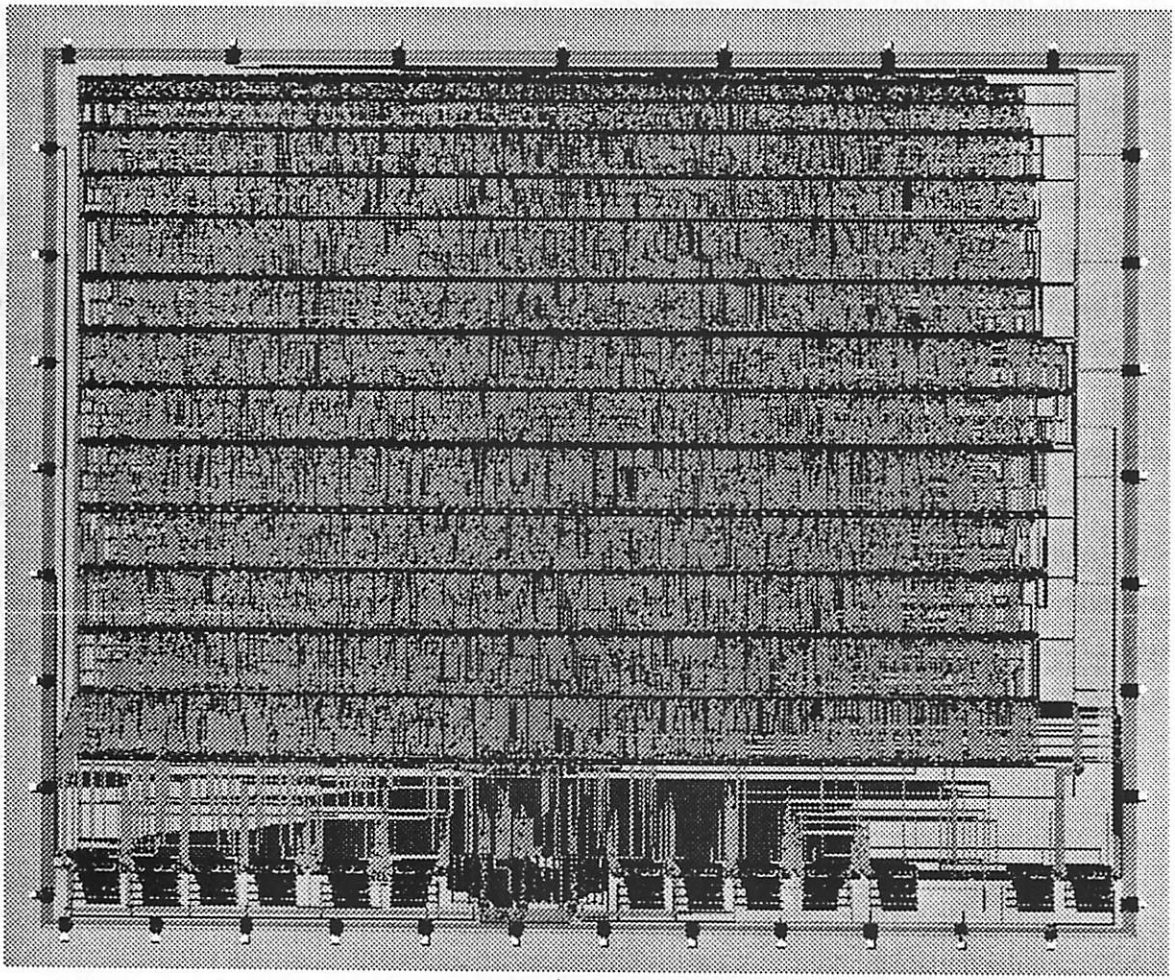


Figure 7.7: Chip layout for the handwritten recognition system

7.3 Application to Image Processing

The second large application used to exemplify the application of the algorithms was an image processing task. For this task, only the `muesli` algorithm was used as a compact final implementation of the final classifier is important for the interest of this approach.

7.3.1 Problem Description and Encoding

Although this problem was artificially created, it represents an application that may be interesting for this type of approach because of the need to process rapidly large amounts of instances, which, in this case, correspond to pixels.

The objective is to reconstruct an original image from a corrupted version of that image. In this experiment, 16 level gray scale images obtained by scanning sections of bank notes were corrupted by switching each bit with 5% probability. Samples of this image were used to train a network in the reconstruction of the original image. The training set consisted of 5x5 pixel regions of corrupted images (100 binary variables per sample) labeled with the correct value of the center pixel. Figure 7.8 shows a detail of the reconstruction performed in an independent test image by the network obtained.

7.3.2 VLSI Implementation

The combinational network derived by the learning algorithm was again optimized using standard logic minimization procedures and mapped to a standard cell structure using the tools in the `octtools` suite. The resulting cell is shown in figure 7.9. It occupies an area of 60 sq. mm using 0.8μ technology and the routing area occupies 86.7% of the total area. Again, this fraction represents an unusually high fraction of the total area and the comments made in section 7.2 apply. In a real application, this cell would be used as a building block for a chip. Additional circuitry could include memory cells together with decoding logic or a bank of shift registers to hold sections of the image if holding the full image is not feasible. It would also need to include decoding logic that would generate the right pixel value from the values of the 15 outputs generated by the combinational logic.

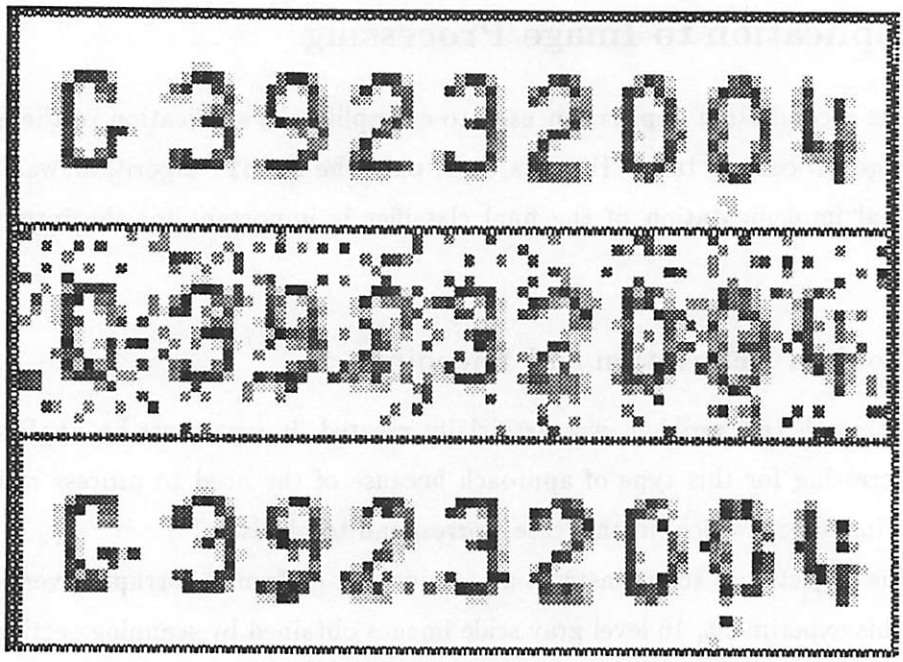


Figure 7.8: Image reconstruction experiment

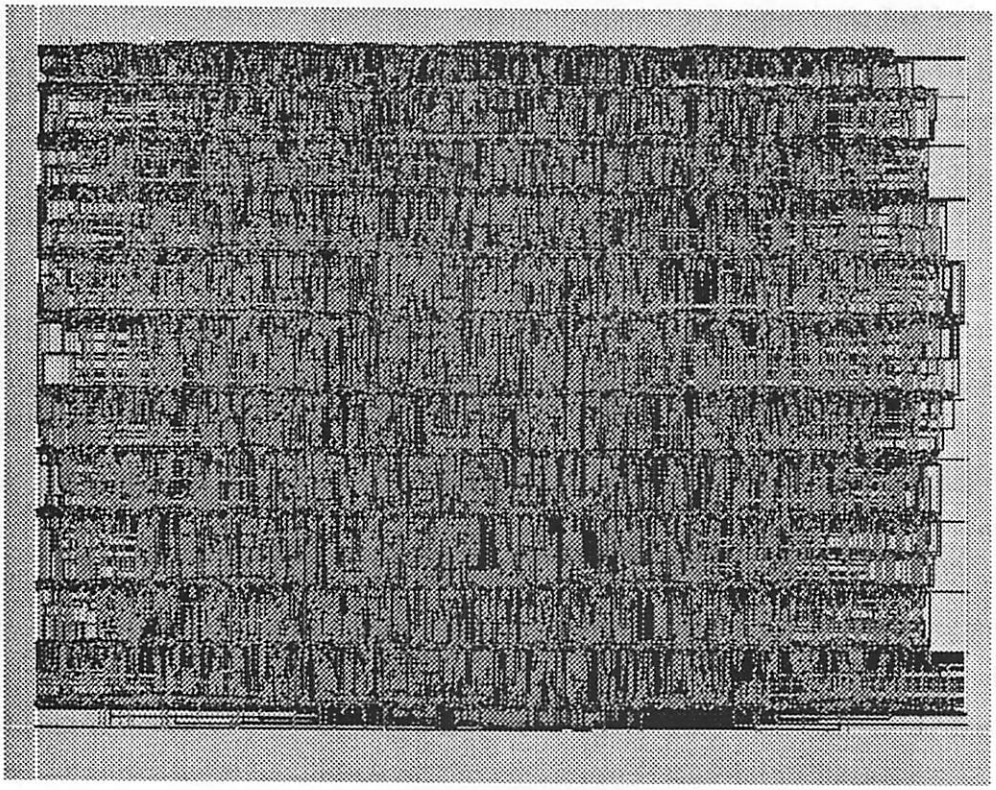


Figure 7.9: Standard cell layout for the image reconstruction network

Chapter 8

Conclusions

8.1 Conclusions

The main purpose of the research work described in this dissertation was to study the applicability of discrete optimization algorithms to the problem of inductive inference in discrete domains. Under fairly general assumptions, the inductive inference problem can be transformed into an optimization problem and solved using algorithms designed for this purpose. The general principle known as Occam's razor and the minimum description length principle provide the justification for this transformation.

Several different representations were studied and several algorithms were presented for the optimization of each of these representations. As is to be expected, not all representations and algorithms were equally effective and useful for the inference task they were designed for.

The heuristic algorithms for the optimization of reduced ordered decision graphs, implemented in the smog program, generated representations that were particularly robust in the type of problems they can handle and have shown the overall highest accuracy over the range of problems tested. These problems are representative of the type of problems addressed in the machine learning community. The comparisons performed between this algorithm and the algorithms most commonly used to perform induction from examples have shown that smog consistently outperformed them in terms of generalization accuracy. The performance of this algorithm can be influenced by a number of different parameters. The value of α, the parameter that controls the trade-off between accuracy in the training set and size of the solution did not have a strong influence in the average performance of

the algorithm but can change significantly the performance for particular problems. The particular strategy used to initialize the local optimization algorithm that is the center-piece of `smog` can also influence strongly the quality of the solutions obtained.

The greedy heuristic algorithms for the reduction of combinational Boolean networks presented in chapter 4 and implemented in the `muesli` program were, on the other hand, somewhat less effective in some of the hard problems that `smog` solves effectively. They are, however, much faster in some situations because they construct the solution by considering larger and larger networks while `smog` starts with a large decision graph and reduces it. This makes `muesli` interesting for problems that are very large and where an exact identification of the minimum consistent hypothesis is not critical to the success of the learner.

The algorithms for the synthesis of two-level threshold gate networks implemented in the program `lsat` were judged the least effective overall, both in terms of the generalization accuracy and in terms of computation requirements. It is an open question whether much better algorithms can be obtained for this problem or whether this results are closely related with the choice of that particular architecture. The success enjoyed by approaches based on neural network algorithms seems to show that there is nothing fundamentally wrong with the architecture in itself. However, networks obtained using standard neural network training algorithms are also unable to solve the harder problems where `lsat` performed less well, and, in fact, they fail even in some cases where `lsat` succeeds.

The exact algorithms for the synthesis of finite machines using implicit enumeration techniques also failed to meet the expectations in the sense they their performance is very similar to the one obtained using explicit algorithms published in the machine learning literature. In fact, a detailed comparison has shown almost no differences between the performance of the two approaches. The performance of the implicit algorithms does depend, however, on a variety of parameters like the variable ordering chosen and the type of representation used to manipulate Boolean functions. It remains an open question whether or not changes on some of these parameters can alter the scene significantly and change the relative performance of the two approaches. The exact approach described in chapter 6 is, however, important, because it explored the use of implicit techniques as one possible solution for the problem and made clear both the advantages and limitations of this alternative.

The results obtained in the two large problems addressed in the last part of chapter

7 have shown that the approaches described here can be competitive with alternative ones, specially if digital implementations of the resulting classifiers are important. For example, the application of `muesli` and `smog` to the character recognition problem generated classifiers that have a performance not very different from the one obtained with the best alternatives available. Furthermore, this gap in generalization accuracy can probably be bridged, at least partially, by the use of some domain specific knowledge.

8.2 Future Work

Many questions remain open and are appropriate topics for future research as the wide variety of representations and algorithms addressed in this research limited the amount of effort that was put in each individual algorithm and in possible avenues for improvement.

The limitation of the algorithms to discretely valued attributes is a serious one and limits the applicability of these algorithms to a wider variety of problems. Removing this restriction by finding a way to select appropriate splits in continuously valued variables is an important task that is left as future work. The current version can handle continuously valued attributes by selecting splits in a way similar to the one used in decision tree algorithms, but such a greedy approach is not likely to give the best results. In fact, in decision trees these splits are essentially independent but in decision graphs the idea is to select a split that can be reused. At present, it is not clear how this problem can be formulated and even less clear how it can be solved, but further research in this particular direction will produce interesting results.

Another open question that deserves further investigation is whether or not the performance of the implicit algorithms for the inference of finite state machines can be significantly improved by the use of alternative representations or different orderings. One possible alternative is the use of zero-suppressed RODGs as the support for the Boolean function manipulation routines. Zero-suppressed RODGs are interesting for this application because they are considered more efficient at representing the sparse sets that are likely to be the solutions of the problem. The use of zero-suppressed RODGs does not involve any major changes in the algorithm, but only a change in the package used for Boolean manipulation. Further research in this area may therefore yield very interesting results with a comparatively small investment.

Another interesting direction for future research is the application of implicit solu-

tion techniques to solve the exact RODG minimization problem using the exact algorithms described in chapter 5. Experiments have shown that these techniques can be very effective in the solution of similar problems that involve very high numbers of compatible sets. The application of implicit techniques to this problem should be immediate, given the similarity of the exact formulation presented in section 5.3 to the formulation obtained when the problem is the reduction of incompletely specified finite state machines. It remains to be seen, however, whether or not the implicit algorithms applied to this problem suffer from the same limitations they exhibit when applied to the reduction of tree finite state machines obtained from training sets using the approach described in chapter 6

Finally, heuristic approaches for the induction of small finite state machines consistent with the training set data are also important because, even if the exact approach can be improved, it will still be unable to find solutions for very large problems. Ultimately, this is the most important direction to follow if the minimum description length paradigm is to be applied to hypotheses represented as finite state machines. Large problems will require not only the selection of an approximate solution but also the ability to trade-off accuracy in the training set data for compactness of the generated hypotheses and these two objectives are not compatible with the choice of an exact algorithm for this task.

Finally, for problems where hardware implementations are sought, it would be interesting to change the cost function to take into account other costs involved like the communication complexity between modules in the solution. These considerations may lead to significant reductions in the final size of VLSI implementations by reducing the amount of space dedicated to routing.

Appendix A

Function Manipulation Using RODGs

A.1 Algorithms for RODG Manipulation

This section gives a brief overview of the algorithms that were developed for RODG manipulation and follows closely in form and content the work presented in [13]. For a much more complete description of the algorithms used, the interested reader should consult this reference. This first section is concerned with RODGs defined over Boolean spaces.

Each non-terminal node n in the RODG represents a Boolean function that is denoted by $f(n_i) = (v_i, f(n_i^{\text{then}}), f(n_i^{\text{else}}))$, where v_i is the variable tested at n_i and n_i^{then} and n_i^{else} are the nodes pointed to by the *then* and *else* edges, respectively.

The fundamental operation implemented by the RODG package is the *Ite* operator, defined as:

$$\text{Ite}(f, g, h) = fg \vee \overline{f}h \tag{A.1}$$

It is a simple exercise to verify that all the basic Boolean operations of two variables can be defined using the *Ite* operator with appropriate arguments. For example, $f = ab$ is equivalent to $f = \text{Ite}(a, b, 0)$ and $f = a \oplus b$ is equivalent to $f = \text{Ite}(a, \overline{b}, b)$.

Shannon's decomposition theorem states that

$$f = vf_v \vee \overline{v}f_{\overline{v}} \tag{A.2}$$

where v is a variable and f_v and $f_{\overline{v}}$ represent f evaluated at $v = 1$ and $v = 0$, respectively.

Now, let $f(n_i) = (w_i, f(n_i^{\text{then}}), f(n_i^{\text{else}}))$ and assume that v comes before w_i in the ordering or that $v = w_i$. Finding the cofactors of f with respect to v is trivial:

$$f_v = \begin{cases} f & \text{if } v \neq w_i \\ f(n_i^{\text{then}}) & \text{if } v = w_i \end{cases} \quad f_{\overline{v}} = \begin{cases} f & \text{if } v \neq w_i \\ f(n_i^{\text{else}}) & \text{if } v = w_i \end{cases} \tag{A.3}$$

The following recursive definition gives a simple algorithm for the computation a the function $z = \text{Ite}(f, g, h)$. Let v be the top variable of f, g, h. Then,

$$\begin{aligned} z &= v z_v \vee \overline{v} z_{\overline{v}} \\ &= v(fg \vee \overline{f}h)_v \vee \overline{v}(fg \vee \overline{f}h)_{\overline{v}} \\ &= v(f_v g_v \vee \overline{f}_v h_v) \vee \overline{v}(f_v g_{\overline{v}} \vee \overline{f}_{\overline{v}} h_{\overline{v}}) \\ &= \text{Ite}(v, \text{Ite}(f_v, g_v, h_v), \text{Ite}(f_{\overline{v}}, g_{\overline{v}}, h_{\overline{v}})) \\ &= (v, \text{Ite}(f_v, g_v, h_v), \text{Ite}(f_{\overline{v}}, g_{\overline{v}}, h_{\overline{v}})) \end{aligned} \tag{A.4}$$

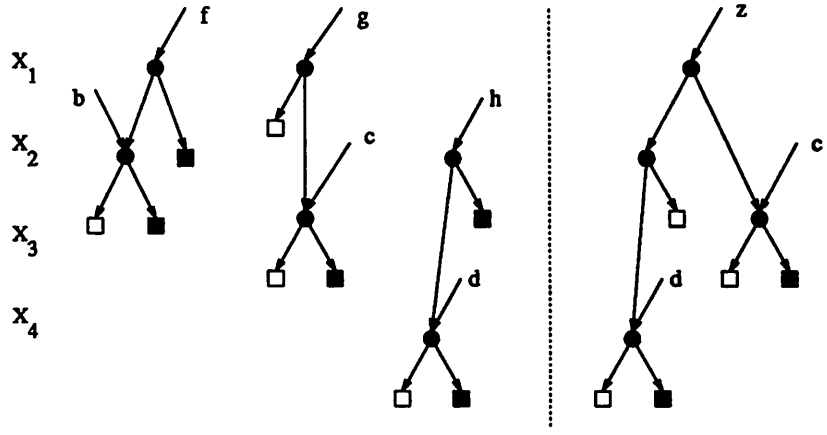
The terminal cases for this recursion are: $\text{Ite}(1, f, g) = \text{Ite}(0, g, f) = \text{Ite}(f, 1, 0) = f$.

A systematic exponential complexity of the procedure is avoided by keeping a table of existing functions. Each element in the table is a triple (v, g, h) and each node in the RODG corresponds to an entry in this table. Before applying the recursive definition (A.4) the algorithm checks to see if the desired function already exists.

Figure A.1 shows an example of the application of the recursive definition in the computation of the function $z = \text{Ite}(f, g, h)$. For clarity, several copies of the terminal nodes are shown. The reader should keep in mind that only one copy of each function is kept at any time. This is true for the terminal nodes and also for the nodes that implement the functions c and d, but depicting only one copy of these nodes would make the diagram too complex to be useful. In this example, the nodes that correspond to the functions c and d do not need to be created from scratch. Since they already exist they are shared by different functions.

A.2 Manipulating Boolean Functions Using RODGs

For a given ordering of the variables, reduced ordered decision graphs are a canonical representation for functions in that domain [18]. This means that given a function $f : \{0,1\}^N \rightarrow \{0,1\}$ and an ordering of the variables, there is one and only one representation for the function f.



$$
\begin{aligned}
z &= \mathrm{Ite}(f, g, h) \\
&= (x_1, \mathrm{Ite}(f_{x_1}, g_{x_1}, h_{x_1}), \mathrm{Ite}(f_{\overline{x_1}}, g_{\overline{x_1}}, h_{\overline{x_1}})) \\
&= (x_1, \mathrm{Ite}(1, c, h), \mathrm{Ite}(b, 0, h)) \\
&= (x_1, c, (x_2, \mathrm{Ite}(b_{x_2}, 0_{x_2}, h_{x_2}), \mathrm{Ite}(b_{\overline{x_2}}, 0_{\overline{x_2}}, h_{\overline{x_2}}))) \\
&= (x_1, c, (x_2, \mathrm{Ite}(1, 0, 1), \mathrm{Ite}(0, 0, d))) \\
&= (x_1, c, (x_2, 0, d))
\end{aligned}
\tag{A.5}
$$

Figure A.1: Computation of $\mathrm{Ite}(f, g, h)$

Packages that manipulate reduced ordered decision graphs are widely available and have become the most commonly used tool for discrete function manipulation in the logic synthesis community [16]. Some of these packages are restricted to Boolean functions [13] (each non-terminal node has exactly two outgoing edges) while others [46] can accept multi-valued attributes.

All these packages provide at least the same basic functionality: the ability to combine functions using basic Boolean and arithmetic operations and the ability to test for containment or equivalence of two functions. They also provide an array of more complex primitives for function manipulation that are not relevant for the work presented here.

Several functions can be represented using a single RODG and each function is usually represented by a pointer to the RODG node that represents the function. Due to the canonicity property described above, the equivalence test (and, therefore, the tautology test) can be performed in constant time. This means that the task of checking two functions represented by their RODGs for equivalence is a trivial one because it reduces to the comparison of two pointers[1].

The algorithms described in this paper make use of only a small fraction of the facilities provided by RODG packages. In particular, we will only use the following primitives for Boolean function manipulation:

- Boolean combination of two existing functions. For example, $f := g \wedge h$ returns a function f that is the Boolean *and* of two existing functions, g and h.
- Complement of an existing function. Example: $f := \overline{g}$.
- Creation of a function from an existing variable. For example, $f :=$ Fvar(i) returns a function f that is 1 when variable v_i is 1 and is 0 otherwise.
- The *if-then-else* operator. For example, $f :=$ Ite(v, g, h) returns the function g for the points where function v is 1 and the function h for the points where v is 0. Although the *Ite* operator is simply a shorthand for the combination $f := (v \wedge g) \vee (\overline{v} \wedge h)$, it is used so often that is deserves separate treatment.

The previous section described how RODG packages manipulate internally Boolean function representations. However, for the purposes of understanding many the algorithms

[1]The reader should not be surprised that a complex problem such as function equivalence check can be solved in constant time once the RODGs for the functions are known. The process of building the RODGs involved may require, in itself, exponential time.

that use them as a tool to represent and manipulate Boolean functions, it is sufficient to understand how the facilities provided by these packages can be used to manipulate Boolean functions.

The pseudo-code in figure A.2 exemplifies how the function $f : \{0,1\}^4 \rightarrow \{0,1\}$ defined below can be obtained using the primitives provided by the package. Figure A.3 shows the successive RODGs created by the package to represent the functions g, h and f defined in (A.6).

$$f(x_1, x_2, x_3, x_4) = \begin{cases} x_2 \wedge x_4 & \text{if } x_1 = 1 \\ \overline{x_3} \vee x_4 & \text{if } x_1 = 0 \end{cases} \tag{A.6}$$

```
Main()
   RODG f,g,h                          Declares f,g and h as functions
   InitializePackage()
   g := Fvar(2) ∧ Fvar(4)              Computes x2 ∧ x4
   h :=!Fvar(3) ∨ Fvar(4)              Computes x̄3 ∨ x4
   f := Ite(Fvar(1),g,h)
```

Figure A.2: Computing the function f

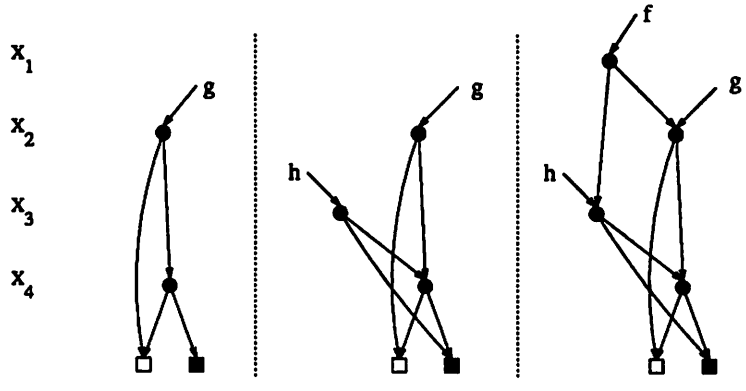


Figure A.3: Successive RODGs created to represent f

A.3 RODGs Defined Over Multi-valued Spaces

Any binary valued function of k discrete variables, $x_1, x_2, \ldots, x_k$

$$F : P_1 \times P_2 \times \cdots \times P_k \rightarrow \{0,1\} \tag{A.7}$$

can be represented by a slightly different type of RODG defined over multi-valued spaces. In the logic synthesis community, these RODGs are known as Multi-valued Decision Diagrams (MDDs). To avoid an awkward notation, this term will be used from now on. An MDD is also a rooted, directed, acyclic graph where each non-terminal node is labeled with the name of one variable. An MDD for F has two terminal nodes n_z and n_o that correspond to the leaves of the graph. Every non-terminal node n_i, labeled with variable v_j, has $|P_j|$ outgoing edges labeled with the possible values of x_j. Each of these edges points to one child node. The value of $\mathcal{F}$ for any point in the input space can be computed by starting at the root and following, at each node, the edge labeled with the value assigned to the variable tested at that node. The value of the function is 0 if this path ends in node n_z and 1 if it ends in node n_o.

The definitions of reduced and ordered are the same as for standard RODGs defined over Boolean spaces. For a given variable ordering, reduced, ordered MDDs are canonical representations for functions defined over that domain, thus implying that two functions can easily be checked for equivalence.

The implicit approach described in chapter 6 used the MDD package described in [46]. This MDD package provides an array of primitives for function manipulation. The reader is referred to that reference for a more detailed description of these primitives.

Apart from the operations supported by Boolean RODG packages, MDD packages also support the creation of functions that express arithmetic relations between variables like. For instance, $f := (x_i = x_j)$ returns the function that is 1 for all points of the input space where $x_i = x_j$.

Figure A.4 depicts the MDDs for the function $f := (x \neq 3)$, $g := (x = y)$ and $h := f \wedge g$, all defined over $P \times P$, $P = \{1,2,3\}$.

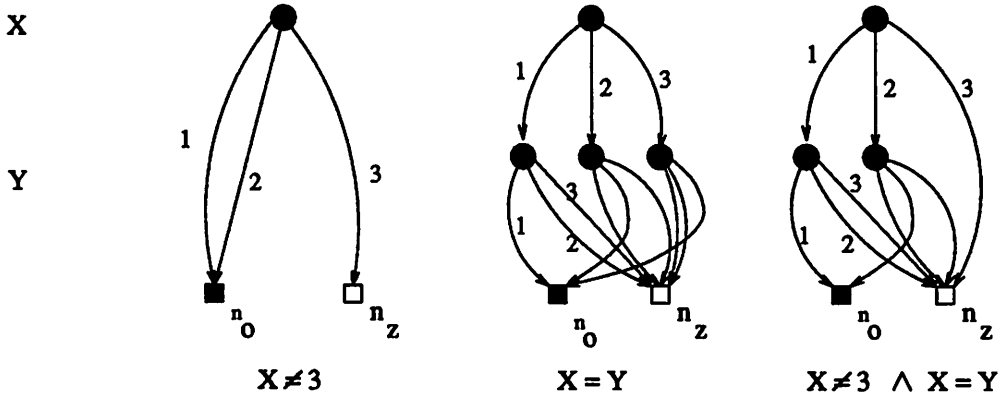


Figure A.4: Graphic representation of the MDDs for functions $X \neq 3$, $X = Y$ and $X \neq 3 \wedge X = Y$.

Appendix B

Description of the Problems Used

B.1 Problems from the Machine Learning Literature

The majority of the problems used have been studied before by other researchers. The problems *dnf1*, *dnf2*, *dnf3*, *dnf4*, *par4_16*, *par5_32*, *mux6* and *mux11* were proposed in [69]. These concepts are identified by the functions implemented by the following Boolean formulas:

dnf1 : $f(x_1...x_{80}) = x_5x_{28}x_{38}x_{72}x_{74}x_{76}+x_2x_{16}x_{40}x_{52}x_{74}+x_{10}x_{21}x_{23}x_{28}x_{30}x_{63}+x_{40}x_{56}x_{58}x_{60}x_{63}x_{72}+ x_6x_{24}x_{36}x_{37}x_{39}x_{48} + x_3x_{17}x_{45}x_{55}x_{72}x_{75} + x_{11}x_{48}x_{50}x_{64}x_{69}x_{74} + x_2x_{15}x_{27}x_{36}x_{50}x_{53} + x_6x_{12}x_{22}x_{45}x_{60}$

dnf2 : $f(x_1...x_{32}) = x_1x_3x_{14}x_{19}x_{26}x_{35}x_{36}+x_8x_{15}x_{31}x_{37}+x_5x_{10}x_{14}x_{27}x_{29}+x_{18}x_{20}x_{30}x_{36}+ x_2x_3x_9x_{19}x_{24} + x_{24}x_{25}x_{27}x_{36}x_{37} + x_6x_7x_{14}x_{25}x_{26}x_{31}x_{34} + x_1x_6x_{22}x_{30}$

dnf3 : $f(x_1...x_{32}) = x_1x_2x_6x_8x_{25}x_{28}\overline{x_{29}}+x_2x_9x_{14}\overline{x_{16}}\ \overline{x_{22}}x_{25}+x_1\overline{x_4}\overline{x_{19}}\overline{x_{22}}x_{27}x_{28}+\overline{x_2}x_{10}x_{14}\overline{x_{21}}\overline{x_{24}}+ x_{11}x_{17}x_{19}x_{21}\overline{x_{25}} + \overline{x_1}\overline{x_4}x_{13}\overline{x_{25}}$

dnf4 : $f(x_1...x_{64}) = x_1x_4x_{13}x_{57}\overline{x_{59}}+x_{18}\overline{x_{22}}\overline{x_{24}}+x_{30}\overline{x_{46}}x_{48}\overline{x_{58}}+\overline{x_9}x_{12}\overline{x_{38}}x_{55}+\overline{x_5}x_{29}\overline{x_{48}}+ x_{23}x_{33}x_{40}x_{52} + x_4\overline{x_{26}}\ \overline{x_{38}}\ \overline{x_{52}} + x_6x_{11}x_{36}\overline{x_{55}} + \overline{x_6}\ \overline{x_9}\ \overline{x_{10}}x_{39}\overline{x_{46}} + x_3x_4x_{21}\overline{x_{37}}\ \overline{x_{55}}$

mux6 : $f(x_1...x_{16}) = \overline{x_1}\,\overline{x_2}x_3 + \overline{x_1}x_2x_4 + x_1\,\overline{x_2}x_5 + x_1x_2x_6$

mux11 : $f(x_1...x_{32}) = \overline{x_1}\,\overline{x_2}\,\overline{x_3}x_4 + \overline{x_1}\,\overline{x_2}x_3x_5 + \overline{x_1}x_2\,\overline{x_3}x_6 + \overline{x_1}x_2x_3x_7 + x_1\,\overline{x_2}\,\overline{x_3}x_8 + x_1\,\overline{x_2}x_3x_9 + x_1x_2\,\overline{x_3}x_{10} + x_1x_2x_3x_{11}$

xor4_16 : $f(x_1...x_{16}) = x_1 \oplus x_2 \oplus x_3 \oplus x_4$

xor5_32 : $f(x_1...x_{32}) = x_1 \oplus x_2 \oplus x_3 \oplus x_4 \oplus x_5$

Other concepts defined by simple Boolean expressions are the following:

dnfa : $f(x_1...x_6) = x_1\overline{x_2}\,\overline{x_6} + \overline{x_1}x_2x_3x_6 + x_3x_6$

dnfb : $f(x_1...x_7) = x_2\overline{x_3}\,\overline{x_4}x_5 + x_1x_4\overline{x_5}x_7 + \overline{x_1}\,\overline{x_2}x_4$

dnfc : $f(x_1...x_9) = \overline{x_2}x_3\overline{x_5}\,\overline{x_8}x_9 + \overline{x_1}\overline{x_2}x_3\overline{x_5}x_6\overline{x_8}x_9 + \overline{x_2}\,\overline{x_3}x_7x_8 + \overline{x_1}\,\overline{x_2}x_4\overline{x_5}\,\overline{x_6} + \overline{x_2}\,\overline{x_3}x_5x_6\overline{x_7}\,\overline{x_8}\,\overline{x_9}$

dnfd : $f(x_1...x_{11}) = \overline{x_2}\,\overline{x_3}x_4\overline{x_5}\,\overline{x_7}\,\overline{x_8}x_9 + \overline{x_4}x_5\overline{x_6}x_{11} + x_4\overline{x_5}\,\overline{x_6}\,\overline{x_{11}} + x_1x_4\overline{x_5}\,\overline{x_6}x_8\overline{x_9}\,\overline{x_{10}} + \overline{x_2}x_3x_4\overline{x_5}x_7\overline{x_8} + \overline{x_4}x_5x_6x_{10}x_{11} + \overline{x_2}x_3x_4\overline{x_5}x_8\overline{x_9}\,\overline{x_{10}} + \overline{x_2}\,\overline{x_3}x_4\overline{x_6}\,\overline{x_9} + x_2\overline{x_3}\,\overline{x_4}$

xor6 : $f(x_1...x_6) = x_1 \oplus x_2... \oplus x_6$

xor7 : $f(x_1...x_7) = x_1 \oplus x_2... \oplus x_7$

xor8 : $f(x_1...x_8) = x_1 \oplus x_2... \oplus x_8$

xor9 : $f(x_1...x_9) = x_1 \oplus x_2... \oplus x_9$

xor10 : $f(x_1...x_{10}) = x_1 \oplus x_2... \oplus x_{10}$

sm12 : $f(x_1...x_{12}) = (x_1x_2 + x_3x_4 + x_5x_6)(x_7x_8 + x_9x_{10} + x_{11}x_{12})$

str18 : $f(x_1...x_{18}) = (x_1x_2x_3 + x_4x_5x_6 + x_7x_8x_9)(x_{10}x_{11}x_{12} + x_{13}x_{14}x_{15} + x_{16}x_{17}x_{18})$

heel9 : $f(x_1...x_9) = x_1x_2x_3 + x_4x_5x_6 + x_7x_8x_9$

heel : $f(x_1...x_{18}) = x_1x_2x_3 + x_4x_5x_6 + x_7x_8x_9 + x_{10}x_{11}x_{12} + x_{13}x_{14}x_{15} + x_{16}x_{17}x_{18}$

Problems *monk1*, *monk2* and *monk3* were proposed in [98] and are the encoding of concepts in a hypothetic robot world.

The problems *tictactoe*, *vote*, *mushroom*, *breast* and *splice* are from the UCI database [62] and are described in detail in the online documentation publicly available.

The problems *krkp* and *kkp* result from the encoding of chess positions. The first one is described in [90] and is the encoding of a King+Rook vs. King+Pawn chess ending using high level attributes. The *kkp* problem is obtained from the encoding of the chess endings described in section 1.1.1.

B.2 Problems from the Wright Laboratory Set

This set of problems has been assembled by a research group at the Pattern Theory Program of the Air Force Wright Laboratory. A more complete description of these problems can be found in [34].

All functions are of the form $f(x_1...x_8) \rightarrow \{0,1\}$. The functions are intended to be representative of a wide variety of problems for testing machine learning systems.

- Randomly generated functions: *rnd1*, *rnd2*, and *rnd3*.
- Randomly generated functions with a fixed number of minority elements: *rnd_m1*, *rnd_m5*, *rnd_m10*, *rnd_m25* and *rnd_m50*.
- Random functions with irrelevant variables: *rmdvv36*.
- Boolean expressions [33] : *kdd1*, *kdd2*, *kdd3*, *kdd4*, *kdd5*, *kdd6*, *kdd7*, *kdd8*, *kdd9*, *kdd10*
- Multiplexer functions : *mux6*.
- Deep functions : *and_or_chain8* = $((((x_1x_2 + x_3)x_4 + x_5)x_6 + x_7)x_8)$
- Monkish Problems: 8 binary variable "approximations" to the Monk's problems [98]
- String functions. Palindrome acceptor and variants : *pal*, *pal_output* and *doubley*.
- Interval acceptors. Accept strings with a given number of sequences with only 0's or only 1's : *interval1* (3 or fewer sequences), *interval2* (4 or fewer sequences).
- Sub-string detectors : *substr1* (accepts inputs that contain *101*) and *substr2* (accepts inputs that contain *1100*.
- Pixel images : recognize center pixel given surrounding pixels of characters from the Borland font set. Problem chXfY means character *X* from font *Y*. The problems are: *ch8f0*, *ch15f0*, *ch22f0*, *ch30f0*, *ch47f0*, *ch176f0*, *ch177f0*, *ch74f1*, *ch83f2*, *ch70f3*, *ch52f4*.
- Symmetric functions : *parity*, *contains_4_ones*, *majority_gate*.
- Prime number recognition: *primes8*

- Numerical functions of two 4 bit numbers (the last number means what bit of the results is the output bit) : *add0*, *add2*, *add4*, *greater_then*, *subtraction1*, *subtraction 3*, *modulus2*, *remainder2*.

- Geometric functions : output determined by selecting closest template using Hamming distance : *nnr1* (four templates : {00000000 -, 00001111 +, 11110000 +, 11111111 -}), *nnr2* (two templates : {00111010 -, 11011110 +}), *nnr3* (eight templates : {00010011 +, 00111010 +, 01000101 -, 01011001 -, 01110010 +, 10001101 -, 11000111 +, 11011110 -})

Bibliography

[1] J.A. Anderson and E. Rosenfeld, editors. *Neurocomputing: Foundations of Research.* MIT Press, Cambridge, 1988.

[2] D. Angluin. On the complexity of minimum inference of regular sets. *Inform. Control*, 39(3):337–350, 1978.

[3] D. Angluin. Learning regular sets from queries and counterexamples. *Inform. Comput.*, 75(2):87–106, November 1987.

[4] William Armstrong and Jan Gecsei. Adaptation algorithms for binary tree networks. *IEEE Transactions on Systems, Man, and Cybernetics*, 9:276–285, 1979.

[5] Timur Ash. Dynamic node creation in backpropagation networks. *Connection Science*, 1:365–375, 1989.

[6] Les Atlas, Ronald Cole, Yeshwant Muthusamy, Alan Lippman, Jerome Connor, Dong Park, Mohamed El-Sharkawi, and Robert J. Marks II. A performance comparison of trained multi-layer perceptrons and trained classification trees. *IEEE Proceedings*, 1990.

[7] K. A. Bartlett, R. K. Brayton, G. D. Hachtel, R. M. Jacoby, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. Multi-level logic minimization using implicit don't cares. *IEEE Transactions on CAD*, 1988.

[8] S. Becker and Y. Le Cun. Improving the convergence of back-propagation learning with second order methods. In D. Touretzky, G. Hinton, and T. Sejnowski, editors, *Proceedings of the 1988 Connectionist Models Summer School*, pages 29–37, San Mateo, 1989. Morgan Kaufmann.

[9] A. W. Biermann and R. Krishnaswamy. Constructing programs from example computations. *IEEE Trans. on Software Engineering*, SE-2:141–153, 1976.

[10] A. W. B. R. I. Biermann and F. E. Petry. Speeding up the synthesis of programs from traces. *IEEE Trans. on Computers*, C-24:122–136, 1975.

[11] A. Blum and R. L. Rivest. Training a 3-node neural net is NP-Complete. In *Advances in Neural Information Processing Systems I*, pages 494–501. Morgan Kaufmann, 1989.

[12] B. Boser and E. Sackinger. An analog neural network processor with programmable network topology. In *ISSCC91 Digest of Technical Papers*. IEEE, 1991.

[13] K. Brace, R. Rudell, and R. Bryant. Efficient implementation of a BDD package. In *Design Automation Conference*, June 1989.

[14] R. Brayton, G. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, 1984.

[15] R. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. MIS: A multiple-level logic optimization system. *IEEE Transactions on Computer-Aided Design*, November 1987.

[16] R. K. Brayton, G. D. Hachtel, and A. L. Sangiovanni-Vincentelli. Multilevel logic synthesis. *Proceedings of the IEEE*, 78:264–300, 1990.

[17] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Wadsworth International Group, 1984.

[18] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, 1986.

[19] R. Carraghan and P. M. Pardalos. An exact algorithm for the maximum clique problem. *Operations Research Letters*, 9:375–382, November 1990.

[20] G. J. Chaitin. On the length of programs for computing finite binary sequences. *Journal ACM*, 16:145–159, 1969.

[21] Olivier Coudert, Christian Berthet, and Jean Christophe Madre. Verification of synchronous sequential machines based on symbolic execution. In J. Sifakis, editor, *Proceedings of the Workshop on Automatic Verification Methods for Finite State Systems*,

volume 407 of *Lecture Notes in Computer Science*, pages 365–373. Springer-Verlag, June 1989.

[22] S. Das and M. Mozer. A unified gradient-descent/clustering algorithm architecture for finite state machine induction. In *Advances in Neural Information Processing Systems 6*, Denver, CO, 1993. Morgan Kaufmann.

[23] T. G. Dietterich and G. Bakiri. Error-correcting output codes: A general method for improving multiclass inductive learning programs. In *Proceedings of the Ninth National Conference on Artificial Intelligence (AAAI-91)*, pages 572–577. AAAI Press, 1991.

[24] S.E. Fahlman and C. Lebiere. The cascade-correlation learning architecture. In D.S. Touretzky, editor, *Advances in Neural Information Processing Systems*, volume 2, pages 524–532, San Mateo, 1990. Morgan Kaufmann.

[25] M Frean. The upstart algorithm: A method for constructing and training feedforward neural networks. *Neural Computation*, 2:198–209, 1990.

[26] Steven J. Friedman and Kenneth J. Supowit. Finding the optimal variable ordering for binary decision diagrams. *IEEE Trans. Comput.*, 39(5):710–713, May 1990.

[27] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman, New York, 1979.

[28] M. D. Garris and R. A. Wilkinson. *NIST Special Database 3 : Handwritten Segmented Characters.* National Institute of Standards and Technology, February 1992.

[29] C. L. Giles, C. B. Miller, D. Chen, H. H. Chen, G. Z. Sun, and Y. C. Lee. Learning and extracting finite state automata with second-order recurrent neural networks. *Neural Computation*, 4:393–405, 1992.

[30] E. M. Gold. Language identification in the limit. *Inform. Control*, 10:447–474, 1967.

[31] E. M. Gold. System identification via state characterization. *Automatica*, 8:621–636, 1972.

[32] E. M. Gold. Complexity of automaton identification from given data. *Inform. Control*, 37:302–320, 1978.

[33] J. A. Goldman. Pattern theoretic knowledge discovery. In *6th International IEEE Conference on Tools with AI*, 1994.

[34] Jeffrey A. Goldman. Machine learning: A comparative study of pattern theory and C4.5. Technical Report WL-TR-94-1102, Wright Laboratory, USAF, WL/AART, WPAFB, OH 45433-6543, June 1994.

[35] Nelson Goodman. *Fact, Fiction and Forecast.* Harvard University Press, 1983.

[36] A. Grasselli and F. Luccio. A method for minimizing the number of internal states in incompletely specified sequential networks. *IRE Transactions on Electronic Computers*, EC-14(3):350–359, June 1965.

[37] James N. Gray and Michael A. Harrison. The theory of sequential relations. *Information and Control*, 9, 1966.

[38] G. Hachtel, J.-K. Rho, F. Somenzi, and R. Jacoby. Exact and heuristic algorithms for the minimization of incompletely specified state machines. In *The Proceedings of the European Design Automation Conference*, 1991.

[39] S.J. Hanson and L. Pratt. A comparison of different biases for minimal network construction with back-propagation. In D.S. Touretzky, editor, *Advances in Neural Information Processing Systems*, volume 1, pages 177–185, San Mateo, 1989. Morgan Kaufmann.

[40] D. Haussler, M. Kearns, and R. E. Schapire. Bounds on the sample complexity of Bayesian learning using information theory and the VC dimension. In *Proc. 4th Annu. Workshop on Comput. Learning Theory*, pages 61–74, San Mateo, CA, 1991. Morgan Kaufmann.

[41] G.E. Hinton and T.J. Sejnowski. Learning and relearning in Boltzmann machines. In D.E. Rumelhart and J.L. McClelland, editors, *Parallel Distributed Processing*, volume 1, chapter 7, pages 282–317. MIT Press, Cambridge, 1986.

[42] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading, MA, 1979.

[43] Alan Hutchinson. *Algorithmic Learning.* Oxford University Press, New York, 1994.

[44] N. Ishiura, H. Sawada, and S. Yajima. Minimization of binary decision diagrams based on exchanges of variables. In *ICCAD*, pages 472–475. IEEE Computer Society Press, 1991.

[45] J. S. Judd. *Neural Network Design and the Complexity of Learning.* MIT Press, 1990.

[46] T. Kam and R.K. Brayton. Multi-valued decision diagrams. *Tech. Report No. UCB/ERL M90/125*, December 1990.

[47] T. Kam, T. Villa, R. K. Brayton, and A. Sangiovanni-Vincentelli. A fully implicit algorithm for exact state minimization. Proc. Design Automat. Conf., 1994.

[48] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, pages 291–307, February 1970.

[49] S. Kirkpatrick, C.D. Gelatt Jr., , and M.P. Vecchi. Optimization by simulated annealing. *Science*, 220, 1983. Reprinted in [1].

[50] Ron Kohavi. Bottom-up induction of oblivious read-once decision graphs. In *European Conference in Machine Learning*, 1994.

[51] A. N. Kolmogorov. Three approaches to the quantitative definition of information. *Problems Information Transmission*, 1:1–7, 1965.

[52] A. Kramer. Optimization techniques for neural networks. Technical Report UCB-ERL-M89-1, UC Berkeley, Berkeley, CA, 1989.

[53] K. J. Lang. Random DFA's can be approximately learned from sparse uniform examples. In *Proc. 5th Annu. Workshop on Comput. Learning Theory*, pages 45–52. ACM Press, New York, NY, 1992.

[54] Y. Le Cun, B. Boser, J.S. Denker, D. Henderson, R.E. Howard, W. Hubbard, and L.D. Jackel. Handwritten digit recognition with a back-propagation network. In D.S. Touretzky, editor, *Advances in Neural Information Processing Systems*, volume 2, pages 396–404, San Mateo, 1990. Morgan Kaufmann.

[55] Y. Le Cun, J.S. Denker, and S.A. Solla. Optimal brain damage. In D.S. Touretzky, editor, *Advances in Neural Information Processing Systems*, volume 2, pages 598–605, San Mateo, 1990. Morgan Kaufmann.

[56] M. Li and P. M. B Vitányi. *An Introduction to Kolmogorov Complexity.* Addison-Wesley, MA, 1994.

[57] J. J. Mahoney and R. J. Mooney. Initializing ID5R with a domain theory: some negative results. Technical Report 91-154, CS Dept., University of Texas at Austin, Austin, TX, 1991.

[58] N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller, and E. Teller. Equation of state calculations for fast computing machines. *Journal of Chemical Physics*, 21:1087–1092, 1953.

[59] Paul L. Meyer. *Introductory Probability and Statistical Applications.* Addison-Wesley, 1965.

[60] T. M. Mitchell. The need for biases in learning generalizations. Technical Report CBM-TR-117, Rutgers University, New Brunswick, NJ, 1980.

[61] S. Muroga. *Threshold Logic and its Applications.* Wiley-Interscience, 1971.

[62] P. M. Murphy and D. W. Aha. *Repository of Machine Learning Databases - Machine readable data repository.* University of California, Irvine, 1991.

[63] Alan F. Murray and Anthony V. W. Smith. Asynchronous vlsi neural networks using pulse-stream arithmetic. *IEEE Journal of Solid-State Circuits*, 23:3:688–697, 1988.

[64] B. K. Natarajan. *Machine Learning: A Theoretical Approach.* Morgan Kaufmann, San Mateo, CA, 1991.

[65] Arlindo L. Oliveira. Logic synthesis using threshold gates. UC Berkeley EE290LS Class Report, 1990.

[66] Arlindo L. Oliveira and A. Sangiovanni-Vincentelli. Learning complex boolean functions : Algorithms and applications. In *Advances in Neural Information Processing Systems 6*, Denver, CO, 1993. Morgan Kaufmann.

[67] J. J. Oliver. Decision graphs - an extension of decision trees. Technical Report 92/173, Monash University, Clayton, Victoria 3168, Australia, 1993.

[68] Y. Le Cun P. Simard and John Denker. Efficient pattern recognition using a new transformation distance. In *Advances in Neural Information Processing Systems 5*, pages 51–58, Denver, CO, 1992.

[69] G. Pagallo and D. Haussler. Boolean feature discovery in empirical learning. *Machine Learning*, 5(1):71–100, 1990.

[70] Giulia Pagallo. *Adaptive Decision Tree Algorithms for Learning From Examples.* PhD thesis, UC Santa Cruz, 1990.

[71] M. Paull and S. Unger. Minimizing the number of states in incompletely specified state machines. *IRE Transactions on Electronic Computers*, September 1959.

[72] C. F. Pfleeger. State reduction in incompletely specified finite state machines. *IEEE Trans. Computers*, C-22:1099–1102, 1973.

[73] Vera Pless. *Introduction to the theory of error-correcting codes.* Wiley, New York, 1989.

[74] Jordan B. Pollack. The induction of dynamical recognizers. *Machine Learning*, 7:123–148, 1991.

[75] S. Porat and J. A. Feldman. Learning automata from ordered examples. In *Proc. 1st Annu. Workshop on Comput. Learning Theory*, pages 386–396, San Mateo, CA, 1988. Morgan Kaufmann.

[76] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

[77] J. R. Quinlan. An empirical comparison of genetic and decision-tree classifiers. In *Fifth International Conference on Machine Learning*, pages 135–141, 1988.

[78] J. R. Quinlan. *C4.5 - Programs for Machine Learning.* Morgan Kaufmann, San Mateo, 1993.

[79] J. R. Quinlan and R. L. Rivest. Inferring decision trees using the Minimum Description Length Principle. *Inform. Comput.*, 80(3):227–248, March 1989. (An early version appeared as MIT LCS Technical report MIT/LCS/TM-339 (September 1987).).

[80] J. Rissanen. Modeling by shortest data description. *Automatica*, 14:465–471, 1978.

[81] J. Rissanen. Stochastic complexity and modeling. *Ann. of Statist.*, 14(3):1080–1100, 1986.

[82] R. Rudell and A. Sangiovanni-Vincentelli. Multiple-valued minimization for PLA optimization. *IEEE Transactions on Computer-Aided Design*, September 1987.

[83] Richard Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *ICCAD*, pages 42–47. IEEE Computer Society Press, 1993.

[84] D.E. Rumelhart, G.E. Hinton, and R.J. Williams. Learning internal representations by error propagation. In D.E. Rumelhart and J.L. McClelland, editors, *Parallel Distributed Processing*, volume 1, chapter 8, pages 318–362. MIT Press, Cambridge, 1986. Reprinted in [1].

[85] K. Hamaguchi S. Tani and S. Yajima. The complexity of the optimal variable ordering problems of shared binary decision diagrams. In *Algorithms and Computation, 4th Internation Symposium*, pages 389–98, 1993.

[86] Eduard Säckinger, Bernard Boser, and Lawrence D. Jackel. A neurocomputer board based on the anna neural network chip. In *Advances in Neural Information Processing Systems 4*, pages 773–780. Morgan Kaufmann, 1992.

[87] Hamid Savoj. *Don't Cares in Multi-Level Network Optimization.* PhD thesis, UC Berkeley, 1992.

[88] Cullen Schaffer. A conservation law for generalization performance. In *Proceedings of the Eleventh International Conference in Machine Learning*, San Mateo, 1994. Morgan Kaufmann.

[89] R. E. Schapire. *The Design and Analysis of Efficient Learning Algorithms.* MIT Press, Cambridge, MA, 1992.

[90] Alen D. Shapiro. *Structured Induction in Expert Systems.* Addison-Wesley, 1987.

[91] Thomas R. Shiple, Ramin Hojati, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli. Exact minimization of BDDs using don't cares. Private communication, 1993.

[92] Thomas R. Shiple, Ramin Hojati, Alberto L. Sangiovanni-Vincentelli, and Robert K. Brayton. Heuristic minimization of BDDs using don't cares. In *Proc. Design Automat. Conf.*, pages 225–231, San Diego, CA, June 1994.

[93] K. Y. Siu and J. Bruck. Neural computation of arithmetic functions. *Proceedings of IEEE*, 10:1669–1675, 1990.

[94] K. Y. Siu and J. Bruck. On the power of threshold circuits with small weights. *SIAM Journal of Discrete Math*, page to appear, 1994.

[95] R.J. Solomonoff. A formal theory of inductive inference, part 1 and part 2. *Information and Control*, 7:1–22, 224–254, 1964.

[96] R. Spickelmier, editor. *Oct Tools Distribution 2.1.* University of California, Berkeley, March 1988.

[97] Yasuhiko Takenaga and Shuzo Yajima. NP-completeness of minimum binary decision diagram identification. Technical Report COMP 92-99, Institute of Electronics, Information and Communication Engine ers (of Japan), March 1993.

[98] S. B. Thrun, J. Bala, E. Bloedorn, I. Bratko, B. Cestnik, J. Cheng, K. de Jong, S. Dzeroski, S. E. Fahlman, D. Fisher, R. Hamann, K. Kaufaman, S. Keller, I. Kononenko, J. Kreusiger, R. S. Michalski, T. Mitchell, P. Pachowitz, Y. Reich, H. Vafaic, W. Van de Weldel, W. WEnzel, J. Wnek, and J. Zhang. The monk's problems: a performance comparison of different learning algorithms. Technical Report CMU-CS-91-197, Carnegie Mellon University, 1991.

[99] M. Tomita. Dynamic construction of finite-state automata from examples using hill-climbing. In *Proc. Fourth Annual Cognitive Science Conference*, page 105, 1982.

[100] B. A. Trakhtenbrot and Y. M. Barzdin. *Finite Automata.* North-Holland, Amsterdam, 1973.

[101] D. S. Yang, L. Rendell, and G. Blix. Fringe-like feature construction: A comparative study and a unifying scheme. In *Proceedings of the Eight International Conference in Machine Learning*, pages 223–227, San Mateo, 1991. Morgan Kaufmann.